libEOMP: A Portable OpenMP Runtime Library Based on MCA APIs for Embedded Systems

Cheng Wang*, Sunita Chandrasekaran*, Barbara Chapman*and Jim Holt†
*Dept. of Computer Science, University of Houston, Houston, TX, 77004, USA
†Freescale Semiconductor Inc., Austin, TX, 78735, USA
{cwang35, sunita, chapman}@cs.uh.edu, rwbl70@freescale.com

ABSTRACT

In recent years rapid revolution of Multiprocessor System-on-Chip (MPSoC) poses new challenges for programming such architectures in an efficient manner. In order to explore potential hardware concurrency, software developers are still expected to handle many of the low-level details of programming including utilizing DMA, ensuring cache coherency, and inserting synchronization primitives explicitly. Software portability is yet another issue: the state-of-the-art is that hardware vendors supply vendor-specific software development toolchains which makes it harder for applications to be ported to many different possible architectures without re-structuring the code, while at the same time ensuring efficiency.

In this paper, we extend the usage of a high-level programming model, OpenMP, to multicore embedded systems. To address the architectural challenges, we propose a lightweight unified OpenMP runtime library, *libEOMP*, by leveraging the MCA (Multicore Association) APIs as the target of our OpenMP translation. MCA APIs support device-level communication and resource management for multicore embedded systems. We have implemented and evaluated *libEOMP* on an embedded platform supplied by Freescale Semiconductor. We observed that *libEOMP* not only performed as well as optimized vendor-specific OpenMP runtime libraries but also achieved better portability, programmability and productivity.

Categories and Subject Descriptors

D.3 [**Programming Languages**]: Processors—*Run-time Environments*

General Terms

Languages, Performance, Standardization

Keywords

OpenMP, MCA, Runtime, Embedded System

1. INTRODUCTION

Multicore embedded systems are widely used in telecommunication systems, robotics, automotive vision systems, medical applications, life critical systems and more. Today, they usually consist of homogeneous/heterogeneous cores operating on different ISAs, operating systems and dedicated memory systems in order to provide high throughput, low latency and energy-efficient solutions. Although multicore embedded systems provide lots of potential, the lack of multicore software development tools and standards has created a barrier to their full adoption. Programmers are required to write low-level code, schedule work units and manage synchronization between cores if they are to reap significant benefits from these systems. As the system complexity increases, it is not possible to expect programmers to handle all the low-level details in order to exploit the platform's concurrency. Handling these manually is not only time consuming, but an error-prone and laborious a task to perform.

Even worse, software portability is almost non-existent. The state of the art is that hardware vendors supply vendor-specific development toolchains that are tied to the details of the device they were originally designed for; this may preclude use of the software on any future device even from the same family. Some of the existing approaches that address this issue include defining language extensions or using parallel programming libraries, but there were no well-accepted joint standards that have been adopted in the embedded systems domain.

To address this issue, a group of vendors and software companies formed the Multicore Association (MCA) [3]. The main aim of this association is to reduce the complexity involved in writing software for multicore chips. It has put together a cohesive set of APIs to standardize communication (MCAPI), resource management (MRAPI), and virtualization spanning cores on different chips. The association also has an active working group to provide a tasking API (MTAPI). We have been active members in the working group contributing towards the design of MTAPI specification. Since MCA APIs are vendor-independent application-layer specifications, they do not require architectural or OS support, hence they enable system developers to write portable program codes that will scale through different architectures. Today several vendors from embedded domain have supported the MCA APIs [5].

The MCA APIs could be used to provide a uniform abstraction of the low-level complexities of an embedded platform to the programmer. However, they offer a low-level library-based protocol that would make programming still tedious. Programmers still found it challenging to explore the potential capabilities of the MCA APIs. Moreover they do not help programmers to explore fine-grained data parallelism in embedded applications. In order to improve programmer's productivity, we also need a high-level programming model that could express the concurrency in a given algorithm easily, and leverage the burden of exploring the low-level details from the programmer to compilers. This can be accomplished by using OpenMP [8], a high-level programming model with a simple interface that is an attractive choice for programmers and developers. OpenMP consists of a collection of directives, library routines and environment variables that may be used in conjunction with C, C++ or Fortran to express a shared memory parallel computation.

However, mapping the existing OpenMP implementation techniques to embedded systems is still non-trivial and some obstacles exist. Typically an OpenMP compiler translates an OpenMP directive into multithreaded code containing function calls to a customized runtime library. The runtime library manages the parallel execution on the multicore system, and includes functions for thread creation and management, work scheduling, and synchronization. To meet its responsibilities, the current runtime implementations usually depend on other system components such as the SMP GNU/Linux and thread libraries. Embedded system, however, may lack some of such features.

For instance, although the POSIX threads have been supported by some cache-coherent SMP-based platforms, it requires a SMP GNU/Linux underneath. In addtion, a large number of embedded systems still do not provide such an OS and library support, including the platforms [9] and [22]. In some cases embedded applications are running on bare-metal processors, where OSes and thread libraries do not even exist. Such thread libraries are moreover incapable of capturing all characteristics of embedded systems, such as heterogenous cores and separate address space. These severely obstruct developing the OpenMP on embedded systems.

Consequently, an efficient OpenMP runtime library for multicore embedded systems is highly required. In this paper, we propose a new portable OpenMP runtime library for multicore embedded systems, *libEOMP*, where the 'E' stands for *Embedded.* It exploits the capabilities of the MCA APIs to fill the gap between the existing runtime implementations for general-purpose architecture and the new challenges for multicore embedded systems, to support an implementation of the *de facto* shared memory programming standard OpenMP. Our effort includes selecting appropriate characteristics of the MCA APIs, determining the translation strategy as well as delivering the overall runtime design and implementation for high-performance mapping. Figure 1 gives an overview of our *Embedded* OpenMP solution stack. We evaluated *libEOMP* on a Freescale embedded platform. Results from embedded benchmarks showed that *libEOMP* not only performs as well as vendor-specific approaches but also promises portability, programmability and productivity.

