2015 IEEE International Parallel and Distributed Processing Symposium Workshop

OpenMP-MCA:Leveraging Multiprocessor Embedded Systems using industry standards

Peng Sun, Sunita Chandrasekaran and Barbara Chapman
Department of Computer Science, University of Houston, Houston, TX, 77004, USA
{psun5,sunita,chapman}@cs.uh.edu

***Abstract*—Multicore embedded systems are rapidly emerging. Hardware designers are packing more and more features into their design. Introducing heterogeneity in these systems, i.e. adding cores of varying types does provide opportunities to solve problems in different aspects. However, this presents several challenges to embedded system programmers since software is still not mature enough to efficiently exploit the capabilities of the emerging hardware rich with cores of varying types.**

Programmers still rely on understanding and using low-level hardware-specific API. This approach is not only very time-consuming but also tedious and error-prone. Moreover, the solutions developed are very closely tied to a particular hardware raising significant concerns with software portability. What we need is an industry standard that will enable better programming practices for both current and future embedded systems. To that end, in our project, we have explored the possibility of using existing standards such as OpenMP that provides portable high-level programming constructs along with another industry-driven standard for multicore systems, MCA. For our work, we have considered the GNU compiler since it is the compiler that mostly used in the embedded system domain facilitating open source development. We target a platform consisting of twelve PowerPC e6500 64-bit dual-threaded cores. We create a portable software solution by studying the GNU OpenMP runtime library and extending it to incorporate MCA libraries. The solution abstracts the low-level details of the target platform and the results show that the additional MCA layer does not incur any overhead. The results are competitive when compared with a proprietary toolchain.

***Keywords*—OpenMP; Runtime Library; Embedded Systems;**

1. Introduction

Programming embedded systems is a real challenge. While developing software solutions for such platforms, portability is a major concern since vendors use proprietary, vendor-specific software toolchains that are tied to specific platforms. We have been devising methods to create a novel software toolchain that will abstract the low-level details of the hardware and provide a generic programming interface that could be portable across more than one platform[1], [2], [3]. This work has attracted broad interest in the relevant industry sectors since it promises simplified and faster software development process[4].

The development process becomes more and more challenging when the multicore systems contain cores of more than one type. Such heterogeneous embedded multicore systems usually consist of different kinds of computation units of different ISAs; these systems also have memory hierarchies that are different from a traditional X86 system. For example, TI's Keystone II architecture[5] consists of several ARM and DSP cores, Nvidia Tegra[6] SoCs consists of several ARM and GPU cores; these cores have local memories(L1/L2) and also shared memory. Besides the rapid increased complexity of the emerging heterogeneous embedded systems, the lack of portable and high-level programming tools will exacerbate the difficulties to program on heterogeneous multicore embedded systems and cause delay in software development process. Time to market drives profitability. Thus, the absence of usable programming approaches means there is a very high barrier to such systems' entry into the world of accelerated computing.

It is important for embedded systems industry to have a widely used industry standard-based approach to program such complex systems. Establishing standard-based approaches could help form fundamental software toolchain that could be used across platforms.

OpenMP[7] has been a simpler, easy-to-use, de-facto programming model for shared memory systems for many years. The model provides fully-featured pragma-based directives for programmers to explore the potential data parallelism and task parallelism incrementally from serial codes. In July 2013, OpenMP released an OpenMP 4.0 [8] API specification with significant new standard features that included support for accelerators, SIMD constructs to vectorize both serial as well as parallelized loops, error handling, thread affinity, tasking extensions and several others. This is good news for embedded systems since OpenMP is shifting from being solely focused on shared memory systems, which is not a typical scenario with embedded systems. However, due to the high complexity level of both hardware and software support in embedded systems, there are still only a few embedded vendors who could adopt the accelerator support in OpenMP on their devices[9]; still tedious to utilize OpenMP on the platform. Although OpenMP's suitability for embedded systems is still under research and exploration, it is one of the strongest candidates to move serial codes to parallel systems and parallelize codes for accelerated systems.

