Portable Mapping of OpenMP to Multicore Embedded Systems Using MCA APIs

Cheng Wang[‡] Sunita Chandrasekaran[‡] Peng Sun[‡] Barbara Chapman[‡] Jim Holt[†]

[‡]Department of Computer Science, University of Houston, Houston, TX, 77004, USA
[†]Freescale Semiconductor Inc., Austin, TX, 78735, USA
{cwang35, sunita, psun5, chapman}@cs.uh.edu, rwbl70@freescale.com

Abstract

Multicore embedded systems are being widely used in telecommunication systems, robotics, medical applications and more. While they offer a high-performance with low-power solution, programming in an efficient way is still a challenge. In order to exploit the capabilities that the hardware offers, software developers are expected to handle many of the low-level details of programming including utilizing DMA, ensuring cache coherency, and inserting synchronization primitives explicitly. The state-of-the-art involves solutions where the software toolchain is too vendor-specific thus tying the software to a particular hardware leaving no room for portability.

In this paper we present a runtime system to explore mapping a high-level programming model, OpenMP, on to multicore embedded systems. A key feature of our scheme is that unlike the existing approaches that largely rely on POSIX threads, our approach leverages the Multicore Association (MCA) APIs as an OpenMP translation layer. The MCA APIs is a set of low-level APIs handling resource management, inter-process communications and task scheduling for multicore embedded systems. By deploying the MCA APIs, our runtime is able to effectively capture the characteristics of multicore embedded systems compared with the POSIX threads. Furthermore, the MCA layer enables our runtime implementation to be portable across various architectures. Thus programmers only need to maintain a single OpenMP code base which is compatible by various compilers, while on the other hand, the code is portable across different possible types of platforms. We have evaluated our runtime system using several embedded benchmarks. The experiments demonstrate promising and competitive performance compared to the native approach for the platform.

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

General Terms Languages, Performance, Standardization

Keywords OpenMP; MCA; Runtime optimizations; Embedded systems

1. Introduction

Multicore embedded systems usually consist of embedded CPUs, sensors and accelerators to provide high-performance but low-power solutions. Although these embedded systems offer great hardware capabilities, the limited availability of multicore software programming models and standards pose a challenge for their full adoption. Programmers typically have to write low-level codes, schedule task units and manage synchronization explicitly between cores. As the hardware complexity is rapidly growing, it is nearly impossible to expect programmers to manually handle all the low-level details. This is not only time-consuming but an error-prone approach.

Another major concern is the software portability. The state-of-the-art for programming embedded systems includes proprietary vendor-specific software development toolchains that are tightly coupled with specific platforms. As a result, software developers have to largely restructure the code if they want to port it to other platforms, while at the same time, ensure the performance. This leads to a less-productive and error-prone software development process that is unacceptable for ever growing complexity of embedded hardware. If the multicore embedded industry is to quickly adopt multicore embedded devices, one of the key factors to consider is to move from proprietary solutions to open standards.

The Multicore Association(MCA) APIs [4], an extensive industry standard founded by a group of vendors from semiconductor and embedded software industries. The road map of MCA consists of an extensive set of APIs that support multicore communication (MCAPI), resource management (MRAPI), and virtualization spanning cores on different chips. MCA also has an active working group to provide a set of APIs for task management (MTAPI) charged with creating an industry-standard specification that supports coordination of tasks on embedded parallel systems. We have been active members in this working group, participating in the design of MTAPI specification. In addition, the MCA APIs provide a standard interface for programmers which is independent of any operating systems and devices thus allowing it to be portable across various possible architectures. Standards take a long time to develop and establish. There are better chances of more and more programmers embracing such standards if OS, processor and tool vendors provide MCA API as part of their implementation. We see that there is a significant number of vendors that are supporting MCA APIs [5, 10].

However, a point to note is that the MCA APIs are low-level library-based protocols that could make programming still tedious. Programmers still have to explore the features of MCA APIs and largely restructure the code using MCA APIs. As a result, a high-level programming model is needed that could help to express concurrency in a given application easily, while on the other hand,

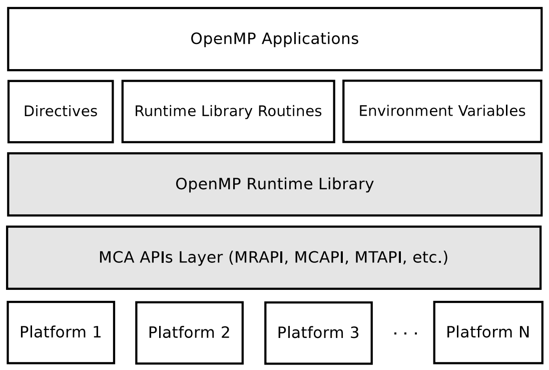


Figure 1. Overview of the embedded OpenMP solution stack.

provide sufficient features which are capable of capturing the low-level details of the underlying systems.

In this paper, we propose a compiler-runtime approach that offers a well-known high-level programming model, OpenMP to programmers, while the underlying OpenMP runtime library exploits the capabilities of MCA APIs to hide low-level details of the platform. OpenMP is a *de facto* standard for shared memory parallel programming. It provides a set of high-level directives, runtime library routines and environment variables that enable programmers to easily express data and task parallelism in an incremental programming style. The compiler handles the low-level details of thread management, loop scheduling and synchronization primitives. OpenMP code is portable across a number of compilers and architectures, which allows the programmer to focus on the application instead of low-level details of the platform. Embedded programmers could therefore benefit from such software portability and programmability.