The rest of the paper is organized as follows. In Section 2, we discuss the state of the art in programming multicore embedded systems. In Section 3 we briefly discuss the MCA APIs and how they can be used to implement OpenMP. We explain in detail the design and implementation strategies of

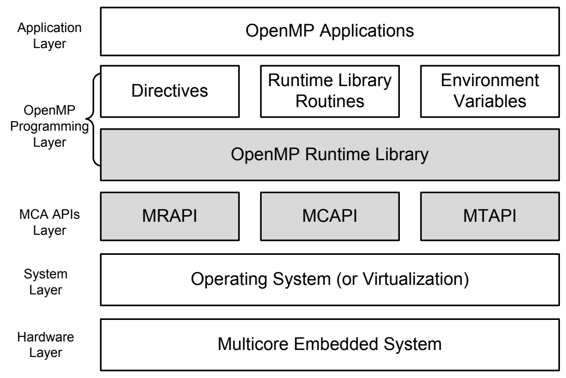


Figure 1: Overview of the *Embedded* OpenMP solution stack.

our novel approach in Section 4 and 5. Results of our evaluation are given in Section 6 and, finally, section 7 presents the conclusions and future work.

2. RELATED WORK

In this section, we discuss the primary programming techniques for multicore embedded systems, which can be categorized as either language extensions, parallel programming libraries or pragma-based approaches.

Language extensions: Assembly and C languages are the most common ones supported by embedded systems vendors. Assembly languages are difficult to use and they are machine-dependent. In embedded market, compilers are not 100% ANSI C compliant. There are two reasons. Primarily, some of the C features are difficult to implement on embedded processors. Secondly, in some cases vendors may require to extend C language because of the need to support special features on the chip. However, the major drawback of these vendor-specific SDKs is the workload of managing the limited on-chip resources is shifted from hardware to the programmer. Hence they find it difficult to learn and is challenging to exploit the underlying platform. Moreover, the code once written using the extended C language, may not be reused for other platforms, while at the same time, guarantee the performance.

Some language extension-based approaches try to abstract the low-level details of the platform from the programmer. SoC-C [24] enables programmers to manage distributed memory and express pipeline parallelism. However SoC-C does not help explore fine-grained data parallelism which is commonly available in embedded applications. Another effort is Offload [13] which provides extensions to C++, that offloads the code to Cell BE processor. However, this approach is unable to be ported to other platforms other than the Cell. OpenCL [7] is a language-based parallel programming on heterogeneous systems from multicore CPUs to accelerators, and Objective-C [6] is the primary language for developing the iPhone applications. Other embedded systems such as FPGAs can also be programmed using similar language extension based approaches such as ImpulseC [23].

However, the issue of these language-extensions is that these approaches require programmers to explicitly parallelize the codes and handle concurrency, scheduling and synchronization, which is less productive, error-prone and time consuming. It is hard to achieve high performance since the programmers have to manually tune the applications and exploit the hardware complexities.

Parallel programming libraries: A parallel programming library which defines a set of APIs is usually comes with a compiler toolchain. This approach abstracts the hardware complexities from programmers and provides coarse-grained data parallelism to some extent. These APIs include MPI and POSIX threads. However, these approaches (e.g. MPI) are either too heavyweight for embedded systems or insufficient for capturing the special characteristics of multicore embedded systems (e.g. the ability of programmers to specify the attribute of memory, such as on-chip SRAM or off-chip DDR). Moreover, these libraries usually rely on specific architecture or OS, which creates portability issues. For instance, the IBM Data Communication and Synchronization (DaCS) [1] library provides a set of data movement primitives that help programmers take advantage of the Cell processor's DMA engines, however it is not portable to other platforms other than the Cell architecture. The MCA APIs, on the other hand, are especially designed for addressing the portability issues for developing multicore embedded applications. MCA offers a set of APIs including MRAPI, MCAPI and MTAPI which capture the general features of embedded systems such as heterogeneity and distinct memory address space. Since they are platform-independent standards, they enable the programmer to write portable programs that will scale through different architectures.

Pragma-based approaches: These approaches are high-level, straightforward and easy to use. It allows the compiler and runtime system to exploit the hardware complexities thus abstracting these details from the programmer. Hence the performance of applications is highly dependent on efficient compiler and runtime implementations. Compared with language extension and library-based approaches, pragma-based approach requires only minimal modification to be performed to parallelize a given code. This drastically reduces the time taken by the programmers.

There are also many efforts to implement the OpenMP on some architectures other than general-purpose CPUs, which includes multicore DSP [17], Cell [22], and GPGPU [19]. Initial experiences of adapting OpenMP for high performance embedded systems is discussed in [12], in which OpenMP was implemented on TI's multicore MPSoC platform by performing a source-to-source translation with the OpenUH compiler [20]. TI's experience on the TMSC6678 multicore DSP platform was also reported in [17]. The Omni OpenMP compiler [25] also adopted a source-to-source approach and implemented on three embedded SMP architectures in [15]. However, none of these solutions can be ported to platforms beyond the initial design. *libEOMP* is a portable solution that will solve the issue.

In order to meet some of the particular characteristics of multicore embedded systems, certain OpenMP extensions were discussed in [16] and [11]. OpenMDSP [16] proposed a extension of OpenMP designed for multicore DSPs. Three classes of directives were proposed: data placement, distributed array and stream access directives. The main goal of these extensions was to fill the gap between the OpenMP memory model and the memory hierarchy of multicore DSPs. Cao et al. [11] proposed an extension for OpenMP tasks on the Cell BE. However, the major drawback of this extension is that it has not been standardized. This implies that the extensions are tied to specific compilers in which they are implemented. Compared to this approach, *libEOMP* resides on the runtime layer thus is transparent to programmers and does not require to learn any new extensions but just to insert normal OpenMP directives to parallelize the code as usual. The underlying details will be handled by the compiler and runtime implementations.

To summarize, we see that there are various approaches to programming multicore systems, but most of them either involve significant manual intervention to explore parallelism or provide solutions that are heavily customized for a specific device. So if the program needs to be ported to another device, the code needs to be rewritten significantly. As a result, programmers are not focusing on the parallel solution of a problem but instead are busy dealing with hardware details of the platform. It is a challenge to resolve these difficulties and still be able to meet stringent time-to-market requirements which are an important factors in the world of embedded systems. An ideal solution is to provide standard APIs for multicore applications to industries. These APIs need to be fast, lightweight, scalable and portable across multiple target platforms and OSes.

