Heap Data Management for Limited Local Memory (LLM) Multi-core Processors

Ke Bai
Compiler Microarchitecture Lab
Arizona State University
Tempe, AZ
Ke.Bai@asu.edu

Aviral Shrivastava
Compiler Microarchitecture Lab
Arizona State University
Tempe, AZ
Aviral.Shrivastava@asu.edu

ABSTRACT

This paper presents a scheme to manage heap data in the local memory present in each core of a limited local memory (LLM) multi-core processor. While it is possible to manage heap data semi-automatically using software cache, managing heap data of a core through software cache may require changing the code of the other threads. Cross thread modifications are difficult to code and debug, and only become more difficult as we scale the number of cores. We propose a semi-automatic, and scalable scheme for heap data management that hides this complexity in a library with a much natural programming interface. Furthermore, for embedded applications, where the maximum heap size can be known at compile time, we propose optimizations on the heap management to significantly improve the application performance. Experiments on several benchmarks of MiBench executing on the Sony Playstation 3 show that our scheme is easier to use, and if we know the maximum size of heap data, then our optimizations can improve application performance by an average of 14%.

Categories and Subject Descriptors

D.3.4 [**Software**]: Programming languages, Processors–Code generation, Compilers, Optimization

General Terms

Algorithm, Design, Experimentation, Performance

Keywords

Heap, local memory, scratch pad memory, embedded systems, multi-core processor, IBM Cell, PS3, MPI

1. INTRODUCTION

As we transition from multi-core (few cores) to many-core (hundreds of cores) architectures, scaling the memory architecture is one of the most important issues. Maintaining the illusion of a single unified memory architecture in hardware is becoming too expensive. This is because of two main reasons. First is, that the power and performance overheads of automatic memory management in hardware, i.e. by caches is becoming prohibitive. Caches consume about half of the processor energy on single-core processor [8], and are expected consume much larger fraction with increase in number of cores. The second reason is that cache coherency protocols do not scale to hundreds and thousands of cores [12]. Limited local memory (LLM) architectures, in which each core has a small local memory, is coming up as a promising scalable memory architecture for multi-cores. Modern and futuristic processors, especially in the embedded domain are already being designed in LLM architecture. Examples are the experimental 80-core Intel processor [29], and the Cell architecture from IBM [13].

In a LLM processor, each core has access to only a small local memory, and transfers between the global memory and the local memory have to be explicitly specified in the application code. For example, the Synergistic Processing Elements (SPEs) in the IBM Cell processor can only access a local 256 KB memory, and all code and data that the SPE uses must reside in the local memory. There are two closely coupled challenges in developing applications for such architectures. First is to parallelize the given application. This is typically done by humans, who (re-) write the application in a threaded manner. The second is to efficiently execute each thread on a core. If a core is unable to execute the thread mapped to it, then the programmer must change the way application is parallelized. This can be extremely hard, because often applications have some natural ways to parallelize them, and finding out other ways to parallelize can be formidable. Therefore this paper primarily deals with the second challenge of making a thread of application execute (and efficiently execute) on a core.

If all the application code and data can fit into the local memory, extremely efficient execution is achieved – and in fact, this is the promise of LLM architecture. However, if the application code and data does not completely fit into the local memory, then explicit commands to bring the required data in the local memory before it is used, and move the not-needed data back to the global memory must be inserted into the application.

While management is needed for all code and data on the local memory, managing heap data is especially important, since it is dynamic in nature and may not be known at compile time. Heap data can easily overwrite stack data and cause several kinds of program failure ranging from a ap-

CODES+ISSS'10, October 24–29, 2010, Scottsdale, Arizona, USA.

plication crash, going into infinite loop, or simply a wrong result. If a program is not recursive then only heap data is dynamic, and may therefore cause a failure. In fact, the Cell Programmer's Guide[2] suggests to "avoid using heap variables". We believe, this is extremely limiting to programmer's creativity and productivity, and a scheme to efficiently manage application heap data in a constant and small amount of memory on the local memory is needed.

One way to semi-automatically manage heap data in the local memory in the core of a LLM processor is through the use of software cache [5]. Software cache is essentially a cache implemented in software, using data structures. Software cache is best suited to manage global data, which is declared and allocated once. However, since heap data is allocated dynamically in the application, managing the heap data of an application thread using software cache requires several changes in the application code. Not only it requires changing the code of the thread in some non-intuitive ways, it also requires changing the thread on the main core. In fact, the user must create a new thread on the main core. This solution is not only difficult to implement and debug, but also becomes more complex as we scale the number of cores.

This paper proposes to hide the programming complexity in a library with simple programming interface. We enhance the GCC compiler for the IBM Cell with this library, compile benchmarks from MiBench [14] and others using it, and then measure the runtime on Sony Playstation 3. Our experiments show that our heap management scheme, while being simple and natural for the programmer, performs at 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%.