As OpenMP was original designed for high performance computing systems, there are some challenges that prevent us from directly using OpenMP for multicore embedded systems. One of the hurdles is that embedded systems may lack some of the features which can be commonly found in general-purpose processors. For instance, a traditional OpenMP compiler translates non-executable pragmas (parallel, for, and so on) into multithreaded code with function calls to a customized runtime library, which typically relies on POSIX threads and SMP GNU/linux. However, when comes to embedded space, there are a large number of embedded platforms (e.g., Texas Instruments TMS320C6678 multicore DSP [1]) do not come with such support. In certain scenarios, embedded applications may run on bare-metal processors, where operating systems and thread libraries do not even exist. Moreover, modern embedded systems usually consist of heterogeneous processors operating on different ISAs, OSes and non-cache coherent memory subsystems with separate memory space, where POSIX threads alone is incapable of capturing all these features of embedded systems. Although there are some recent proprietary implementations that support OpenMP for their platforms [24], this approach significantly sacrifices the portability property of OpenMP. In contrast, our approach utilizes the MCA APIs as the translation layer, thus the implementation is portable across various possible architectures. We will evaluate the portability and performance under different platforms in Section 6.

Figure 1 shows the overview of our embedded OpenMP solution stack. A typical OpenMP compiler transforms OpenMP directives into multi-threaded code with function calls to a customized runtime library. An efficient runtime library creates and schedules threads, manages shared memory and implements synchronization primitives. In this paper, we designed and implemented an unified OpenMP runtime library, namely *libEOMP*, that utilizes the MCA APIs as a translation layer. The MCA APIs could help to bridge the gap between the existing OpenMP runtime implementations for general-purpose platforms and the unique challenges for multicore embedded systems. In addition, since our OpenMP runtime is built on top of the MCA layer and it is architecture independent, it is thus portable across various types of architectures. We evaluated our *libEOMP* using micro benchmark suite and real embedded applications on two state-of-the-art multicore embedded platforms. Experimental results showed that the *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 discuss in detail the design and implementation strategies of our novel approach in Section 4 and 5. Results of our evaluation are discussed in Section 6 and, finally, section 7 presents the conclusions and future work.

2. Related Work

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

Language extensions: Assembly and C are the most commonly used programming languages for embedded systems. Assembly language is efficient but difficult to use. It also depends on specific architecture hence offers no portability. In the embedded market, compilers are not 100% ANSI C compliant, which is mainly because some of the C features are difficult to implement on embedded processors. Besides, vendors are usually required to extend C language due to the need to support special features on the chip. However, the major drawback of these vendor-specific SDKs is that the workload of managing limited on-chip resources has been shifted from hardware to programmers. Hence they can hardly exploit the underlying platform without the knowledge of the hardware details.

Some language extension-based approaches try to abstract the low-level details of the platform from the programmer. For instance, SoC-C [32] enables programmers to manage distributed memory and express pipeline parallelism. However SoC-C does not help explore data parallelism which is commonly available in embedded applications. Another effort is Offload [16] which provides extensions to C++, that offloads the code to Cell BE processor. However, this approach is unable to be ported to any other platforms other than the Cell processor. OpenCL [8] is a parallel programming language on heterogeneous systems, and Objective-C [7] is the primary language for developing the iPhone applications. Lime [11] is also a Java-based approach supporting heterogeneous processors. Other embedded systems such as FPGAs can also be programmed using similar language extension based approaches such as ImpulseC [31], Handel-C [28], Streams-C [19]. However, an important point to note is that these language extensions are still very low-level and do not quite express data and task parallelism easily. Programmers are required to manually restructure the source code and manage the barrier, loop scheduling, reduction, data sharing and synchronization, which leads to a non-trivial, less productive and error-prone technique.

Parallel programming libraries: Many programming languages also offer a set of customized runtime libraries (APIs) to ease programming to some extent. For instance, the IBM Data Communication and Synchronization (DaCS) [2] library also pro-

Table 1. Overview of the Resource Management API (MRAPI) feature set.

Features	Description
Domain and Nodes	Real execution entities,e.g., process, thread or accelerator
Memory primitives	Allocate and manage on-chip/off-chip shared/remote memory
Synchronization primitives	Mutexes, semaphore, and reader/writer locks
Metadata primitives	Retrieve hardware information

vides 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 Cell architectures. Several universal libraries such as MPI [35] and POSIX threads [17] do not require any specific compiler support. However the implementation of MPI is too heavyweight for embedded systems. As we have discussed before, the POSIX threads is yet insufficient to tackle all the characteristics for modern multiprocessor SoC. Apart from the MCA APIs, Phalanx [18] also provides an unified runtime system aiming to ease of programming on heterogeneous machines. By leveraging the GASNet runtime, Phalanx is also able to run across all the nodes of a distributed-memory machine.

Pragma-based approaches: Pragma-based approaches, such as OpenMP [9], 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 programmers. Compared with language extension and library-based approaches, pragma-based approaches require only minimal modification to be done to parallelize a given code. There exists many efforts to implement OpenMP on some architectures other than general-purpose CPUs, including multicore DSP [24], Cell [30], and GPGPU [25, 26]. In addition, there also exists a large number of work trying to map a high-level programming model to heterogeneous platforms, such as CnC [34] and HiCUDA [21].

An initial experience of adapting OpenMP for multicore embedded systems is discussed in [14], in which OpenMP was implemented on Texas Instrument's multicore MPSoC platform by performing a source-to-source translation with the OpenUH compiler [27]. The Omni OpenMP compiler [33] which also adopted a source-to-source approach was implemented on three embedded SMP architectures in [22]. Texas Instrument's experience on porting the OpenMP to their TMSC6678 multicore DSP platform was also reported in [24]. However, the main limitation is none of these solutions can be ported to other platforms beyond the initial design. On the contrary, our *libEOMP* is built on top of the MCA APIs thus its implementation is seamlessly portable.