3. BACKGROUND

In this section we provide some background details of the MCA APIs in particular the resource management API (MRAPI) we employed in this paper. The MRAPI provides essential capabilities required to manage shared resources in embedded systems, including heterogeneous/homogeneous cores on-chips, hardware accelerators and memory regions. The key features of MRAPI include [4]:

Domain and Nodes: A MCA domain is a unique system global entity that may contain one or more MCA nodes, which can be a process, thread, processor or hardware accelerator. The definition of domain and nodes is essential because it has to be portable to any generic multicore embedded systems and abstract the physical entities, for which that the concept of threads may not exist at all.

Synchronization Primitives: MRAPI synchronization primitives inherit the essential feature sets as other thread libraries for programming multicore platforms, including `mutexes`, `semaphore` and `reader/writer locks`. But rather than coupling with architecture-dependent components, the MRAPI synchronization primitives provide richer functionalities to fulfill the characteristics for embedded systems. For example, embedded memory hierarchy is usually much more complicated, which usually consists of on-chip scratchpad memory and off-chip DDR memory. It may also include the dedicated private memory as well. A mutex attribute is therefore necessary to set the effective scope that the mutex can be shared, for which is impossible for POSIX threads.

Memory Primitives: MRAPI supports two different notions of memory, `shared memory` and `remote memory`. `Shared memory` provides the ability to create and access physically coherent shared memory like POSIX shared memory, but the `remote memory` primitives provides the feature based on the observations that modern heterogeneous embedded systems often contain multiple memory spaces which are dedicated to each core. These memories usually maintain a distinct address space that cannot be accessible from other threads. Thus the `remote memory` primitives aim to manage data movement between these memory spaces but be transparent to end users via the approaches without involving the CPU cycles, such as DMA, Serial RapidIO (SRIO), or software cache.

Metadata Primitives: Aims to obtain the information regarding hardware and application execution statistics which may be used in upper-layer service such as debugging and profiling.

We see that OpenMP and MRAPI share common mapping relationships. The concept of nodes naturally maps to OpenMP threads and tasks. We adopt the MRAPI synchronization primitives to implement the OpenMP synchronization directives, such as `barrier` and `critical`. We utilize the MRAPI shared memory to manage shared resources in order to fulfill the OpenMP memory model, which provides a relaxed-consistency, shared-memory model [8]. Several optimization techniques especially for embedded systems, however, are still essential which will be discussed in detail in Section 4.

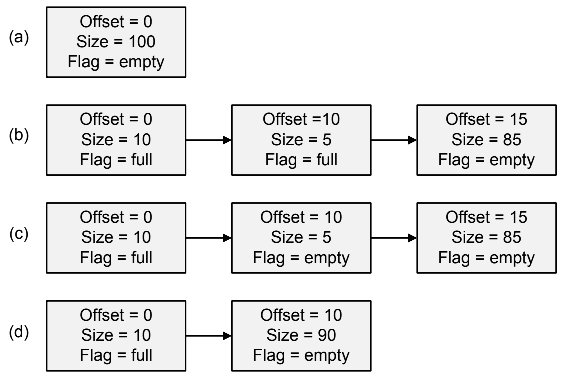


Figure 2: An example of our OpenMP runtime shared memory usage.

4. RUNTIME DESIGN AND KEY OPTIMIZATIONS

In this section, we focus on the runtime system and present the overall design for the mapping to embedded systems. Although the work division between an OpenMP compiler and an runtime library is implementation-defined, the essential function sets of an runtime include creating and scheduling threads, managing shared memory and implementing synchronization primitives. The scope of this paper is not to present the optimal strategies for each of the functions, instead, we inherit some of the algorithms from the OpenUH compiler but rethink how to explore the capabilities of MRAPI to map these techniques onto embedded systems. Several optimization strategies, however, are still needed to be considered. Our main objective is to make our runtime portable, i.e., it will not depend on any dedicated hardware or OS. In addition, we also need to ensure that the additional MCA layer does not incur any significant performance penalty. In the following subsections, we will discuss in detail the design and optimization strategies of each of the OpenMP functionalities.

4.1 Memory Model

The OpenMP API specifies a relaxed-consistency, shared memory model [8]. That means all threads access the same, global shared memory, but can have their own temporary view of private data, and it is the programmer's responsibility to protect global shared data. Therefore, one of the essential jobs of *libEOMP* runtime is to create and manage the shared memory. It is straight-forward to fulfill this requirement in general-purpose multicore, shared-memory architectures, because the functionality provided by OpenMP is restricted to the scope of a single OS image and there exists a physical shared memory that can be addressed by all threads. However the challenge with embedded system is that OpenMP threads may not have directly accessible memory or may run on loosely coupled cores with different ISAs or operating systems. Therefore, the memory allocation method such as *malloc()* is unable to handle the new challenge. As discussed in Section 3, *libEOMP* utilizes the MRAPI `shared memory` primitives to create a segment of shared memory, which also defines a list of nodes that can access that memory. For the variables which are private to each thread(e.g. private, firstprivate, and threadprivate), we use the MRAPI `remote memory` to hold these data. The data transfers between the remote memory as well as the shared memory are handled by DMA which is implicitly implemented by the MRAPI library.

Furthermore, a linked list is used to manage the allocation and deallocation of shared memory. Each node of the linked list represents a distinct allocation segment in the MRAPI shared memory and contains the offset of the starting address of the MRAPI shared memory, the size of the allocated memory, and a flag to indicate whether the slot is empty (not used) or full (used). Figure 2 shows an example of how the linked list is maintained during the allocation and deallocation of MRAPI shared memory. In Figure 2(a), the runtime creates 100 bytes of MRAPI shared memory during initialization. In Figure 2(b), two new nodes with a full flag are inserted into the linked list to request two distinct memory allocations. The node with an empty flag represents the unrequested shared memory. In Figure 2(c), a reclaim of memory with offset address 5 sets the node flag to empty. We also merge neighboring nodes with empty flags as shown in Figure 2(d).