2. LLM PROCESSORS

2.1 Limited Local Memory Architecture

The IBM Cell Broadband Engine[13] is a very good example of a limited local memory (LLM) architecture. As shown in Figure 1, it is a 9-core processor, with one *main core* (the Power Processing Element, or PPE in Cell) and eight small *execution cores* (the Synergistic Processing Elements, or SPEs in Cell). The main core in the Cell processor is a 2-way Simultaneous Multi-threaded Power 5 core, while each of the execution cores work on only one thread at a time in a non-pre-emptive fashion. Only the main core has Operating System, and it has direct access to the global memory through a coherent L2 cache, while each execution core has a 256 KB local store memory. Data communications between the local memory on the execution core and the global memory should be explicitly managed in the software through direct memory access (DMA) engine. The interconnect bus is 128-byte wide, with more than 300 GB/s capacity, and has over 100-deep request queue.

2.2 Thread-based Programming Paradigm

Programming on an LLM architecture is based on Message passing interface (MPI) style *thread model*. It requires programmers to have a *main thread*. This main controller thread creates, distributes data and tasks and even collect results from the *execution threads*. The main thread runs on the main core, while the execution threads are scheduled on the execution cores. A very simple application in this multi-

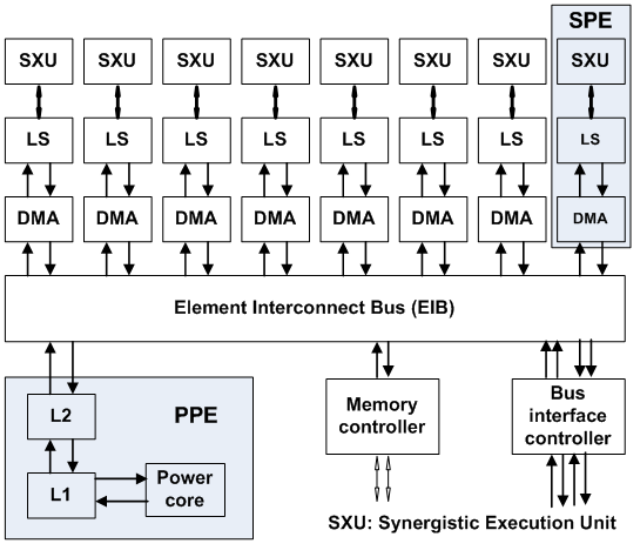

Figure 1: The IBM Cell/B.E. is a good example of limited local memory (LLM) architectures. There are 8 Synergistic Processing Elements, or SPEs. Each SPE can access only a small local memory, and all data transfers between the local memory and the global memory take place through explicit DMA calls.

core programming paradigm is illustrated in Figure 2. In the pseudo code, the main thread, executing on the main core initiates several execution threads on the execution cores. In the execution core thread student data structures are initialized, and operated on. Student data structure contains two fields, *id* (int) and *score* (float) for each student.

2.3 Need of Heap Management

Normally, the local memory on the execution core is divided into three segments by the software: the text region (program code and data), heap variable region and stack variable region [3]. The text region is where the compiled code of the program itself resides. The function frames reside in the stack space, which starts from the top of the memory, growing downwards, while the heap variables, (defined through *malloc*) are allocated in the heap region, starting from the top of the code region, and growing upwards. The three segments share the local store, and since local store is a constrained resource and 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 fine, but large values of N can cause catastrophic failures, e.g., application crash, execution core going into an infinite loop. However, even worse is when output is just subtly incorrect and that too only sometimes. 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 productivity of the programmer. What is needed is a scheme that limited local memory multi-core programmers can use to efficiently and intuitively manage heap memory of the application.

```
main(){
  for (i=0; i<6; i++) {
    pthread_create ( ...spe_context_run(speID) ...);
  }
}
```

(a) PPU Code

```
typedef struct {
  int id;
  float score;
} Student;

main(){
for (i=0; i<N; i++)  {
    student[i] = malloc(sizeof(Student));
    student[i].id = i;
    printf("%d\n", student[i].id);
  }
}
```

(b) SPU Code

Figure 2: Outline of a threaded program on the Cell processor: (a) PPU creates threads on each SPU. (b) On each SPU some student records are allocated and accessed.

3. RELATED WORK

Local memories in each core of an LLM are raw memories, not managed in hardware; they are software controlled. They are very similar to the Scratch Pad Memories (SPMs) popular in embedded systems. Banakar et. al [8] 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 [22]. They demonstrated that for the same memory area, SPMs consume 40% less energy and 34% less die area. However, the absence of the memory management logic in the hardware shifts the burden of managing data on the SPM to the programmer.

Techniques have been proposed to manage code [27, 5, 32, 23, 31, 26, 10, 11, 28, 15, 30, 24], global data [27, 6, 31, 18, 17, 20, 28, 30] and stack data [6, 23, 25, 20, 28], on the SPM, but little work has been done towards managing heap data [25, 21, 9].

We have previously proposed code management techniques [24, 16] and stack management scheme [19] for the local memory of LLM processors, e.g. Cell processor. This work would focus on managing heap data on local memories of LLM processors, and fundamentally differs from the existing work on SPMs. The difference originates from the usage models of SPMs in embedded systems and local memories in LLM systems. Figure 3 illustrates the difference. It shows that in the embedded systems e.g. the ARM architecture, the SPM was 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, 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 in distributed memory multi-core processors is to enable the execution of application. As a result, previous SPM researches [25, 21, 9] have focused on the question of "what to map" on the SPM. The "what to map" is not even an option for LLM processors. Important questions are in using local memories are: i) Given that the application code has to be manually changed to make applications work, what will be a set of intuitive and simple changes that must be made to the application program to make this happen, and ii) These changes should eventually result in less number of coarse-grain communications between the local memory and the global memory.