Another widely-used industry standard for embedded platforms is from the *Multicore Association(MCA)*[10] that was founded by a group of leading vendors from the semiconductor industry. The working group of MCA conducted a set of industry standards that could be used as the infrastructure to improve time to market for applications, including resource management API, communication API, and task management API. They are independent of any system architectures and operating systems, providing a unique interface for embedded developers to program portable software. However, MCA API is still low level and library-based, by providing a higher-level parallel programming model such as OpenMP, the embedded developers could efficiently parallelize applications and fully

978-1-4673-7684-6/15 $31.00 © 2015 IEEE
DOI 10.1109/IPDPSW.2015.13

explore the hardware potential.

In this paper, we have expanded on our previous research work of investigating the adaptability of OpenMP and its translation to MCA for multicore embedded systems.

This paper makes the following contributions:

- Explore the suitability of OpenMP programming model for embedded systems by extending its runtime support with MCA API, the embedded system industry standard API;
- Abstract the low-level details of T4240RDB platform (PowerPC cores) by extending the MCA resource management API to handle thread-level node management and memory management
- Demonstrate the effectiveness of the new toolchain porting parallel applications on the embedded platform

The organization of the paper: Section 2 gives an overview of the OpenMP programming model and the MCA libraries. In Section 3 we review the state-of-the-art techniques to program multicore embedded systems. Section 4 provides an overview of the target platform chosen for this work and the process of setting the platform up. Section 5 discusses the design and implementation of extending GNU OpenMP runtime library using MCA libraries, Section 6 presents the performance analysis of the GNU OpenMP runtime library with MCA libraries. Section 7 concludes the paper and discusses future work.

2. Background

In this section, we briefly introduce OpenMP and MCA API highlighting some of the key features and its usability especially for the latter since the former is already a well-known model.

A. OpenMP

OpenMP [7] is a well-known portable and scalable programming model that facilitates programmers with simple, but versatile interface for developing parallel applications. By inserting pragmas into an original serial program, the model makes it easy to leverage underlying hardware rapidly and effortlessly. OpenMP's primary model of parallel execution is fork - join.

```
1 void sum(int n, float *a, float *b)
2 {
3  int i;
4 #pragma omp parallel for
5  for (i=1; i<n; i++)
6  b[i] = (a[i] + a[i-1]) / 2.0;
7 }
```

Listing 1: OpenMP Example

As shown in the code snippet in Listing 1, the program begins with a single thread execution, when encounters the parallel region marked as *#pragma omp parallel for*, the master thread will create (fork) a team of worker threads to compute the workloads in parallel. After the computation, all the forked worker threads will synchronize and terminate (join), leaving only the master thread to continue.

Although programmers in the general-purpose domain have conveniently enjoyed parallelizing serial codes using OpenMP for many years, embedded world has not been able to exploit this luxury, yet. Reasons behind are that embedded systems lack some of the features that are in general-purpose computers, which are essential for an OpenMP execution. OpenMP implementations rely heavily on threading libraries and operating system facilities. However, the embedded systems domain, may lack such features; some embedded systems do not even have an OS leading to programming on bare-metal. These systems have usually limited hardware resources so they cannot afford thread oversubscription. It might seem simple enough just to offload most compute-intensive portions of the code to the accelerator or a specialized processor using conventional methods such as remote procedure calls, but alternate approaches such as dataflow-oriented have proved to be more efficient due to lesser application deadlocks; for devices like FPGAs[11]. This requires that programmers rethink ways to program codes for the type of specialized hardware resources an embedded device can offer.

As we can see, it remains a challenge to create an efficient programming (almost universal) interface for the embedded world without having substantial details of the hardware design and approaches to optimize an application for such platforms.

B. MCA Libraries

To address some of the programming issues for embedded platforms, the Multicore Association (MCA) was founded by the semiconductor, embedded software companies, and academies establishing industry standards for programming embedded systems. Among several functionality, the Multicore Association API[10] includes *MRAPI* (the multicore resource management API), *MCAPI* (the multicore communication API) and *MTAPI* (the multicore task management API). With these three sets of API designed purely for the embedded systems, MCA API aims to abstract the lower level hardware details and provide a unique API interface across platforms, to turbo the software development.