In practice, once a segment of OpenMP memory is allocated, it will not be reclaimed until the end of the program. Thus the latter two cases in Figure 2(c) and (d) will rarely occur. As a result, some other memory management approaches(e.g. buddy allocator) which are widely used in operating systems are sub-optimal here, as it will have too much overhead in managing memory allocation and deallocation, thus creating too many internal/external fragmentation in memory. The linked list, on the other hand, will not waste memory which is restricted in embedded systems. We are also considering other alternative memory management strategies. However, just as discussed before, the scope of this paper is to not obtain an optimal approach, but to validate that many existing techniques can be mapped to embedded systems via the MCA APIs.

We use offset addresses instead of absolute address because the prototype MRAPI shared memory is implemented using the inter-process communication (IPC) shared memory library. Consequently, it is possible that each node may attach the shared memory segment to a different area of its respective address space, which is common for heterogeneous embedded systems. By returning the offset address instead of the absolute address of the MRAPI shared memory, we can maintain addressing consistency between nodes.

4.2 Thread Creation and Management

The OpenMP API uses the fork-join model of parallel execution [8]. An OpenMP program begins with a single sequential thread of execution, which is termed as *master* thread. When the `parallel` construct is encountered, a team of *worker* threads will be created, and the program block enclosed in the parallel region will be executed concurrently among the *worker* threads. At the end of the parallel region, all the worker threads will wait for each other until all the threads have finished execution, after which the master thread will continue execution of the code. Therefore, one of the important roles of the OpenMP runtime library is to create and manage the underlying threads.

An OpenMP compiler will usually translate an OpenMP directive into corresponding runtime library function calls. One major technique is called outlining: an independent function will be created by the compiler that encapsulates the task within that directive region, this is adopted by most of the open source compilers like OpenUH [20] and Omni [25]. For instance, the `parallel` will be translated into the runtime library function call, say *_ompc_fork()*, which will pass the parameters like how many threads are created, what are the assigned tasks and the pointer that maps the tasks to the thread.

Several traditional OpenMP runtime implementations for general-purpose high-performance computing domain usually utilize POSIX threads or other thread libraries to create and manage the OpenMP threads. However, there are several obstructions that prevent us to adopt this approach for embedded systems, and we have discussed it before.

We used the MRAPI *nodes* and its corresponding primitives to create and manage the OpenMP *threads*. The major difference between MRAPI *nodes* and OpenMP *threads* is that MCA *nodes* offer high-level semantics over *threads*, thus hiding the real entities underneath from the programmer. More importantly, in traditional OpenMP implementations, all threads in a team must be identical. However, it may not be the truth for embedded systems where threads running on different cores can be heterogeneous as well. Therefore, the MRAPI *nodes* relax this condition that allow each node to have its own attributes with particular data structures. MCA *nodes* may also create multiple OpenMP *threads* inside the *nodes* to support nested parallelism as well.

4.2.1 Optimizing Thread Creation

Although the idea of thread creation and management is straightforward, there are still several issues that must be resolved to achieve the potential benefits of our approach. One question is: when to create and destroy the worker threads? If they are created and destroyed each time the `parallel` region is encountered, there is obviously a cause for a potential overhead. Another potential issue is, if we allow threads to be created concurrently in each parallel region, without setting an upper bound to the number of threads that can be created, there is a possibility that unlimited number of threads could be created. For instance, although number of threads in a team is fixed, if the nested parallelism is supported, there could still be many sub-teams requesting for unlimited number of threads recursively that will exhaust the limited system resources available in embedded systems especially like CPU cycles and memory. A solution is to create a thread pool to limit the maximum number of threads being created concurrently.

In a thread pool, a team of worker threads is created only once during the *libEOMP* runtime initialization. These threads go back to sleep until a `parallel region` is encountered and a task is assigned. Once the threads finish their work, they will return to the pool and wait until a new task is assigned to them. If the pool has no available threads, the tasks have to wait to be assigned until a new thread is made available. As a result, the *worker* threads are created only once and are reused during the entire program execution time, minimizing thread creation overhead.

Although the concept of pool has been existing for some time, there are several issues must be resolved before we adopt this technique to embedded systems. Typically, the number of threads in the pool are pre-defined and it is usually much larger than the total number of CPUs on the platform. However, it consumes too many resources (e.g. memory, CPU cycles) that are not abundant in embedded systems. One possible solution is to defer creating threads pool until the programmer specifies the number of threads in a team. However, that information may be determined until runtime thus it is too late to create the thread pool. Furthermore, this approach is also unable to support the changing of team size at runtime. Therefore, we adopt another alternative approach that primarily obtains the number of CPUs that are available on the platform, by using the MRAPI *metadata* primitives, and only generates the number of threads that is equal to the number of CPUs. Thread over-subscription is not allowed. If the size of the worker team requested by the programmer is less than the number of threads available in the pool, the idle threads will go back to sleep which will further save the system energy. Consequently, this approach guarantees that no system resources will be wasted.

4.2.2 Optimizing Threads Waiting and Awakening

In the above section, we see that once the thread pool has been created, there are idle threads in the pool waiting to be scheduled. The utilization of the thread pool must be managed efficiently since idleness of threads affects the runtime performance significantly. Thread waiting/awakening is handled by conditional variables and signaling in traditional high-performance computing implementations. Conditional variables provided by POSIX threads require that a thread be waiting explicitly for the conditional variable to receive it. During this process, the CPU cycles are released and can be scheduled to other jobs.

The main disadvantage of this approach is the POSIX threads conditional variables and signaling are always used with mutex locks, i.e., the conditional variable is protected by a single mutex lock and all the worker threads will compete for the mutex lock when they receive the wake-up signal. This causes a large performance overhead when the system scales up.

Consequently, we propose a *distributed spin_waiting* mechanism. Figure 3 shows the state transition diagram for one MRAPI node (thread). As shown in the figure, there is a total of five states in total. When a new node is created as part of the initialization phase, the node is set to the *spin_waiting* state, i.e. it is waiting for new tasks to be assigned. Once it receives a new task the state changes to *ready*, which means it can be dispatched by the system scheduler, after which the state changes to *executing.* When the execution is complete, the node goes back to *spin_waiting* in which thread is

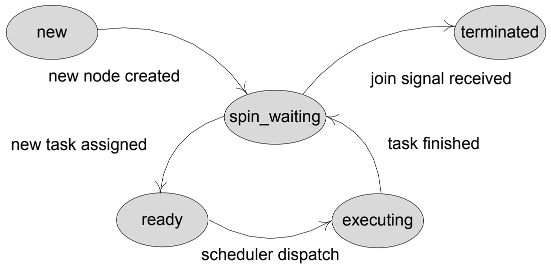


Figure 3: The state transition diagram of an MCA node (or a thread).

polling for a new task, and this cycle is repeated. After all the tasks have finished their execution, the nodes reach the *terminate* state, i.e. the fork-join stage.