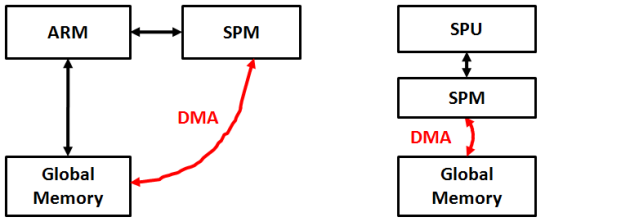

Figure 3: In the ARM processor, the SPM was in addition to the regular cache hierarchy, while in the Cell/B.E., local store is an essential part of the memory hierarchy.

One way to manage heap data on the local memory of each core in a LLM processor, e.g. the IBM Cell is by using *software cache* [7]. However, there are several limitations in managing heap data through software cache. We discuss this approach and it's limitations in greater detail in the next section, Section 4, and then describe our approach to meet this challenge in Section 5, and then finally experimentally demonstrate the need and usefulness of our approach in Section 7.

4. HEAP DATA MANAGEMENT THROUGH SOFTWARE CACHE

The Software Cache is a semi-automatic way to manage large amounts of data in a constant amount of SPM/local memory space [5]. Software cache data structure is located in the execution core with a programmer-defined size in the global data segment. In order to use software cache, the programmer has to first declare that they intend to manage certain data structure through software cache, and then manually replace every access of that data by a read/write from the software cache. Software cache access first checks whether the data is in the cache data structure on the local memory or not. If it is, then the program can directly read/write the data from/to the cache, otherwise, a Direct Memory Access (DMA), is performed to get the required data from the global memory to the local memory, and then it can be used. As new data comes in the cache data structure, older data may be evicted out to the main memory.

Figure 4 gives an example of how the heap data of the application described in Figure 2 may be managed through software cache. Note that there is no published scheme to manage heap data using software cache, and we have thought of this technique as one way to manage heap data using software cache. First let us take a look at the execution

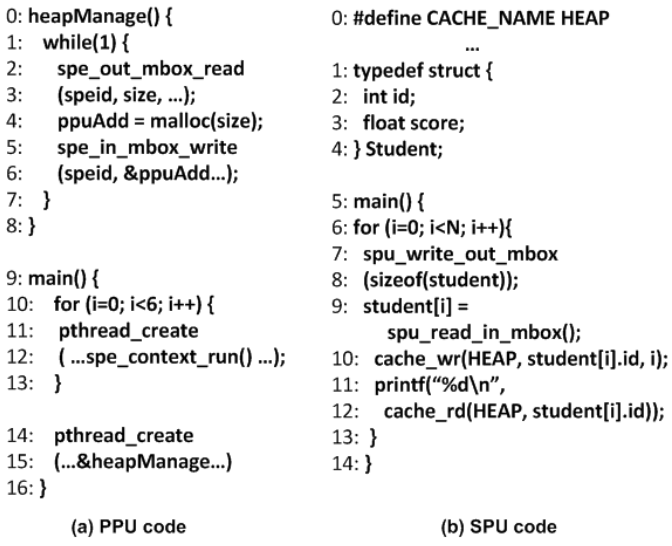

Figure 4: Using software cache to manage heap data in the Cell processor. Important observations: i) The SPU must transmit all its memory management functions to the PPU, ii) Need an extra memory management thread on the PPU.

thread in Figure 4 (b). The first line shows the declaration of the software cache named HEAP. More declaration statements are needed, but are skipped here for clarity. Since the number of students is unknown and can be large, the student data structures must be allocated in the global memory. However, in the original code, the student data structures are *malloc-ed* in the execution thread. Therefore, we need a way to communicate memory requirements from the execution threads/cores to the main thread/core. This requires a change in the structure of the multi-threaded program. 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 student data structure in the global memory and sends its address back to the execution thread/core (SPE). The execution core (SPE) uses this address to access the student data structure that actually resides in the global memory, through software cache. To enable this scheme, we need a new thread on the main core (PPE), *heapManage*, which waits for requests from the execution thread/core (SPE), allocates the requested data structure in global memory heap, and sends back the allocated address to the execution thread/core (SPE). Similar steps need to be taken when *free-ing* up the allocated memory, but are skipped for simplicity in the example. Some of the complexities in managing heap data through software cache are:

1. The interface of the software cache requires that the data should be allocated on main core, and the execution cores must access the data using the global address. To use software cache, if an execution thread/core allocates/frees certain variables (using malloc/free), then these allocation requests must be transmitted to the main core. Users have to program this communication and allocation/free manually. In addition, to enable that main core handle the execution thread memory management requests, users have to manually create a new thread, which will wait and serve requests from execution threads. Another interesting aspect of this communication is that normally the execution cores do the bidding of the main core, but to support this heap management the main core has to serve execution core requests. This reversal of roles makes this programming non-intuitive and complicated.

```
typedef struct node
{
    int weight;
    //struct node *link;
    int link;
};
```

Figure 5: Need to change pointer to any other non-pointer/non-structure type for use in software cache, here we change it to *int*

2. Software cache library only supports one data type in a cache. Consider the example of a data structure shown in Figure 5. The figure declares a node with a weight and a pointer to a similar node. Software cache does not support this pointer element, and it must be renamed as any other non-structure and non-pointer data type. This has to be done because the *weight* is *int*, hence we change it to integer for the purpose that the two element can use one cache instead of two different caches. This is un-natural for C programming, and severely reduces readability.