MCAPI is designed to capture the core elements of communication and synchronization required for closely distributed embedded systems, as a message-passing API. Industry vendors such as [12][13] have also provided MCAPI support for their products.

The MCA Tasking API is aiming to manage the task level parallelizations of multicore embedded platforms. It includes complete support of task life-cycle, with optimization of task synchronization, scheduling, and load balancing. Siemens recently released its open-source MTAPI implementation[14] as a part of the EMBB[15] library.

MRAPI seeks to handle resource management challenges of the most critical hardware resources on real products, including shared memory, remote memory, synchronization primitives, and metadata, for both SMP and AMP architectures. MRAPI can support virtually any number of cores, even each with a different instruction set, same or different OSes. Besides that, MRAPI could allow coordinated concurrent access to the systems resources by deploying the synchronization primitives for embedded systems that with limited hardware resources. The MRAPI could support a varies of operating

systems, including Embedded Linux, RTOS, and even Bare-Metal systems.

We would like to summarize some of the fundamental concepts of MRAPI since we will be using these functionality in our software design description.

1) Domain and Nodes: The MRAPI systems are composed of one or more *domain*; each domain is considered as a unique system global entity. An MRAPI *node* is an independent unit of execution, and an MRAPI domain will comprise a team of MRAPI nodes. Each node could map to any execution unit, such like a process, a thread, a thread-pool or a hardware accelerator.

2) Memory Primitives: With the concept of heterogeneity in mind, MRAPI supports two different memory models, the shared memory, and the remote memory. Shared memory primitives could allow users to manage the on-chip or off-chip shared memory directly, with assigned attributes. Unlike the Linux shared memory, which could only be accessed within one operating system's entity, the MRAPI shared memory could be accessed by different nodes running different OSes. The remote memory model enables the access of distinct memories. This model could be physically consecutive or not direct access, for the latter case, some other methods like DMA will need to be used to access to the remote memory. By providing unique API interfaces, MRAPI hides all those memory access details from the end users.

3) Synchronization Primitives: MRAPI offers a set of synchronization primitives including *Mutexes*, *Semaphores* and *Reader/Writer locks*. These synchronization primitives guarantee the MRAPI nodes properly access to the shared resources, to avert data race or race conditions.

4) System Resource Metadata: MRAPI specification provides a facility of retrieving the system metadata in a format of the resource tree, providing details of resources availability for the target system.

Note: In this paper, we have limited our exploration to MRAPI. Since we have not explicitly explored the data movement between the PowerPC cores of the platform to its specialized coprocessors, we have not used MCAPI.

3. Related Work

This section discusses related efforts on programming techniques for multicore embedded systems.

A. Programming Language and Library Efforts

Every embedded system is different. A particular programming challenge with such a system is working close to the hardware. Due to the necessity of tackling low-level details, C is the most used language for embedded application developments. Besides, vendors typically offer Software Development Kits (SDKs), specifically fit their own devices; further makes programming embedded devices not portable and error-prone.

Language extensions have been proposed[16][17] to abstract the low-level operations for a certain set of functionality on embedded systems. OpenCL and CUDA work closer to the platform. OpenCL[18] is a programming language and framework allow programming on heterogeneous platforms[19], [20] . However, the model is low-level and not easy to use. CUDA[21] is a popular programming model for GPU programming, but this is proprietary and limited to NVIDIA devices. There are several efforts that use CUDA to program NVIDIA devices such as Tegra mobile processor, for embedded applications, such efforts include [22], [23], [24]. HSA[25] has been recently introduced by AMD, ARM, and other semiconductor companies; it aims to provide hardware and software solutions to reduce disjoint memory operations for heterogeneous architectures. Besides, there are a number of programming languages designed for other devices such as FPGAs[26] and several others. CAPH[27] implements stream-processing applications on FPGAS.