The main difference between the *distributed spin_waiting* and POSIX conditional variables/signaling is in *distributed spin_waiting*, the spin variables are private to each node. As a result, each node is only needed for polling its own task rather than competing for the global mutex lock when a new task is available in the POSIX approach. Therefore, there is no lock contention overhead. In addition, in order to avoid false sharing, we assigned the spin variable into the cache line size.

4.3 Synchronization Primitives

The OpenMP synchronization primitives usually include the `master, single, critical` and `barrier` constructs. T-he traditional OpenMP runtime implementation in high-performance computing domain usually relies largely on PO-SIX threads synchronization primitives, such as mutexes and semaphores. However, as discussed before, there are several obstacles for the current approach to efficiently map to embedded systems. We have already discussed the motivation to adopt MRAPI synchronization primitives in Section 3, we will now highlight how we employ the MRAPI primitives to implement the OpenMP synchronization constructs in this section.

4.3.1 Support for Barrier Construct

OpenMP relies heavily on barrier operations to synchronize threads in parallel. Implicit barriers are required at the end of parallel regions; they are also used implicitly at the end of work-sharing construct. Explicit barriers are used by OpenMP developers to synchronize the threads in the team. Like the `parallel` construct, the `barrier` construct is typically translated into runtime library calls during compilation. Thus a good `barrier` implementation is essential to achieve good performance and scalability.

Currently there are several algorithms to implement the `barrier` construct [21]. One commonly used approach is called the *centralized blocking barrier* based on a single shared thread counter, mutexes and conditional variables, which is adopted by many current barrier implementations. In the *centralized blocking barrier*, each thread updates a shared counter atomically once it reaches the barrier. All threads will be blocked on a conditional wait until the value of the counter is equal to the team size. The last thread will send a signal to wake up all other threads.

Although this approach works well for high-performance computing domain, as it will release the CPU cycles to other tasks during the waiting time, there are several challenges for porting to embedded systems, as discussed in section 4.2.2. So we adopt an approach called *centralized barrier* [21], as is showed in Algorithm 1. Similar with the *centralized blocking barrier*, each thread updates a shared counter and waits for the value to be equal to the number of threads in the team. But instead of the conditional wait that all threads race for releasing the mutex lock when they receive the barrier point signal, in *centralized barrier* each thread sets a local *spin_waiting* to continuously polling until the barrier point is reached. Therefore, the barrier can be quickly released when all threads reach the synchronization point thus the performance is better. We will evaluate our approach in Section 6.

Algorithm 1: Centralized barrier algorithm

```
Data: global_barrier_flag, global_count
initialization;
barrier_flag←0;
if active_team_size>1 then
    barrier_flag ← global_barrier_flag;
    mrapi_mutex_lock(...);
    global_count++;
    mrapi_mutex_unlock(...);
    if global_count == active_team_size then
        count_barrier ← 0;
        global_barrier_flag ← barrier_flag^1;
    end
    else
        while barrier_flag==global_barrier_flag do
            ;
        end
    end
end
```

We also noticed that the main drawback of the *centralized* barrier approach is that it may scale poorly. There are other barrier algorithms that may yield better scalability (i.e., the tree-like barrier). However, the purpose of using the *centralized* approach in our current barrier implementation is only to validate that we can easily map current techniques to embedded systems by using the MRAPI synchronization primitives; all low-level details will be handled by the MRAPI implementation. The barrier strategy can be further improved with better and optimal strategies.

4.3.2 Support for Critical, Single and Master Construct

The `critical` construct defines a critical section of code that only one thread can access at a time. When the `critical` construct is encountered, the critical section will be outlined and two runtime library calls, *_ompc_critical* and *ompc_end_critical* respectively, will be inserted at the beginning and at the end of the critical section. The former is implemented as an MRAPI *mutex_lock*, and the latter as an MRAPI *mutex_unlock*.

The `single` construct specifies that the encapsulated code can only be executed by a single thread. Therefore, only the thread that encounters the `single` construct will execute the code within that region. The basic idea is that each thread tries to update a global counter, which is protected by MRAPI mutexes. Thus only the first thread that gains access to the mutex can update the global counter and return a flag. Only the thread that has the flag equal can execute that `single` region.

The `master` construct defines that only the master thread will execute the code. Since the *node_id* has been stored in the MRAPI resource tree, it is fairly easy to find the thread that is the master thread.

4.4 Support for Work-Sharing Constructs

The OpenMP `work-sharing` construct defines a key component of data-parallelism that is widely needed in today's multicore embedded systems. The `loop` construct distributes the execution of the associated loop among the members of the thread team that encounters the loop. The schedule clause determines how the iterations of the loop, called chunks, are distributed among the threads. Each thread executes the chunk assigned to it.

In the default schedule type, i.e. static schedule, the loop iterations or chunks are divided among the threads almost equally. The implementation of the static scheduling in *libEOMP* maintains a global task queue which is filled with chunks. Then a scheduler dispatches the chunks in the queue to each thread in a round-robin fashion. When the scheduling type is dynamic, the runtime will assign chunks dynamically to the threads. In this case, although there is a global task queue, private task queues are maintained by each thread. Once the private queue is empty, it will request new tasks from the global queue, which is protected by an exclusive access provided by MRAPI mutex. We will explain more in detail about the performance evaluation of both scheduling strategies in Section 6.

4.5 Support for Runtime Library Routines and Environment Variables

OpenMP also defines a group of ever-growing runtime library routines and environment variables that are easy to use. We have only implemented the most commonly used ones. For e.g. *omp_get_num_threads* to get the number of threads in a team and *omp_get_thread_num* to obtain the *thread_id*. The environment variable `OMP_NUM_THREADS` that sets the maximum number of threads to be used in parallel.