3. Even if we know 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 and therefore there is little scope for optimization on the management overhead.

5. OUR APPROACH

5.1 Overview

The objective of our approach is to hide the additional complexity in managing heap memory in a limited space on the local memory in a library, that is intuitive to use, and requires minimal changes in the program. In our technique, programmers do not need to worry about the data type for their heap variables. Figure 6 shows the pseudo-code of how to use our heap management on the example shown in Figure 2. Note that the heap is declared and allocated/free-ed only on the execution thread/core (SPE). User does not need to write an extra thread on the main core (PPE) for heap data management. In fact, the main thread does not change at all. Programmers 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. The only change the user needs to make is to add the functions *p2s* and *s2p* before and after any access to a heap variable. These modifications are a proper subset of what is required to manage heap data through software cache, do not change the structure of a multi-threaded program, and are easy for the programmer. We expose the

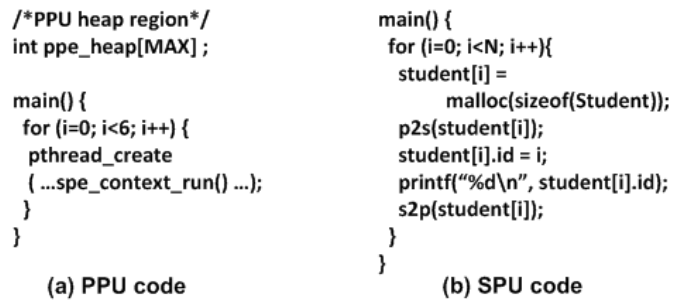

Figure 6: Using our approach to manage heap data. We redefine the *malloc* and *free* on the SPE to automatically interact with the PPE. The only changes required from the original multi-threaded program is that the user has to add a call to *p2s* function before and *s2p* function after accessing a heap variable.

global address and local address to programmers, since we do not need to perform checking every time as one disadvantage of using software cache.

5.2 Application Programming Interface (API)

The fundamental problem in Limited Local Memory (LLM) architecture is that a program variable can have two addresses, depending on the memory in which it is located, and that unlike in cache-based architectures, the program must access the variables by correct address. The software cache implementation hides the address of the variable in the local memory. It exposes only the global address of the variable and lets the programmer use only that. While this keeps programming very much like that in cache-based architectures, it however requires address translation every single time the variable is accessed, and results in high overhead. Our heap management approach exposes the local address of the variable to the programmer, so that the programmer can use it directly, and not perform the address translation every time. The function *p2s(global address ga)* brings the program variable at global address *ga* to the local memory, if it is not already there, and returns the local address, *la* of the variable. The counterpart functionality is encapsulated in the function *s2p(local address la)*. In addition to these two new implemented functions, the API of our heap data management approach also redefines two existing functions. If there is enough memory space in the heap region defined in the local store, the *malloc* function returns a pointer to it, otherwise it evicts older heap variable(s) to global memory to make sufficient space for this heap variable, and returns a pointer to it. One important point to note here is that even though the *malloc* function may allocate space on the local memory, it returns the global memory address of the allocated heap variable. This is so that we always use the global address to access the heap variables, including when writing them in data structures, e.g., linked list. Arbitrary sized linked lists cannot work with local addresses of heap variables. The *free* function also takes in the global address of the variable.

5.3 Global Memory Management

In order to support almost (unlimited) heap memory, we have to manage the heap data and the heap management table in the main 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 4, however, this separate thread is a part of the library in our implementation, and the user does not have to explicitly code it. 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. Consider the SPU code in Figure 2. The program allocates a student data structure, and then accesses one field (student.id) of it. When the program accesses any part of a allocated data structure, if the whole data structure is brought into the local memory, then the heap management is done at programmer defined granularity. If only the exact field, e.g., the integer field of student.id is brought into the local memory, then the heap management is being done at word level of granularity. A finer granularity of heap management is beneficial, if the allocated data structures are large, and only a small part of them are used in the algorithm. Finally, heap management is performed at a coarse granularity by grouping the allocated objects in a block, and if a part of any of them is accessed, then a whole block of them are brought into the local memory. This is very effective when the allocated objects are small. One important advantage of 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.