Extensions to OpenMP were also proposed in order to address the architectural challenges for embedded systems. OpenMDSP [23] proposed extensions for OpenMP to target multicore DSPs. In order to fill the gap between the OpenMP memory model and the memory hierarchy of multicore DSPs, three classes of directives were proposed: data placement, distributed array and stream access directives. Cao et al. [13] also proposed an extension for OpenMP tasks on the Cell BE. A major drawback of these existing approaches is that they are not standardized solutions which amortized the portability of OpenMP.

Our approach inherits the advantages of pragma-based approaches discussed above. Moreover, our approach neither requires the programmer to understand low-level details of the hardware nor requires the programmer to learn any new parallel programming language. The programmer simply needs to insert standard OpenMP directives wherever necessary to parallelize the code and allow the compiler and runtime to handle the low-level details.

To summarize, we see that there are various approaches to programming multicore systems, but most of these 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 other devices, there will involve significant efforts on code restructuring. Software developers are therefore not able to focus 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 issues while still be able to meet stringent time-to-market requirements which are an important factor in the world of embedded systems. An ideal solution is to provide a set of standard APIs for multicore applications, which must be fast, lightweight, scalable and portable across multiple target platforms and OSes.

3. Background

In this section, we introduce some background details to MCA APIs, in particular the resource management API (MRAPI) [6] that we have employed in this paper. The MRAPI provides essential feature sets required to manage shared resources in multicore embedded systems, including homogeneous/heterogeneous core on-chip, hardware accelerators and memory regions. Table 1 gives an overview of the MRAPI feature set.

Domain and Nodes: *Domain* and *nodes* define the overall granularity of the resources. An MCA *Domain* is a global entity which can consist of one or more MCA *nodes*. The major difference between MCA *nodes* and POSIX *threads* is that *nodes* offer a high-level semantics over *threads*, thus hiding real entities of the execution. Furthermore, the POSIX *threads* require that all threads in a team must be identical. However, this may not be the case for embedded systems where threads running on different cores can be heterogeneous as well. Therefore, the MRAPI *nodes* relaxes this condition and allows *nodes* in a team to be distinct with its own attributes and data structures. For instance, a team of *nodes* can consist of CPUs, DSPs and accelerators.

Memory Primitives: MRAPI supports two different types of memory, *shared memory* and *remote memory*. *Shared memory* provides the ability to allocate, access and delete on-chip and off-chip memory. But unlike the POSIX threads where we are not able to directly control the property of the shared memory, the MRAPI *shared memory*, on the other hand, allows programmers to specify attributes of shared memory as on-chip SRAM or off-chip DDR memory. In addition, modern embedded systems often consist of heterogeneous cores where they have separate memory address space that may not be directly accessible by other nodes. MRAPI also provides another memory primitives, `remote memory`, which enables the data movement between these memory space without involving CPU cycles but using DMA, serial rapidIO (SRIO) or software cache. MRAPI keeps the data movement operation hidden from the end-users.

Synchronization primitives: MRAPI *synchronization* inherits the essential feature sets from other thread libraries for shared memory programming, including that of *mutexes, semaphore* and *reader/writer locks*. But unlike the POSIX threads, the MRAPI synchronization primitives provide richer functionality to fulfill the characteristics for embedded systems. For example, MRAPI locks can be shared by all *nodes* as well as by only a group of *nodes*.

Metadata primitives: Using *metadata* primitives, we could gather information about hardware and application execution statistics that may be used for debugging and profiling purposes.

We see that OpenMP and MRAPI share common mapping relationships. The concept of *nodes* naturally maps on 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* and *remote memory* to address the OpenMP memory model that provides a relaxed-consistency, shared-memory model. We will discuss these in detail in Section 4.

4. Runtime Design and Key Optimizations

In this section, we focus on the runtime system and present the overall design choices for using MCA APIs. In our prior contribution [36], we presented an initial implementation of the *libEOMP*, the purpose was to only validate the functional correctness of each of the design phases. In our current paper we largely extend our prior work and perform a number of improvements to the design and implementation to achieve better results.

Typically an OpenMP compiler translates OpenMP directives into multithreaded code which consists of functions calls to a customized runtime library. Although the work division between an OpenMP compiler and an runtime library is implementation-defined, the essential functionality provided by a runtime includes:

- Efficient handling threads creation, synchronization and management.
- Parallel distributing loop iterations among threads when assisting the compiler to transform work sharing construct.
- Fulfilling the OpenMP memory model and managing the shared memory.
- Supporting runtime library routines and environment variables.

In addition, our runtime design inherits basic features from some of the commonly used open-source OpenMP implementations such as OpenUH [27], but since the features are not tailored for embedded systems, we had to adopt several strategies to customize the implementations for embedded systems. In the following subsections, we discuss the improvements made to the OpenMP implementation optimized for embedded systems.

4.1 Execution Model

OpenMP uses the fork-join model of parallel execution. An OpenMP program begins with a single sequential thread of execution, which is termed as *master* thread. When a parallel region 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 parallel region, all the *worker* threads will wait for each other until all the executions are finished allowing the *master* thread to run to completion.Several traditional approaches employ the POSIX threads or system functions to create and manage OpenMP threads. However this is not favorable to embedded systems as discussed in section 3. We map OpenMP threads to MCA *nodes* in our design. However, several optimization for efficient handling threads creation and management need to be performed before we adapt the existing techniques to embedded systems.