5. IMPLEMENTATION

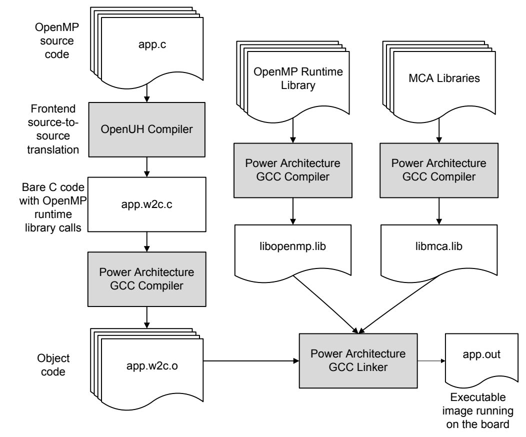
In this section, we will discuss the *libEOMP* implementation on a Freescale embedded platform, the corresponding source-to-source translation of a given code and its code generation process.

5.1 Architecture Overview

We implemented *libEOMP* on a Freescale P1022 Reference Design Kit (RDK) [2], which is a state-of-the-art dual-core Power Architecture^{TM} multicore platform from Freescale. It supports 36-bit physical addressing and double precision floating point. The memory hierarchy consists of three levels: 32KB I/D L1, with 256KB shared L2, and 512 MB 64-bit off-chip DDR memory.

5.2 Compilation Overview

Figure 4 shows the overview of the compilation process. For a given application, e.g. `app.c` with OpenMP directives, we used the OpenUH compiler [20] to perform a source-to-source translation of `app.c` into an intermediate file called



OpenMP source code
app.c
OpenMP Runtime Library
MCA Libraries
Frontend source-to-source translation
OpenUH Compiler
Power Architecture GCC Compiler
Power Architecture GCC Compiler
Bare C code with OpenMP runtime library calls
app.w2c.c
libopenmp.lib
libmca.lib
Power Architecture GCC Compiler
Object code
app.w2c.o
Power Architecture GCC Linker
app.out
Executable image running on the board

Figure 4: Overview of the cross-compilation process.

`app.w2c.c`. This `app.w2c.c` file is a bare C code with runtime library function calls, which will be fed into the backend native compiler, Power Architecture gcc compiler, a compiler toolchain for the Freescale e500v2 processor. The output is the object code `app.w2c.o`. During the linking phase, the linker will link all the object codes together with the runtime libraries `libopenmp` and `libmca` which were previously compiled by the native compiler.

5.3 Source-to-Source Translation

We use OpenUH compiler [20] as the frontend to perform the source-to-source translation. OpenUH is a branch of the open-source Open64 compiler suite for C, C++ and Fortran 95/2003, OpenMP 3.1, Co-array Fortran and UPC. OpenUH has an IR-to-source translator (*whirl2c* and *whirl2f*) that can translate the intermediate representations (IR) to backend compilable source code. The transformation of OpenMP, which involves lowering OpenMP pragmas into corresponding bare code with runtime library calls, is mainly performed in two steps: OMP_Prelower and LOWER_MP [20]. The former performs the preprocessing while the latter performs the major translations. After the *whirl2c* procedure, generated files will be fed into any suitable target compiler and linked with the runtime libraries to generate executable images.

6. PERFORMANCE EVALUATION

The main purpose of evaluating *libEOMP* is to demonstrate that the additional MCA layer does not incur any significant performance overhead but helps in portability. Our baseline is to compare the performance of the *libEOMP* with a vendor-specific runtime library which is optimized for the target platform. Porting an OpenMP code to more than one platform using *libEOMP* is possible provided the platform is supported with MCA APIs.

We have considered two benchmarks, ranging from micro benchmarks to real embedded applications, EPCC micro benchmarks [10] and MiBench [14]. We have utilized a prototype implementation of MRAPI version 2.0.3, provided by Freescale Semiconductor. To compare *libEOMP* with that of the currently available optimized library, we have also compiled the benchmarks and linked with a native vendor-specific runtime library *libGOMP* from Power Architecture GCC compiler toolchain for the Freescale e500v2 processor. We calculate the total execution time and the speedup achieved in both cases i.e. for *libEOMP* and *libGOMP*.

6.1 Evaluation on EPCC Micro Benchmarks

The EPCC Micro Benchmark suite is a set of programs that measure the overhead of each of the OpenMP directives and evaluates different OpenMP runtime library implementations. It includes two benchmarks: *syncbench* to evaluate the overhead for each of the OpenMP directives, while *schedbench* measures the loop scheduling overhead using different chunk sizes.

6.1.1 Syncbench

We evaluated the *syncbench* benchmark with *libEOMP* and the results are tabulated in Table 1. For each of the OpenMP directives, we run the experiment 10 times, with 10,000 iterations and calculate the average time taken. The table shows that the overall performance of *libEOMP* is quite competitive with that of the vendor-specific OpenMP runtime library *libGOMP*. The time difference is less than 1 microsecond, barely noticeable by programmers. The difference may be due to the function calls added by the additional MCA APIs layer in *libEOMP*. Moreover, the MRAPI library that we have used is only a prototype implementation and it does not guarantee the best performance.

The table 1 also shows that directives such as the `parallel` and the `single` constructs in *libEOMP* even outperform those in *libGOMP*. This is due to the sophisticated thread creation and optimized thread management techniques as discussed in section 4.2 that we have used to implement the `parallel` construct.

We also see that the presence of the implicit barrier hidden in most of the OpenMP directives dominates the major part of the performance. For example, the overhead of the `parallel` construct is dominated by two `barrier` constructs, one at the beginning of the parallel region and one at the end of the parallel region. At the beginning of the parallel region, the `barrier` construct ensures that all the worker threads are ready for execution. At the end of the parallel region, as per the OpenMP specification, there is a need for a `barrier` construct that ensures that all the worker threads have finished execution. So we could see that the implementation of the `barrier` construct is quite crucial for performance.