5.4 Local Heap Management

In order to manage unlimited heap data in a limited space on the local memory, we need to keep a mapping of global to local addresses. This data structure is called the heap management table. The local memory space for heap management S is divided into a constant space required for heap data H, and a constant space required for heap management table, T, such that $S = H + T$. A *malloc* may add an entry to this table, and a *free* may result in the removal of an entry in the heap management table. The table is accessed at every call to *p2s* and *s2p* functions. Since this number can be large, one is tempted to maintain this table in the local memory. However, the size of the table can also grow arbitrarily large. Therefore, we only maintain a part of this table in the local memory. All the sizes, S, H, and T are fixed at compile time. Every time when we want to add an entry we check if there is place to write the new entry. If yes, then we can just write the new entry, else we can write the new entry after making space for a new entry by evicting some of the older entries to the main memory. The number of entries we evict at a time is the granularity of management, and the heap table management can also be performed at several granularities, from a single element to the entire table size. We leave the exploration of the effect of the granularity of the management of the heap management table as a future concern. In this work we manage the heap management table at the whole table size granularity. Thus we evict the whole table, and bring a full table back into the local memory, when needed.

6. OPTIMIZATION FOR EMBEDDED SYSTEMS

We implemented the heap data structures and the heap management table in the main global memory using dynamic data structures. While this is clearly needed to support (almost) unlimited heap data, it comes with high performance penalty. In order to do this, the *malloc* function must eventually be mapped to insert operation in the dynamic data structure on the main core. Fundamentally this requires some communication between the local and the main thread, which can interpret messages from the local thread and translate them as inserts in the data structure in the main thread. In the Cell processor this can be achieved through another thread on the main processor and a mailbox-based communication between the execution cores and the main core. This communication is in addition to the actual heap data that has to be transferred between the local and the main core. Clearly this has high overheads.

In embedded systems, where it may be possible to define a upper bound on the heap memory needed, several optimizations can be done to minimize the overheads. If the maximum heap size (i.e., assuming no free's) is known at compile time, we can operate profiling to keep this size. Then the heap data structure and heap management table can be declared as static data structures, and we can allocate all the heap variables contiguously in the pre-defined space. When a heap data is needed, we can resolve the address in the execution core so that a direct memory access (DMA) can be directly used to transfer the data between the local and the global memory. This completely eliminates the need for the extra thread in the main core, and therefore avoids all the performance penalty associated with the slow mailbox-style communication. Furthermore, if possible, and especially because the heap management table entries may be much smaller than the heap data, the whole heap management table may be housed in the local memory, resulting in additional performance optimization.

7. EXPERIMENTS

7.1 Experimental Setup

We conduct our experiments on IBM Cell processor in Sony Play Station 3. The system runs a Linux Fedora 9 [1], that gives us access to 6 of the 8 SPEs. We implement our scheme on Mibench suite [14]. Note that all these benchmarks are single threaded. We have modified them to perform all the input/output in the PPE, and perform all the execution in the SPEs. We evaluate the effectiveness of our heap management technique and optimization by comparing the runtimes of (i) benchmarks without any heap management (baseline) (ii) benchmarks with heap management to support arbitrary heap data size and (iii) benchmarks with heap size optimizations done. We use *_mftb()* for measuring the runtime of PPE and *spu_decrementer()* for measuring the runtime of SPE [4]. Since we compute the runtimes in the presence of operating system, we run each experiment 10 times, and use the average runtime as our measure of runtime. The benchmarks implemented are detailed in Table 1. *dijkstra, fft, fft_inv*, and *stringsearch* are from the Mibench suite [14], *DFS*, *MST* and *red black tree* are some other algorithms that are likely to be used in the application domain intended for the Cell processor. The heap size requirements for all the benchmarks is also noted in Table 1. Most of our experiments are on this configuration of one PPE and only one SPE. However, in our last experiment on scalability, we create multiple threads of same computation on various number of cores.

Benchmarks	Description	Heap Size (bytes)
Dijkstra	find shortest path	5040
fft	fft algorithm	16416
fft_inv	fft_inv algorithm	16416
stringsearch	search strings	4096
DFS	depth first search	16000
MST	minimum spanning tree	336
rbTree	red black tree	2476

Table 1: We choose several benchmarks from MiBench and elsewhere that use heap variables. The table also shows the maximum heap data each application needs.

7.2 Unrestricted Heap Size

To demonstrate the need of our technique, we execute the *rbTree* benchmark *with* and *without* heap management. The red black tree is a binary search benchmark with each node in the tree data structure using 24 bytes. Each node is dynamically allocated and thus uses heap. In the benchmark , the code and global data occupy 15312 bytes in total. The rest of the space is shared between the heap and the stack data. Without any heap management, we can allocate only $n_0 = 6800$ nodes (almost 160 KB) without any heap management, exceeding which causes the program to crash.

We run the benchmark using our scheme with nodes from 1 to 65536, which is nearly 10 times larger than that can be run without heap management. We make an initial allocation in the local memory to manage heap data of 150 KB. Hence no heap data DMA happens between the global memory and the local memory until the 150 KB region is full. Furthermore, we set the number of entries in the heap management table as 256, which consumes about 4 KB. These parameters are chosen for fair comparison of time *with* and *without* heap management techniques.

The first observation from Figure 7 is that our technique does not seem to have restrictions on the heap size of the application. Both the heap management table, and memory allocation in the global memory is dynamically managed. The runtime increases when we use our management scheme, because DMA needs to be performed for the management of the heap data and heap management table. This is also the reason why there is a leap after we allocate more than 6800 nodes. However, our technique guarantees that the application can be run for any program parameters, without any further changes by the user.