4.1.1 Optimizing the thread pool

The *thread pool* technique is commonly used for managing OpenMP threads. A team of *worker* threads is created only once during the runtime initialization. These *worker* threads wait until a parallel region is encountered, the *master* thread then assigns a microtask to them. A microtask consists of information about the entry function, data pointer and a wake-up message to wake-up the *worker* threads. Upon completion of the microtasks, the *worker* threads will wait in the pool until a new microtask is assigned. As a result, the *worker* threads are created only once but can be reused during the entire program execution time. This helps in reducing the thread creation overhead.

Although the concept of thread pool is straightforward, there are several issues that must be resolved before we adopt this approach to embedded systems. Conventionally a large number of threads (for example 256) are created in the thread pool for the need of nested parallelism. This may lead to thread oversubscription that typically high-performance computing systems can afford and sometimes oversubscription has the potential to improve performance with better load balancing and CPU utilization. Especially when executing applications that have control and data dependencies that requires threads to wait for each other. But with the dedicated environments (i.e., embedded systems) thread oversubscription degrades performance since the computational resources are relatively limited. Moreover, this approach is too heavyweight for embedded systems since memory in these systems is not available in abundance. Hence thread creation and usage in embedded systems requires careful management.

In our design, we adopt a simple heuristic to create an *elastic* thread pool. Primarily we obtain the number of *nodes* available on the platform, by using the MRAPI *metadata* primitive, then we only generate that many number of threads as required in the thread pool. As a result, thread oversubscription will not occur. At a later stage, if the programmer specifies the number of threads that is less than the size of the thread pool, the idle threads will stall freeing the CPU cycles by using our worker-initiated idle threads handling approach discussed below.

4.1.2 Worker-initiated idle threads handling

Here we discuss about managing the idle threads who are waiting in the pool to be assigned with microtasks. In the above discussion, we saw that once the thread pool has been created, there are idle threads in the pool waiting to be scheduled. Those idle threads must be well-handled as they should neither consume too much system resource nor have a long time delay when they receive a wake-up signal.

Traditionally, idle threads are handled by the master thread: i.e., if a worker thread finishes its execution, it simply declares itself as idle, by changing the status of the message queue. It then sleeps on a conditional wait until a wake-up signal with a new microtask is received. This is conventionally accomplished using the POSIX threads mutex lock and conditional variables. We call this *master-initiated* approach as it is the master thread's responsibility to wake up all the worker threads. The advantage of this approach is that CPU cycles will be released when worker threads are idle using conditional wait. Thus it is well-adopted in most of conventional OpenMP runtime implementation with a large thread pool, as it would be catastrophic if hundreds of idle threads are busy with polling new microtasks. However, the main drawback of the master-initiated approach is that all worker threads will compete for the mutex lock when they receive the wake-up signal. That is, worker threads have to acquire the lock before they can proceed to execute the new microtask. This will cause a large performance overhead when the system scales up.

In our runtime design, we introduce a *worker-initiated* idle threads handling mechanism. The key idea of our approach is to pass the duty of waking-up the idle threads to the worker threads, thus freeing the master thread of this responsibility and avoiding the need of using mutex locks. Specifically, each worker thread maintains a private message queue. When the master thread assigns a new microtask, it simply puts the new microtask into the queue. When a worker thread finishes its execution, it will check for a new microtask from its private queue. Therefore worker threads do not need to compete for the mutex lock and there is also no signaling overhead associated with inter-process communication.

It is still possible that this technique will increase the workload for the worker threads. However, such an overhead will not count into the critical path, as the communication is initialized by the idle worker threads instead of the busy master thread. In reality, costs with respect to this kind of polling operation is negligible since they perform a read operation from a local message queue and no inter-process communication takes place. Furthermore, we also introduce an interval, δ, between two polling attempts. The value of δ could be based on either randomness or the Poisson distribution. In our evaluation, we choose a heuristics value which equals to the average overhead of the runtime to finish a microtask. Therefore, the idle threads can sleep around until next microtask available. This approach outperformed the vendor-specific OpenMP implementation that will be discussed in section 6.

4.2 Memory Model

OpenMP specifies a shared memory, relaxed-consistency model [9]; i.e., threads access the same, global shared memory, while could have their own temporary view of private data until a synchronization point is reached where the private data is then written back to the global memory. The shared memory, relaxed-consistency model is trivial to achieve in general purpose CPUs where a large shared memory within a compute node with cache-coherent hardware modules always exist. However, the memory hierarchy for embedded systems is much more complicated. The cache coherency is not automatically supported by the embedded hardware. Moreover, they may also consist of on-chip and off-chip shared memory along with local memory with separate address space.

We use MRAPI memory primitives to manage the OpenMP memory. For the non-cache coherent embedded systems, shared data is stored into the MRAPI *shared memory*, which maps into the on-chip/off-chip shared memory. We configure the local cache as the scratchpad memory that is mapped into the MRAPI *remote memory* to store private data. Data transfers between *remote memory* is achieved through DMA, software cache or RapidIO, which will be handled by the *remote memory* implicitly.

4.3 Synchronization

In OpenMP, synchronization primitive consists of *barrier, critical, single* and *master*. *Barrier* synchronizes all threads in the team, while the *critical* region specifies only one thread that can execute at a time. The *single* construct that defines the region can only be executed by one thread, while *master* specifies that region be executed only by the master thread. As we discussed before, we utilize the MRAPI synchronization primitives to implement OpenMP synchronization directives.