Many different facets of MCA are currently explored such as[28] explores MCAPI abstraction also for hardware components to allow portability between software and hardware. MCAPI was implemented on FPGA, and MCAPI's suitability for MP-SoC is discussed in[29]. A proof-of-concept of MRAPI on a Xilinx Virtex-5 FPGA has been implemented in[30]. High-level languages such as UML includes complex channel semantics and provides automatic code generation for interconnection and deployment of system components based on MCAPI[31].

In this paper, we have investigated how our previous work[1], [2] could be extended to support more than just one platform; demonstrating portability and enhanced an OpenMP runtime component of a widely-used compiler toolchain in the embedded world; GNU OpenMP. Previously, we had also explored the usage of MCAPI API for a heterogeneous platforms [3] to create communication between the host and a specialized bare-metal accelerator; this was quite a challenging task.

B. High-level Parallel Programming model

OpenMP programming model has been used for several embedded systems, such as [32], [33], [34]. TI[9] supports accelerator features from the OpenMP 4.0 standard. OpenMDSP[35], an extension of OpenMP, is designed for multi-core DSPs to fill the gap between the OpenMP memory model and the memory hierarchy of multi-core DSPs. However, the approach is not generic enough to be used for other systems. The authors in [36] conducts similar research. In[37], the authors discuss an OpenMP compiler for use of distributed scratchpad memory. Some tools, which can automatically generate OpenMP directives from serial C/C++ codes, and run on general computer and embedded system, have been discussed in[38]. OmpSs[39] programming model helps program on clusters of GPUs, and it has been extended to use CUDA and OpenCL recently[40].

OpenACC[41] is another popular pragma-based programming model to program to heterogeneous platforms consisting of accelerators such as GPUs, related work includes[42], [43]. However, to the best of our knowledge, there is no work on targeting OpenACC on embedded systems.

We see that there are a number of models, language extensions to program accelerators and heterogeneous systems[44] but their adaptability and suitability for embedded platforms is questionable for the several reasons we have previously

discussed. In our work we have combined a high-level widely-used programming model, OpenMP with an embedded-specific industry standard, MCA and targeted a platform with PowerPC cores.

4. Experimental Setup

For our work, we have chosen T4240RDB platform from *Freescale*'s QorlQ family. Since this hardware platform is not a typical X86 platform, we believe it is important for the reader to know about the features of the platform and its system setup procedures before we discuss the design and implementation of software developed.

A. T4 Processor

The T4240RDB platform features twenty-four virtual threads from twelve PowerPC e6500 cores, running at 1.8GHz and providing the rich I/O capabilities. The Freescale T4 processor family is commonly used in networking productions like routers, switches, gateways to fully utilize its combined control, data path support and application layer processing and also for general purpose embedded computing systems. Twelve PowerPC e6500 64-bit dual threaded cores with integrated AlitiVec SIMD processing units are clustered in T4240RDB board, manufactured in a 28nm process. The e6500 core includes a 16 GFLOPS AlitiVec technology execution unit that supports SIMD architecture, to achieve DSP-like performance for math-intensive applications and therefore could be considered to be mapped to the OpenMP 4.0 SIMD support. E6500 cores are clustered to four cores with a shared multibank L2 cache, and the three clusters in T4240RDB board are connected by the CoreNet coherency fabric, sharing a 1.5MB CoreNet (L3) cache. Additionally, it has hardware support for L1 and L2 cache coherency. The design of e6500 cores also deploys many low-power techniques, including pervasive virtualization and cascading power management.

T4240RDB board provides support for small hypervisor for embedded systems based on Power Architecture technology. By using Freescale embedded hypervisor, we could add a layer of software that enables an efficient and secure partitioning of a multicore system, including partition of a system's CPUs, memory and I/O devices, with each partition capable of executing a different or the same guest operating systems. We plan to use MCAPI to exploit the hypervisor in the near future. Figure 2 illustrates how the Freescale Hypervisor manages the embedded systems hardware and partitions on the system.