7.3 Impact of Heap Management Parameters

The runtime of applications running with our heap management are most affected by two parameters: First is the amount of space in the local memory that is used for storing heap data and the heap management table, and the second is the granularity that we choose for heap data management.

The total memory space for heap (S) can be partitioned as $S = H + T$. Let n_H be the number of heap chunks that

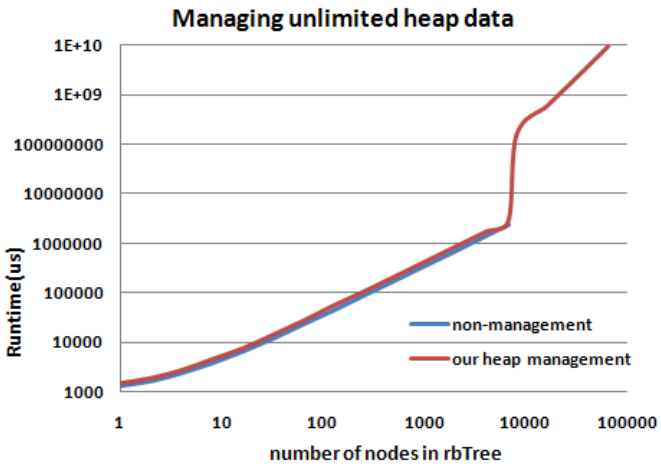

Figure 7: Without heap management, the program will run with at most 6800 nodes; with management we get a flexibility of using applications that require larger heap.

can be allocated in the fixed heap region. Hence, the total space for heap data should be $H = n_H * s_H$, where s_H is the size of one heap chunk. Also, $T = n_E * s_E$, where n_E is the number of entries in the local store and s_E is the size of one entry. For a given S, there are several ways to partition the memory into H and T, and how to partition the memory is a topic of further investigation, but here, we assume $n_H = n_E$, and thereby get a unique partition of S into H and T. For each benchmark, we compute the minimum and the maximum heap size that is required. We execute applications for the entire range of the heap sizes, and plot the runtimes in Figure 8. The most clear message from this graph is that the runtime of the application improves as we increase the heap data region size on the local memory.

The second major effect on heap management is due to the granularity of transactions between the local memory and the global memory. We tune the granularity in multiples of the memory blocks that are malloc-ed. For all our

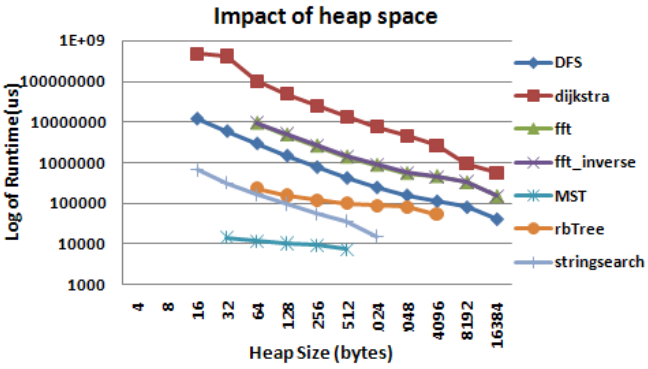

Figure 8: Heap management with dynamic memory allocation in *global memory* of PPE for heap objects from *local store* of SPE, the heap region size for each benchmark is set from the minimum size to the maximum size.

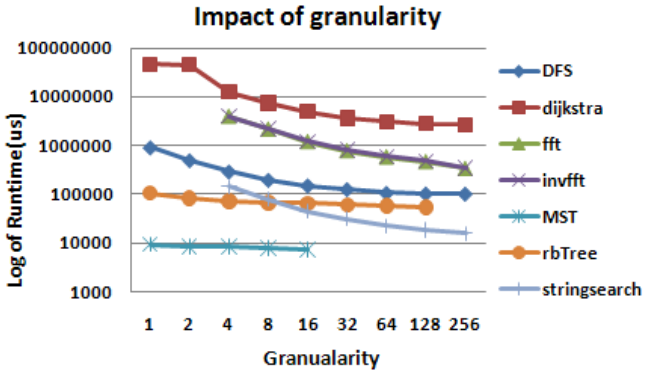

Figure 9: Benchmarks with different granularities and size for heap region is set to be 4096 bytes. Granularity defines the number of heap variables we consider for heap data DMA at a time.

Benchmarks	Dynamic management (us)	Static management (us)	Average Improvement
Dijkstra	48167280	39317728	18.4%
fft	2837261	2708675	4.5%
fft_inv	2923532	2788252	4.5%
stringsearch	130029	127650	1.8%
DFS	1000519	957732	4.3%
MST	9909	5615	43.3%
rbTree	98044	77376	21.1%

Table 2: Average performance enhancement for each benchmark with different heap region size and different granularity is 14% . If a static buffer can be assigned in the memory for heap management, it eliminates the blocking calls caused due to dynamic memory manager thread in the PPE.

benchmarks, we set heap size as 4096 bytes. From Figure 9, the effect of granularity is consistent across most benchmarks, where, for a given heap space on the local memory, the runtimes decrease as we increase the granularity.