4.3.1 Barrier Implementation

OpenMP heavily relies on barrier operations to synchronize threads. Implicit barriers are required at the end of the parallel region; they are also used implicitly at the end of the work-sharing construct. Explicit barriers are used by the OpenMP developers to synchronize the threads in the team. An efficient barrier implementation is therefore essential to achieve good performance and scalability.

In our runtime design, we employ the tournament algorithm [29] to implement the OpenMP barrier. Compared to the traditional centralized barrier and blocking barrier algorithms that require $O(n)$ operations on critical path, the tournament barrier only takes $log(n)$ operations, where n is the number of processes. As a result, it is clear to see that the tournament barrier is able to offer better scalability.

Algorithm 1: Tournament Barrier Algorithm

```
Data: round_t rounds[p][log P]
Initialization{...};
round_t round=current_thd→myround;
while 1 do
    if round→role & LOSER then
        round→opponent = current_thd→sense;
        while champion_sense ≠ current_thd→sense do
            ;
        break;
    else if round→role & WINNER then
        while round→flag ≠ current_thd→sense do
            ;
    else if round→role & CHAMPION then
        while round_flag ≠ current_thd→sense do
            ;
        champion_sense = current_thd→sense;
        break;
```

Algorithm 1 shows the tournament barrier. The idea of the tournament algorithm comes from the tournament game. Two threads play against each other in each game. The role of each thread, i.e. the winner, loser or champion is predefined.The winners from each round play against each other until there is a champion, who will wake-up all threads and release the barrier. We will evaluate the scalability in section 6.

4.3.2 Critical, Single and Master Implementation

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 that thread having the flag can execute that *single* region.

The *master* construct specifies 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 id of the master thread.

4.4 Work-sharing, Scheduling and Runtime library routines Implementation

OpenMP also has a *work-sharing* construct that 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

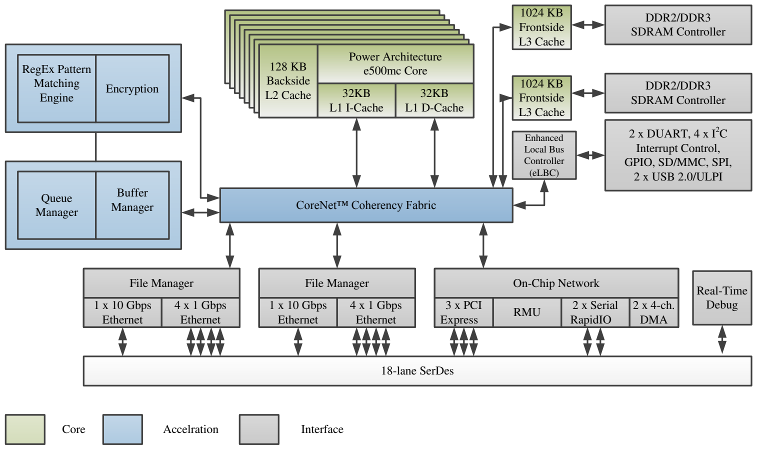


Figure 2. Block diagram of the Freescale QorIQ P4080 multicore platform.

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.

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, *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 discuss the compilation strategies on two Freescale embedded platforms, the corresponding source-to-source translation and the code generation process.

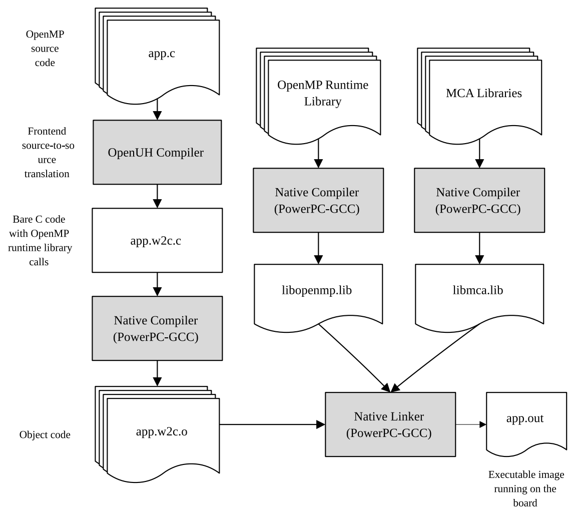
5.1 Architecture Overview

We implemented the *libEOMP* on two Freescale state-of-the-art multicore platforms, one is Freescale P1022 Reference Design Kit (RDK), a dual-core Power Architecture^{TM} multicore platform, while the other one is Freescale QorIQ P4080 eight-core platform [3].

Figure 2 shows the block diagram of the Freescale P4080 platform, which is an eight-core Power Architecture^{TM} e500mc processors supporting dual-issue, out-of-order instructions with superscalar. It has three levels of cache hierarchy: 32KB I/D L1, with 128KB L2 private to each core, and 2M shared L3. It also has two 64-bit DDR2/DDR3 SDRAM memory controllers with ECC and interleaving support. Freescale P1022 RDK has similar configuration, except that it has a dual-core Freescale e500v2 processor with 32KB I/D L1 and 256KB shared L2.