B. T4240RDB Setup

Unlike the general-purpose computer, it is a time consuming procedure to boot up an embedded device and configure it to a certain working condition. We generally describe the procedures for setting up the T4240RDB board here to illustrate the efforts need to set an embedded platform.

The T4 board comes with pre-installed u-boot and the embedded Linux image on the NOR flash. And by default it boots from the NOR flash. In the default configuration, the file system will be refreshed for every reset. We analyzed the default configuration and decided to modify the board's configuration to better serve the project.

We configure the T4240RDB board to load the embedded kernel image from TFTP server while u-boot, and deploy an NFS root file system to mount as shown in Figure 3. Trivial File Transfer Protocol (*TFTP*) is a file transfer protocol that allow a client to access a file efficiently onto a remote host [45]. In this project, the TFTP server is configured in one Linux desktop while the T4240RDB board is the client to obtain the image files in u-boot. NFS [46] is a distributed file system protocol that allow a user on a client to access files over a network like accessing the local storage. We also configure the Linux desktop to host the NFS server to the board. By using NFS, the root file system could be customized and be saved in the remote NFS server, while overcoming the limited local hardware resources on board.

C. Comparing T4240RDB with P4080DS

In this subsection, we highlight the differences in the target platform that we had used for our previous work [1] and the platform that we are using for the work discussed in this paper. (Note: Our goal is to provide a software toolchain that could be used across more than one platform and we have substantially improved our previous toolchain to make it suitable for more than just one platform). Previously, we had used a P4080DS processor, that consisted of eight Freescale e500mc PowerPC cores. The e500mc core is compliant with PowerISA v.2.06 and includes hypervisor and visualization functionality. It supports CoreNet communications fabric for connecting cores and data path accelerators. The eight e500mc cores are connected to the CoreNet fabric directly, unlike the e6500 Cores available on the T4240RDB platform, featuring twelve PowerPC cores, that connects four e6500 cores as a cluster sharing L2 cache and then connected to the CoreNet fabric. The size of L1 cache is the same for both processors, i.e. 32KB. However, the L2 cache size is much larger for T4240 than the 128KB of unified backside L2 cache per e500mc core.

5. Design and Implementation

In this section, we discuss the design and implementation details of our project.

A. Multicore Resource Management API extension

MRAPI naturally supports process-level parallelism by mapping MRAPI nodes on processes and utilizing the MRAPI synchronization primitives to synchronize between nodes. However, such parallelism can be too cumbersome for parallelizing embedded systems. The overhead due to launching a process and inter-process communication (IPC) (causing additional context-switching) can be a performance kill. Each of the processes would run their private address space; thus one process could not access the other process's data unless utilize an IPC method. Unlike processes, threads are light-weight. The latter has an advantage due to its lower cost of creation and the ability to exchange large data structures simply by passing pointers rather than copying. Thread synchronization does come with a penalty, and it is a challenge to create a thread-safe multithreaded program. OpenMP designers provide an easy method to write a multithreaded application without requiring the programmer to worry about creating, synchronizing and destroying threads. So in our work, we explore how MRAPI can enhance OpenMP's thread-level support for embedded platforms.