6.1.2 Schedbench

Schedbench measures the loop scheduling overhead of OpenMP directives. Figure 5 shows the results with different chunk sizes. Figure 5(a) shows the results of overhead due to static scheduling. Overall, from the figure we can see that *libEOMP* runtime performs as fast as *libGOMP*. As a worst case scenario with chunk size 1, the time taken by *libEOMP* is 61.09 μsec while the time taken by *libGOMP* is 58.77 μsec; the time difference is only 2.32 μsec. The best case scenario is when no chunk size is specified in the `static` clause and the iteration space is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each thread. In this case, *libEOMP* takes 54.90 μsec while *libGOMP* takes 53.48μsec; the time difference is only 1.42 μsec. This clearly shows that adding the MCA layer did not incur too much overhead.

Figure 5(a) also shows that the execution time of different chunk sizes follows an identical pattern. This is because each thread will receive roughly an equal number of chunks no

Table 1: Average execution time(μs) for EPCC syncbench.

DIRECTIVE	libEOMP	libGOMP
PARALLEL	8.62	9.10
FOR	8.46	7.14
PARALLEL FOR	9.44	9.20
BARRIER	7.61	7.13
SINGLE	2.15	6.82
CRITICAL	8.66	6.56
REDUCTION	10.44	9.22

matter the size of the data is. The smaller the chunk size, the poorer the performance, because the scheduler assigns chunks in a round-robin fashion to each thread. It is often likely that some threads may be starving for chunks while the scheduler is busy serving other threads. The runtime appears to perform the best with the default chunk size (i.e. when no chunk size is specified) because in this case each thread will get the largest chunk size and the scheduler will only need to assign chunks to a thread once. In this way the loop scheduling overhead is minimized.

Figure 5(b) shows the overheads due to dynamic scheduling method. The figure shows that *libEOMP* performs worse than *libGOMP* when the chunk size is small but is quite competitive with *libGOMP* when the chunk size is large. The difference between *libGOMP* and *libEOMP* is 1.49 μsec. As discussed in section 4.4, each thread maintains a private task queue. Once the private task queue is empty the thread will request for new tasks from the global task queue, which requires exclusive access, i.e. no other threads can access the global task queue at this point. If the chunk size is too small, it is likely that the threads will consume the chunks quickly and make a request to the global queue for more chunks. So if more and more threads begin to request chunks from the global queue concurrently, then there are other issues that must be dealt with: e.g. acquiring and releasing locks by different threads to access chunks from the global queue can be a very expensive process.

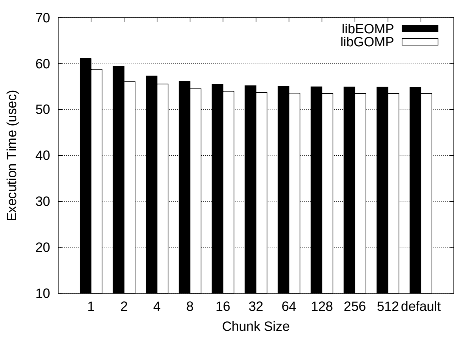
The dynamic scheduling mechanism can be improved to exhibit better performance. One of the approaches to improve performance is to use distributed task queues instead of a single global queue which may reduce thread management traffic. Workstealing [18] mechanism can be used, such that a thread that runs out of tasks can steal tasks from the queues of other threads (future work).

6.2 Evaluation of MiBench

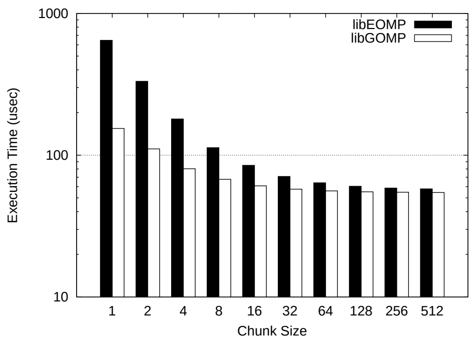
In this section, we evaluated *libEOMP* using some of the MiBench benchmarks which is a suitable benchmark suite for embedded systems. It is divided into multiple classes in order to represent different embedded domains. Three benchmarks, FFT, Dijkstra, and DGEMM (Double-precision Floating-point Matrix Multiply), are used.

We first parallelized these benchmarks by using OpenMP since the original version of the code in MiBench is sequential. In the FFT algorithm, $\log N$ stages are needed for a given wave of length N and the tasks within each stage are equally distributed to each of the threads. In the DGEMM and Dijkstra algorithms, the data size is evenly divided among threads.

Figure 6 shows the comparative results of execution time

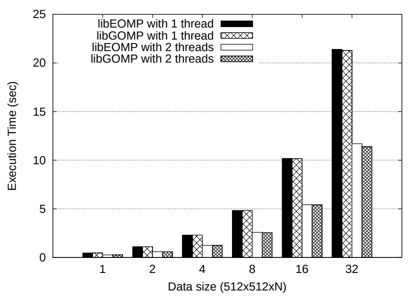


(a) Static scheduling overhead.

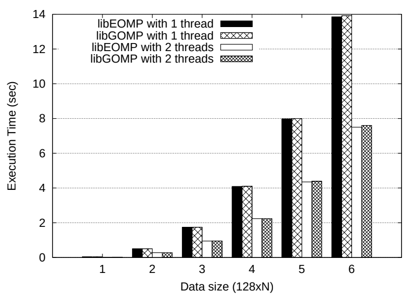


(b) Dynamic scheduling overhead.

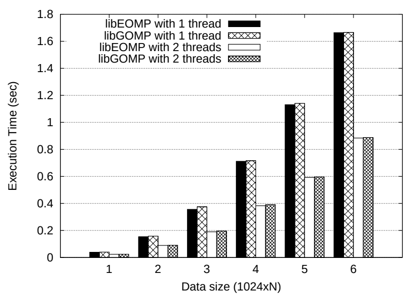
Figure 5: Loop scheduling overheads of EPCC *schedbench* using different chunk sizes



(a) FFT



(b) DGEMM



(c) Dijkstra

Figure 6: Execution time of FFT, DGEMM and Dijkstra with varying data size using 1 and 2 threads.

using different data sizes and numbers of threads. From the figure we see that *libEOMP* achieves almost the same performance as that of *libGOMP*. For the Dijkstra, *libEOMP* outperforms *libGOMP* due to the efficient implementations of the `parallel` and `single` constructs as discussed in Section 4. Figure 6 also shows that *libEOMP* performs well when the data size is increased dramatically (the data size in FFT is increased to 8 Mbyte) showing good scalability.