5.2 Compilation Overview

Figure 3 shows the overview of the compilation process. We use the OpenUH compiler [27] as the front-end to perform a source-to-source translation. OpenUH is a branch of the open-source Open64 compiler suite for C, C++ and FORTRAN 95/2003. It



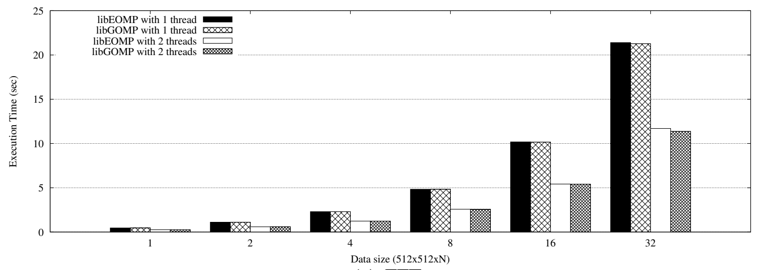
OpenMP source code
app.c
OpenMP Runtime Library
MCA Libraries
Frontend source-to-source translation
OpenUH Compiler
Native Compiler (PowerPC-GCC)
Native Compiler (PowerPC-GCC)
Bare C code with OpenMP runtime library calls
app.w2c.c
libopenmp.lib
libmca.lib
Native Compiler (PowerPC-GCC)
Object code
app.w2c.o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

Figure 3. Overview of the cross-compilation process.

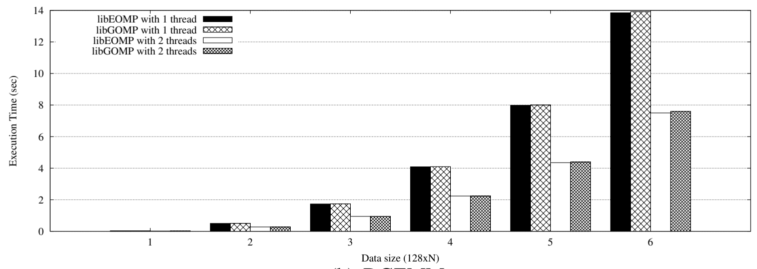
also supports OpenMP 3.0 and almost 3.1, Co-array FORTRAN and UPC. The OpenUH transforms OpenMP pragmas into an intermediate file, called *whirl*, mainly by two steps: *omp_prelower* and *lower_mp*. The former performs the preprocessing while the latter performs the major translations. It also has another translator, called *whirl2c and whirl2f*, that translates the IR to back-end compatible C and FORTRAN code. Then, the generated bare C code with the OpenMP runtime function calls will be fed into the back-end native compiler which generates the object files. For our evaluation, the back-end compiler is the Power Architecture GCC compiler toolchain for both Freescale e500v2 and e500mc processors. During the linking phase, the linker will link all the object codes together with the OpenMP runtime library and MCA libraries to generate the executable files for the target architecture.

6. Performance Evaluation

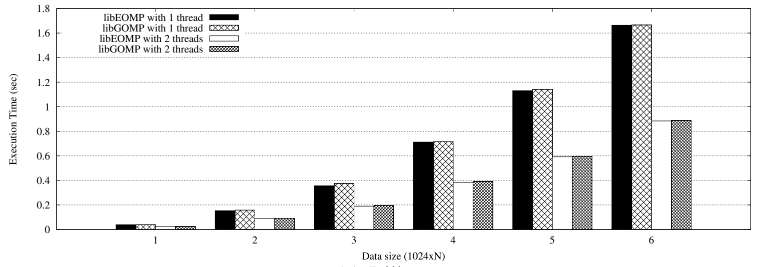
In this section, we evaluated the *libEOMP* for portability and performance on two Freescale platforms. For our experimental purposes, we use the MRAPI implementation provided by Freescale



(a) FFT



(b) DGEMM



(c) Dijkstra

Figure 4. Execution time of FFT, DGEMM and Dijkstra on Freescale P1022 RDK.

Semiconductor Inc.. We have considered two benchmarks, ranging from micro benchmarks to real embedded applications, EPCC micro benchmarks [12] and MiBench [20]. 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 platforms. We calculate the total execution time and the speedup achieved in both *libEOMP* and *libGOMP*.

The experiments include two phases. At the first step we evaluated the *libEOMP* on the Freescale P1022 Reference Design Kit (RDK). As the *libEOMP* is built on top of the MCA APIs, our initial focus has been to ensure the interface is functionally complete and correct to allow OpenMP runtime to target on.

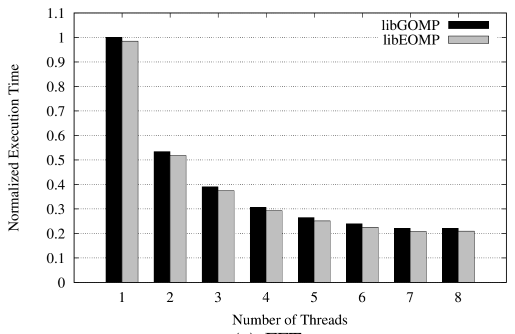
As is shown in Figure 4, we initially compare the performance *libEOMP* over *libGOMP* on the Freescale P1022 RDK platform. Although our purpose in this stage is not to achieve the best performance, the results show that the additional MCA layer does not incur any significant performance penalty. The detail can be found in our prior work [36]. In this paper, we have performed extensive research and developed newer strategies in the runtime design. Specifically, as discussed in Section 4, we designed a new elastic thread pool and worker-initiated idle threads handling strategies. We explored and implemented a new tournament barrier algorithm to achieve better scalability. We evaluated the performance on the eight-core Freescale QorIQ P4080 platform.

Table 2. Relative overhead of libEOMP over libGOMP.