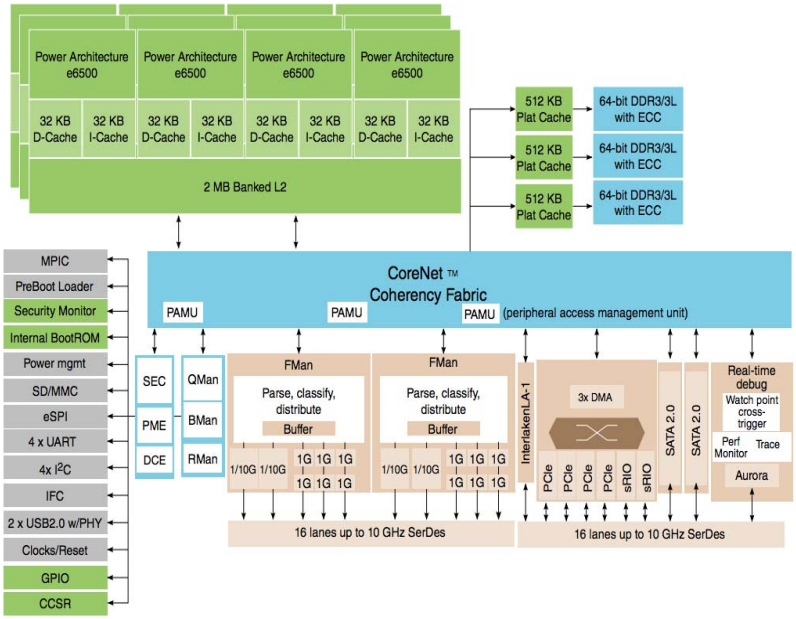

Fig. 1: T4240RDB Block Diagram

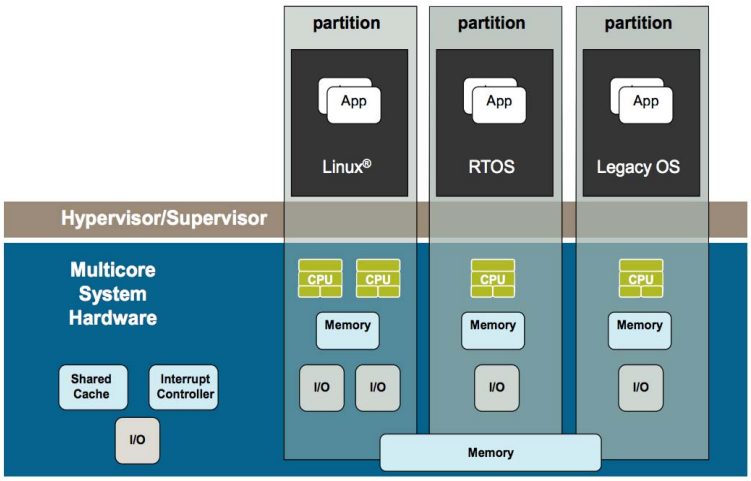

Fig. 2: T4240RDB Hypervisor

1) MRAPI Node Management Extension: We use MRAPI's node initialization process to create threads associated with MRAPI node IDs. MRAPI node initialization process creates new nodes associated with node IDs and registers the related node information in the global MRAPI database that is shared by all the nodes in one domain. (Note: MRAPI features help abstract the hardware resources into four categories: computation entities, memory primitives, synchronization primitives, and system metadata. Such a rich feature set allows MRAPI to be used for process-level and system-level resource management covering a broad range of use cases). Such an approach allows MRAPI to support a team of nodes where one node could be the host, and the other could be the accelerator.

```
typedef struct{
      pthread_t *thread_handle;
      pthread_attr_t *attr;
      void *(*start_routine) (void *);
      void* arg;
} mrapi_thread_parameters_t;

void mrapi_thread_create(
    MRAPI_IN int domain_id,
    MRAPI_IN int node_id,
    MRAPI_OUT mrapi_thread_parameters_t*
     init_parameters,
    MRAPI_OUT mrapi_status_t* status   )
```

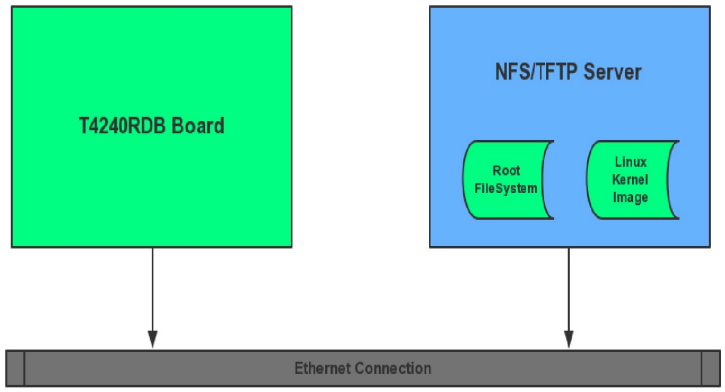

Fig. 3: NFS Development Environment

```
13  {
14      if (mrapi_impl_initialized()){
15          if(mrapi_impl_thread_create(domain_id,
        node_id, init_parameters))
16              *status = MRAPI_SUCCESS;
17      }
18      else{
19          *status = MRAPI_ERR_NODE_NOTINIT;
20      }
21  }
```