7.4 Optimization for Embedded Applications

For extremely embedded applications, where we can get the maximum heap size of the application or thread by profiling, we can improve the application runtime by, first allocating sufficient space in the global memory so that there will be no need of dynamically allocating memory in the PPE. This helps, because dynamic memory allocation in the PPE requires communication between the SPE and the PPE through mailbox, which takes much more time than a direct memory access. If there is enough memory in the local store, then increasing the heap space in the local store to as high as possible, and increasing the granularity also helps. We exploit these techniques for each benchmark, and note the initial, and improved runtime in Table 2. From Table 2, we see that benchmarks have average performance improvement by 14.0%.

7.5 Scalability of Heap Management

To illustrate the scalability of our technique, we execute

identical benchmark on different number of cores. We run all benchmarks in the extreme case with dynamic memory management in the global memory of PPE (least size for heap region in the local store for heap variables), which would have worst performance.

In the Figure 10, we see that the runtime increases as we scale the number of cores. This is because of the competition of DMA request and mailbox use from different cores. There is a gradual increase in runtime for all benchmarks as we scale the number of cores. The increase is larger in case of *rbTree* and *MST*. The reason for the increase is the "not-so-frequent" localized read/write operations like the rest of the benchmarks. There is a scattered heap access which causes frequent access to the global memory increasing the latency to serve each access with increasing number of cores.

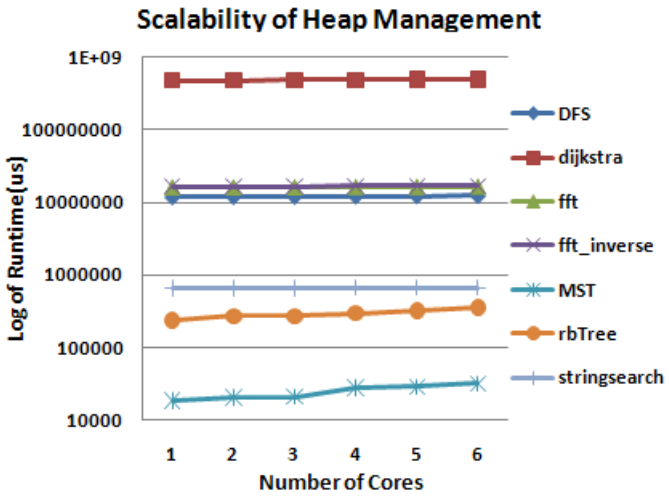

Figure 10: Benchmarks with dynamic memory management in global memory of PPE, configuring with different number of cores

8. SUMMARY AND FUTURE WORK

As we scale the number of cores, scaling the memory architecture becomes the bottleneck. Limited local memory (LLM) architectures are a scalable memory architecture platform that is now popular in modern and futuristic embedded processors, e.g., the IBM Cell. Such processors feature a software controlled local memory in each core. If all the application code and data fit into the local memory of the core, the execution is extremely efficient; however when this is not the case, the application code and data must be managed between the local and the global memory by explicitly inserting DMA commands in the application. While management is needed for all data, it is extremely important to manage heap data since it can easily overwrite other data and cause a failure. One possibility of managing heap data is through the use of software cache, however, it requires users to modify the thread code as well as the main thread code, which not only can be counter-intuitive and laborious, but also error-prone. We propose to automate heap memory management by providing an simple to use programming interface, which consists of redefinition of *malloc* and *free*, and introducing two new functions *p2s*, and *s2p*, which have to be added before and after every heap pointer access. Our experiments on the Sony Playstation 3, that features the IBM Cell demonstrates that i) our technique can support any amount of heap data, ii) it is intuitive and easy to use, and iii) it scales well with number of cores; in specific a single memory management thread on the PPE can service the needs of all the SPE memory requests. Finally we also show that, in extremely embedded systems, the heap management can be optimized to further improve the runtime of the applications by an average of 14%.

Our work foresees improvement in the following approaches. Firstly , the number of table entries is the same as the number of heap objects in the Local Store of SPE. Actually, given a total space for heap variables, we can partition it to heap management table and heap variables to optimize the total DMA transfer between global memory and local store. Secondly, we can reduce the calls to *p2s* and *s2p* functions before/after each heap variable by predicting if the variable will need frequent access again at a later stage. This can be improved further by doing a flow analysis using the control flow graph. Finally, we can take the advantage of prefetching and double buffering technique to reduce the runtime needed for the DMA.

As multi-core architectures with limited local memories become popular, there is an increasing need to run not-so-embedded applications on such architectures. We are addressing an important challenge of executing applications on such architectures without modification of the natural way of programming, with a goal to improve the programmability.

9. ACKNOWLEDGMENT

This research was partially funded by grants from National Science Foundation CCF-0916652, IIP-0856090, NSF I/UCRC for Embedded Systems, Microsoft Research, SFAz, Raytheon and Stardust Foundation. The authors would also like to thank members of the Compiler Microarchitecture Lab for their valuable critique.

10. REFERENCES

[1] *"Fedora 9 Project"*. http://fedoraproject.org/wiki/Releases/9.