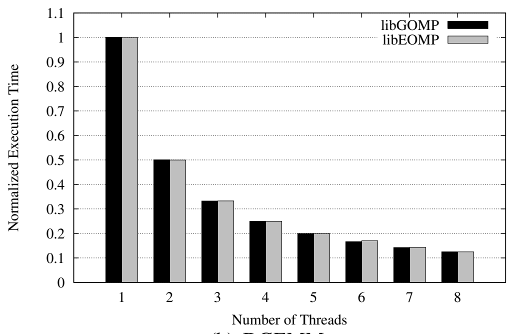
Directive	**2 Threads**	**4 Threads**	**8 Threads**
PARALLEL	0.97	0.96	0.93
FOR	1.01	1.01	1.01
PARALLEL FOR	0.98	0.97	0.94
BARRIER	1.00	1.00	0.97
SINGLE	0.07	0.15	0.46
CRITICAL	1.03	1.04	1.05
REDUCTION	1.01	1.04	1.09

6.1 Overhead Evaluation

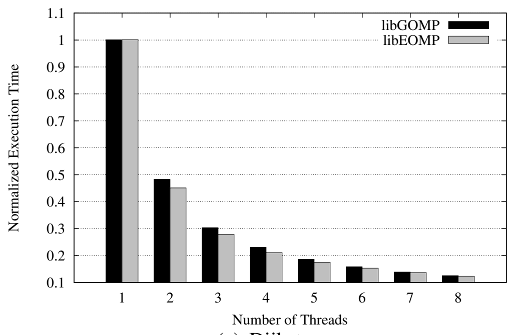
We use EPCC micro benchmark [12] to evaluate the overheads associated with our runtime. EPCC is a set of low-level benchmarks that measures the overhead associated with OpenMP directives. Table 2 shows the percentage overhead of *libEOMP* over *libGOMP* while using different number of threads. Overall, we see that the performance of the two runtime systems is quite competitive. The absolute time difference is actually less than several microseconds but this is barely noticeable by programmers. For the *parallel* construct *libEOMP* even outperforms *libGOMP*. It is because our optimized thread pool approach along with the worker-initiated idle threads handling approach facilitates the *parallel* implementation to perform better. Moreover, better performance with increased number of threads also confirms that our approach scales well. With respect to the *barrier* construct, since we have used the scalable



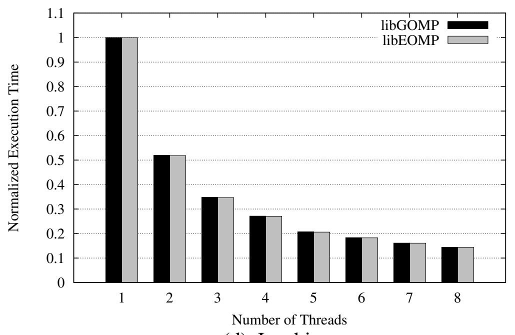
(a) FFT



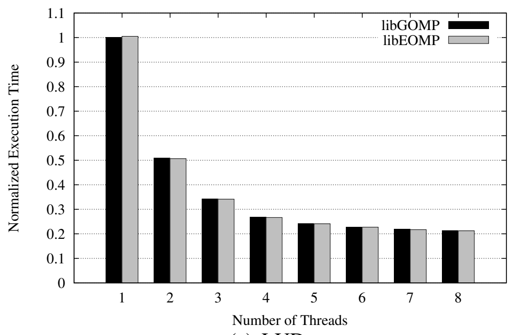
(b) DGEMM



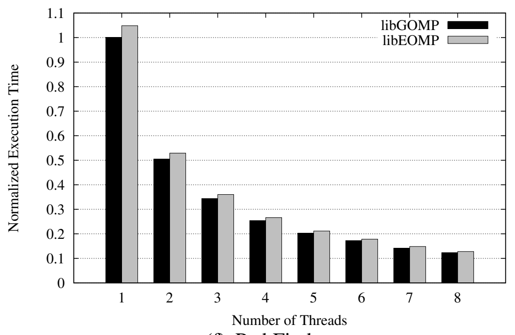
(c) Dijkstra



(d) Jacobi



(e) LUD



(f) PathFinder

Figure 5. Comparison of execution time with *libEOMP* and *libGOMP* under different number of threads.

tournament barrier algorithm, we see that *libEOMP* scales better than *libGOMP*. We also notice that *single* directive in *libEOMP* performs much better than *libGOMP*. Although constructing the *libEOMP* framework is the central theme of this paper and we have not really concentrated on obtaining the optimum performance yet, it is encouraging to see from the table that our approach *libEOMP* can perform better than the vendor-specific approach *libGOMP*.

6.2 Benchmark Evaluation

We evaluated *libEOMP* using several real-world embedded applications chosen from Mibench [20] and Rodinia [15] benchmark suite. Figure 5 shows the normalized execution time of both *libEOMP* and *libGOMP* on the Freescale QorIQ P4080 platform. The results show that *libEOMP* performs competitively over *libGOMP*, which is consistent with the investigation illustrated in Table 2.

With respect to FFT and Dijkstra, *libEOMP* outperforms *libGOMP*. The reason is that these applications are embarrassingly parallel and we have mainly used the *parallel* constructs in the applications. Table 2 already showed that *libEOMP* outperforms *libGOMP* for the *parallel* construct. For DGEMM, Jacobi and LUD, *libEOMP* performed similar to that of *libGOMP* since these applications consist of various dependencies within the program code. Therefore we had to use a combination of OpenMP constructs in order to parallelize the code including synchronization constructs. For the PathFinder we see that the *libEOMP* does not perform as well as the *libGOMP* at one thread, but is quite close to *libGOMP* as the number of threads increases, which shows good scalability.

To summarize, in this paper we have focused on demonstrating that OpenMP could be targeted for embedded systems using MCA APIs. We have also shown that our runtime *libEOMP* can perform as well as the native OpenMP runtime library customized for a specific platform, *libGOMP*. However, as our OpenMP runtime is built on top of MCA APIs, it is independent with any OS or architecture. Thus our runtime is highly portable, as is shown in the experiments. We believe that fine tuning our runtime system further could lead us to even better performance results.