Listing 2: MRAPI Node Extension

As shown in Listing 2, the thread creation operation is accomplished for each node calling the *mrapi_thread_create* function. The function will create a worker thread for the node requested, and register the related thread information inside the domain global database for the calling node. This will be associated with the thread created and managed by the node for later use.

2) MRAPI Memory Management Extension: MRAPI shared memory constructs by default maps the memory allocation onto the system level shared memory, which is an Inter Process Communication (IPC) methodology. However, this is not a suitable methodology for OpenMP and the other thread-level parallel computation, even for embedded systems. To tackle this issue, we extend the MRAPI implementation to offer end-users more choices with memory allocation. For instance, most of the OpenMP global shared data is mapped to the process private heap instead of the system level shared memory, enabling better flexible to share among threads. Data mapped onto the heap could be shared by all threads incurred by the same process, which would facilitate the global data movement. Additionally, embedded devices of different types from different vendors typically support varieties of memory allocation methods. By extending the MRAPI memory model to support thread-level memory, we make it more feasible to utilize varying memory features available on different platforms.

B. Enhancing libGOMP with MCA API

In the embedded industry, the GNU compiler is the most widely used compiler. We explored the suitability of MCA libraries with the GNU compiler based OpenMP runtime library, the libGOMP library. (Previously we had built an OpenMP runtime library called *libEOMP*[1] where we had mapped essential resource management functionality of the MCA libraries to an OpenMP runtime library implemented in OpenUH compiler[47]. OpenUH translates OpenMP directives and function calls into multithreaded code for use with a custom runtime library (RTL).) GCC recently released 4.9.1 that provides support for OpenMP 4.0 in the C/C++ compilers and FORTRAN, which is encouraging for multicore embedded programmers.

Compilers translate the high-level OpenMP pragma-based directives into an Intermediate Representation (IR). The directives provide hints to the compiler to perform code transformations so that the serial code be converted into a parallel code. After translating the OpenMP constructs to C-like IR from original pragmas, the large part of the code is built into a separate runtime library, in this case, it is libGOMP. This provides a set of high-level functions that are used to efficiently implement the OpenMP constructs. The OpenMP runtime typically manages the parallel execution by creating worker threads, managing threads pool, and the synchronization among threads. Besides, an efficient OpenMP runtime can also offer productive scheduling, data locality, and workload balancing techniques.

Most of the cases, the systems offer fully featured OSes and a unique multi-threading library available to be effectively utilized by the OpenMP runtime library. Unfortunately, the embedded systems do not necessarily provide these features since they are heavily customized to cater to specific applications. Hence, we use MRAPI as the translation layer for OpenMP to target such complex systems:

(We used GNU compiler's work-in-progress branch, OpenMP-4.0 in our study.)

1) Node Management: We map the MCA-libGOMP thread allocation to the MRAPI node management constructs, with thread creation, and exit handled internally by MRAPI. During runtime, when the OpenMP runtime library needs to fork a team of worker threads, the MRAPI node initialization is called by runtime. The node initialization procedure would include allocating MRAPI node related data structure, register the node information in the domain-wide global database and create a worker thread associated with the node id. During runtime, worker threads will be represented by its coordinate MRAPI node; each of the worker threads is referred by the MRAPI node_id. After the parallel execution, the MRAPI node, and its associated worker thread, will be finalized by the MRAPI routines, i.e. exit the thread, and release the occupied memory space of MRAPI nodes and the associated worker threads.