[2] Programmer's guide:software development kit for multicore acceleration version 3.1. Technical report.

[3] Programming tutorial:software development kit for multicore acceleration version 3.1. Technical report.

[4] Spu c/c++ language extensions. Technical report.

[5] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri. A post-compiler approach to scratchpad mapping of code. In *Compilers, architecture, and synthesis for embedded systems*, pages 259–267, New York, NY, USA, 2004. ACM Press.

[6] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Trans. on Embedded Computing Sys.*, 1(1):6–26, 2002.

[7] J. Balart, M. Gonzalez, X. Martorell, E. Ayguade, Z. Sura, T. Chen, T. Zhang, K. O'Brien, and K. O'Brien. A novel asynchronous software cache implementation for the cell-be processor. pages 125–140, 2008.

[8] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design

alternative for cache on-chip memory in embedded systems. In *CODES'02:Hardware/software codesign*, pages 73–78, New York, NY, USA, 2002. ACM Press.

[9] A. Dominguez, S. Udayakumaran, and R. Barua. Heap data allocation to scratch-pad memory in embedded systems. *Embedded Computing*, 1(4):521–540, 2005.

[10] B. Egger, C. Kim, C. Jang, Y. Nam, J. Lee, and S. L. Min. A dynamic code placement technique for scratchpad memory using postpass optimization. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 223–233, New York, NY, USA, 2006. ACM.

[11] B. Egger, J. Lee, and H. Shin. Scratchpad memory management for portable systems with a memory management unit. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 321–330, New York, NY, USA, 2006. ACM.

[12] A. Eichenberger et al. Using advanced compiler technology to exploit the performance of the cell broadband enginetm architecture. *IBM Syst. J.*, 45(1):59–84, 2006.

[13] B. Flachs, S. Asano, S. Dhong, H. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H.-J. Oh, S. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, N. Yano, D. Brokenshire, M. Peyravian, V. To, and E. Iwata. The microarchitecture of the synergistic processor for a cell processor. *IEEE Solid-state circuits*, 41(1):63–70, 2006.

[14] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. *WWC-4: IEEE International Workshop on Workload Characterization*, pages 3–14, 2 Dec. 2001.

[15] A. Janapsatya, A. Ignjatović, and S. Parameswaran. A novel instruction scratchpad memory optimization method based on concomitance metric. In *ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation*, pages 612–617, Piscataway, NJ, USA, 2006. IEEE Press.

[16] S. c. Jung, A. Shrivastava, and K. Bai. Dynamic code mapping for limited local memory systems. In *ASAP '10: Proceedings of the 2010 International Conference on Application-specific Systems, Architectures and Processors*, 2010.

[17] M. T. Kandemir, J. Ramanujam, and A. N. Choudhary. Exploiting shared scratch pad memory space in embedded multiprocessor systems. In *DAC*, pages 219–224, 2002.

[18] M. T. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In *DAC*, pages 690–695, 2001.

[19] A. Kannan, A. Shrivastava, A. Pabalkar, and J.-e. Lee. A software solution for dynamic stack management on scratch pad memory. In *ASP-DAC '09: Proceedings of the 2009 Conference on Asia and South Pacific Design Automation*, pages 612–617, Piscataway, NJ, USA, 2009. IEEE Press.

[20] L. Li, L. Gao, and J. Xue. Memory coloring: a compiler approach for scratchpad memory management. In *PACT*, pages 329–338, Sept. 2005.

[21] R. Mcllroy, P. Dickman, and J. Sventek. Efficient dynamic heap allocation of scratch-pad memory. In *ISMM'08:The 7th international symposium on Memory management*, pages 31–40, New York, NY, USA, 2008. ACM Press.

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

[23] N. Nguyen, A. Dominguez, and R. Barua. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. In *CASES*, pages 115–125, 2005.

[24] A. Pabalkar, A. Shrivastava, A. Kannan, and J. Lee. SDRM: Simultaneous determination of regions and function-to-region mapping for scratchpad memories. In *Int'l Conference on High Performance Computing (HiPC)*, December 2008.

[25] F. Poletti, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and J. M. Mendias. An integrated hardware/software approach for run-time scratchpad management. In *DAC*, pages 238–243, 2004.

[26] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, pages 213–218, New York, NY, USA, 2002. ACM.

[27] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Design, automation and test*, page 409, Washington, DC, USA, 2002. IEEE Computer Society.

[28] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *Trans. on Embedded Computing Sys.*, 5(2):472–511, 2006.

[29] S. Vangal et al. An 80-tile sub-100-w teraflops processor in 65-nm cmos. In *IEEE Journal of Solid State Circuits*, pages 29–41. IEEE Press, 2008.

[30] M. Verma and P. Marwedel. Overlay techniques for scratchpad memories in low power embedded processors. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(8):802–815, Aug. 2006.

[31] M. Verma, K. Petzold, L. Wehmeyer, H. Falk, and P. Marwedel. Scratchpad sharing strategies for multiprocess embedded systems: A first approach. In *ESTImedia*, pages 115–120, 2005.

[32] M. Verma, L. Wehmeyer, and P. Marwedel. Cache-aware scratchpad allocation algorithm. In *Design, automation and test*, page 21264, 2004.