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 demonstrate that *libEOMP* not only performs as good as the optimized vendor-specific approach but also offers adequate portability and productivity.

As a near future work, we plan to explore challenges posed by a heterogeneous platform to *libEOMP*. We plan to investigate minor language extensions to OpenMP that will facilitate usage of heterogeneous cores. This simple extension could be used to offload work from general purpose processors to a specialized processor. We also plan to explore task-based applications that are of interest to embedded developers.

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).

References

[1] TMDXEVM6678L EVM Technical Reference Manual Version 1.0, Literature Number: SPRUH58. URL `http://wfcache.advantech.com`.

[2] Data Communication and Synchronization Library for Cell Broadband Engine Programmers Guide and API reference, Version 3.0. URL `http://moss.csc.ncsu.edu/~mueller/cluster/ps3/SDK3.0/docs`.

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

[4] The Multicore Association. URL `http://www.multicore-association.org`.

[5] A Case For MCAPI: CPU-to-CPU Communications in Multicore Designs. URL `http://www.mentor.com/`.

[6] Multicore Resource API (MRAPI) Specification, Version 1.0. URL `http://www.multicore-association.org`.

[7] The Objective-C Programming Languages. URL `http://developer.apple.com`.

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

[9] OpenMP Application Program Interface, Version 3.1, . URL `http://www.openmp.org`.

[10] Polycore MCAPI Offers ThreadX RTOS Support. URL `http://www.eetasia.com`.

[11] J. Auerbach, D. F. Bacon, I. Burcea, P. Cheng, S. J. Fink, R. Rabbah, and S. Shukla. A Compiler and Runtime for Heterogeneous Computing. In *Proceedings of DAC'12*, pages 271–276, NY, USA, 2012. ACM. ISBN 978-1-4503-1199-1. doi: 10.1145/2228360.2228411. URL `http://doi.acm.org/10.1145/2228360.2228411`.

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

[13] 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.

[14] 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. doi: 10.1109/IPDPS.2009.5161107.

[15] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of IISWC'09*, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-5156-2. doi: 10.1109/IISWC.2009.5306797. URL `http://dx.doi.org/10.1109/IISWC.2009.5306797`.

[16] P. Cooper, U. Dolinsky, A. F. Donaldson, A. Richards, C. Riley, and G. Russell. Offload: Automating Code Migration to Heterogeneous Multicore Systems. In *Proceedings of HiPEAC '10*, pages 337–352. Springer-Verlag, 2010.

[17] F. Garcia and J. Fernandez. POSIX Threads Libraries. *Linux J.*, 2000, 2000.

[18] M. Garland, M. Kudlur, and Y. Zheng. Designing a Unified Programming Model for Heterogeneous Machines. In *Proceedings of SC' 12*, pages 67:1–67:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. ISBN 978-1-4673-0804-5. URL `http://dl.acm.org/citation.cfm?id=2388996.2389087`.

[19] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA Computing in the Streams-C High Level Language. In *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, pages 49–56. IEEE, 2000.

[20] 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.

[21] T. D. Han and T. S. Abdelrahman. hiCUDA: High-Level GPGPU Programming. *IEEE Transactions on Parallel and Distributed Systems*, 22:78–90, 2011. ISSN 1045-9219. doi: http://doi.ieeecomputersociety.org/10.1109/TPDS.2010.62.

[22] 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.

[23] J. He, W. Chen, G. Chen, W. Zheng, Z. Tang, and H. Ye. OpenMDSP: Extending OpenMP to Program Multi-Core DSP. In *Proceedings of PACT '11*, pages 288–297. IEEE, 2011.

[24] 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. ISBN 978-1-4673-0804-5. URL `http://dl.acm.org/citation.cfm?id=2388996.2389032`.

[25] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *Proceedings of SC '10*, pages 1–11. IEEE Computer Society, 2010.

[26] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 101–110, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-397-6. doi: 10.1145/1504176.1504194. URL `http://doi.acm.org/10.1145/1504176.1504194`.

[27] 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.

[28] P. Martin. An Analysis of Random Number Generators for a Hardware Implementation of Genetic Programming using FPGAs and Handel-C. In *Proceedings of the genetic and evolutionary computation conference*, pages 837–844. Morgan Kaufmann Publishers Inc., 2002.

[29] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, Feb. 1991. ISSN 0734-2071. doi: 10.1145/103727.103729. URL `http://doi.acm.org/10.1145/103727.103729`.

[30] 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.

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

[32] A. Reid, K. Flautner, E. Grimley-Evans, and Y. Lin. SoC-C: Efficient Programming Abstractions for Heterogeneous Multicore Systems on Chip. In *Proceedings of CASES ' 08*, pages 95–104. ACM, 2008.

[33] 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.

[34] A. Sbîrlea, Y. Zou, Z. Budimlíc, J. Cong, and V. Sarkar. Mapping a Data-flow Programming Model onto Heterogeneous Platforms. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, pages 61–70, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1212-7. doi: 10.1145/2248418.2248428. URL `http://doi.acm.org/10.1145/2248418.2248428`.

[35] D. W. Walker, D. W. Walker, J. J. Dongarra, and J. J. Dongarra. MPI: A Standard Message Passing Interface. *Supercomputer*, 12:56–68, 1996.

[36] C. Wang, S. Chandrasekaran, B. Chapman, and J. Holt. libEOMP: A Portable OpenMP Runtime Library Based on MCA APIs for Embedded Systems. In *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '13, pages 83–92, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1908-9. doi: 10.1145/2442992.2443001. URL `http://doi.acm.org/10.1145/2442992.2443001`.