To summarize, although achieving the best performance is not the main scope of this paper, our objective is to show that *libEOMP* can perform as well as the native vendor-specific OpenMP runtime library, *libGOMP*. *libEOMP* is based on the MCA APIs, which do not rely on any specific architecture or OS, hence the library can be ported easily to multiple platforms which is not the case with *libGOMP*.

7. CONCLUSION AND FUTURE WORK

Programming model on multicore embedded systems is important yet challenging. In this paper, we have described the design and implementation details of a novel OpenMP runtime library, *libEOMP*. *libEOMP* utilizes an industry standard formulated by MCA underneath with high-level programming model, OpenMP atop. Using *libEOMP* the programmer gets enough control to write efficient code, especially when the hardware details are abstracted from the programmer. Evaluation results of *libEOMP* using several benchmarks have demonstrated that *libEOMP* performs not only as competitive as optimized vendor-specific approach but also offers portability and productivity.

Currently we only implemented the *libEOMP* on a homogeneous platform. We have not been able to consider evaluating our novel approach on a heterogeneous platform since a prototype implementation of MRAPI covering features of heterogeneous system does not yet exist. As part of the future work, once the prototype implementation of MRAPI for heterogeneous platform is made available, we plan to make the best use of the same and be able to target a variety of underlying devices. We will also be considering the newest version of OpenMP that will be providing support for heterogeneous systems. The current version of OpenMP 3.1 does not address the concept of heterogeneity or accelerators.

We would like to mention that the prototype of *libEOMP* discussed in this paper will be further improved to provide an optimal solution for the applications under consideration for embedded systems. Currently the prototype has been constructed to demonstrate that OpenMP could be used with MCA APIs to execute applications on embedded systems without the need to be aware of the low-level details of the system.

8. ACKNOWLEDGMENTS

The authors of this paper would like to express our sincere gratitude to the anonymous reviewers for their insightful comments. This research has been supported by grants from Freescale Semiconductor Inc. and Texas Instruments Inc. in association with Semiconductor Research Corporation(SRC).

9. REFERENCES

[1] Data Communication and Synchronization Library for Cell Broadband Engine Programmer's Guide and API

reference, Version 3.0.

[2] Freescale Semiconductor Inc. `http://www.freescale.com/`.

[3] Multicore Association. `http://www.multicore-association.org`.

[4] Multicore Resource API (MRAPI) Specification V1.0. `http://www.multicore-association.org`.

[5] Polycore MCAPI Offers ThreadX RTOS Support.

[6] The Objective-C programming languages. `http://developer.apple.com`.

[7] The OpenCL Specification, Version 1.0. `http://www.khronos.org`.

[8] The OpenMP API Specification for Parallel Programming. `http://openmp.org/wp/`.

[9] TMDXEVM6678L EVM Technical Reference Manual Version 1.0, Literature Number: SPRUH58.

[10] J. Bull. Measuring Synchronisation and Scheduling Overheads in OpenMP. In *Proceedings of the First European Workshop on OpenMP*, pages 99–105, 1999.

[11] Q. Cao, C. Hu, H. He, X. Huang, and S. Li. Support for OpenMP Tasks on Cell Architecture. In *Proc. of the 10th international conference on Algorithms and Architectures for Parallel Processing - Volume Part II*, ICA3PP'10, pages 308–317. Springer-Verlag, 2010.

[12] B. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, and A. Gatherer. Implementing OpenMP on a High Performance Embedded Multicore MPSoC. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8, 2009.

[13] P. Cooper, U. Dolinsky, A. F. Donaldson, A. Richards, C. Riley, and G. Russell. Offload: Automating Code Migration to Heterogeneous Multicore Systems. In *Proc. of the 5th international conference on High Performance Embedded Architectures and Compilers*, HiPEAC'10, pages 337–352. Springer-Verlag, 2010.

[14] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proc. of WWC-4, 2001.*, pages 3–14. IEEE Computer Society, 2001.

[15] T. Hanawa, M. Sato, J. Lee, T. Imada, H. Kimura, and T. Boku. Evaluation of Multicore Processors for Embedded Systems by Parallel Benchmark Program Using OpenMP. *Evolving OpenMP in an Age of Extreme Parallelism*, pages 15–27, 2009.

[16] J. He, W. Chen, G. Chen, W. Zheng, Z. Tang, and H. Ye. OpenMDSP: Extending OpenMP to Program Multi-Core DSP. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 288–297. IEEE, 2011.

[17] F. D. Igual, M. Ali, A. Friedmann, E. Stotzer, T. Wentz, and R. A. van de Geijn. Unleashing the High-Performance and Low-Power of Multi-core DSPs for General-Purpose HPC. In *Proceedings of SC ' 12*, SC '12, pages 26:1–26:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[18] J. LaGrone, A. Aribuki, C. Addison, and B. M. Chapman. A Runtime Implementation of OpenMP Tasks. In *IWOMP*, pages 165–178, 2011.

[19] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *Proc. of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11. IEEE Computer Society, 2010.

[20] C. Liao, O. Hernandez, B. M. Chapman, W. Chen, and W. Zheng. OpenUH: an Optimizing, Portable OpenMP Compiler. *Concurrency and Computation: Practice and Experience*, 19(18):2317–2332, 2007.

[21] R. Nanjegowda, O. Hernandez, B. Chapman, and H. H. Jin. Scalability Evaluation of Barrier Algorithms for OpenMP. In *Proc. of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, IWOMP '09, pages 42–52. Springer-Verlag, 2009.

[22] K. O'Brien, K. O'Brien, Z. Sura, T. Chen, and T. Zhang. Supporting OpenMP on Cell. *Int. J. Parallel Program.*, 36(3):289–311, June 2008.

[23] D. Pellerin and S. Thibault. *Practical FPGA Programming in C.* Prentice Hall Press, 2005.

[24] A. Reid, K. Flautner, E. Grimley-Evans, and Y. Lin. SoC-C: Efficient Programming Abstractions for Heterogeneous Multicore Systems on Chip. In *Proc. of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 95–104. ACM, 2008.

[25] M. Sato, M. S. Shigehisa, K. Kusano, and Y. Tanaka. Design of OpenMP Compiler for an SMP Cluster. In *In EWOMP '99*, pages 32–39, 1999.