2) Memory Mapping: In the OpenMP program runtime, a set of global data structures needs to be maintained. For example, each team of nodes would need to keep a block of work share, to be assigned to each node later for computation. In this project, we extend the MRAPI shared memory constructs to be able to allocate thread-level shared data, as shown in Listing 3

```
void *gomp_malloc (size_t size)
{
  mrapi_shmem_attributes_t shm_attr;
  shm_attr.use_malloc = MCA_TRUE;
  mrapi_status_t mrapi_status;
  mrapi_shmem_create_malloc(SHMEM_DATA_KEY, size,&
    shm_attr,&mrapi_status);
  if (mrapi_status == MRAPI_SUCCESS) {
    return shm_attr.mem_addr;
  }
  else
    gomp_fatal("MRAPI failed memory allocation");
}
```

Listing 3: MRAPI Memory Extension

3) Synchronization: The synchronization primitives of MCA-libGOMP has been mapped to MRAPI Mutexes, preventing critical data races and managing accesses to the shared data. Specifically we use the *mrapi_mutex_create* function to create the Mutex object as the initialization step, and map the Mutex lock and unlock functions to MRAPI Mutex routines as well.

```

/* libGOMP Mutex Lock entry */
static inline void
gomp_mutex_lock (gomp_mutex_t *mutex)
{
  int oldval = 0;
  gomp_mutex_lock_slow(mutex, oldval);
}

/* MCA Mutex Lock entry */
static inline void
gomp_mrapi_mutex_lock (gomp_mrapi_mutex_t *mutex)
{
  mrapi_status_t mrapi_status;
  mrapi_status = MRAPI_SUCCESS;
  mrapi_mutex_lock(mutex->mutex_handle,&(mutex->
    mutex_key),MRAPI_TIMEOUT_INFINITE, &mrapi_status
    );
}
#endif
```

Listing 4: MRAPI Mutex in libGOMP

TABLE I: Relative overhead of MCA-libGOMP versus GNU OpenMP runtime

Directive	4	8	12	16	20	24
Parallel	0.98	1.04	0.73	0.98	0.98	1.03
For	1.00	1.10	1.10	1.01	1.31	1.49
Parallel_for	0.99	1.36	1.05	1.03	0.81	0.95
Barrier	0.90	0.93	1.13	0.90	1.48	1.32
Single	0.41	2.39	1.09	0.97	0.99	1.03
Critical	0.99	1.34	0.99	1.19	1.11	0.45
Reduction	0.98	0.97	1.00	0.94	1.07	1.01

Listing 4 illustrates the MRAPI Mutex routine we used to enhance the proprietary libGOMP. MRAPI Mutex routine will map the lock operation for the target system, thus making this low-level operations portable according various set of systems supporting MCA API.

4) Metadata Information: MRAPI metadata constructs have also been utilized within the OpenMP runtime library. We mainly used the MRAPI metadata trees to retrieve the available number of processors online for node/thread management, to better serve the MRAPI nodes and threads allocation and management accordingly.

6. Performance Analysis

In this section, we have conducted experiments to measure the performance of the extended GNU OpenMP runtime library with MCA API, discussion on the overhead and performance results follows.

A. Overhead analysis

To measure possible overheads caused by enhancing GNU OpenMP runtime library to MCA API, we use EPCC[48] to evaluate the MCA-libGOMP. EPCC is a set of programs that measure the overhead of each of the OpenMP directives and evaluates different OpenMP runtime library implementations. Table I lists the results for MCA-libGOMP compared with the proprietary GNU OpenMP runtime. In the table, we normalize the overhead numbers of MCA-libGOMP divided by the original overheads number, to provide a relative performance number in the table; the smaller number indicating fewer overheads. The table illustrates that MCA-libGOMP does not incur major overhead, and it performs even better for some constructs.

With different thread pool sizes, the PARALLEL construct performs better than libGOMP; while the overheads are slight higher than libGOMP for the *for* construct. Thus, the combined *Parallel for* construct has similar overhead performance. The rest of the constructs are also comparable to libGOMP with different size of threads pool. The performance comparison illustrates that we have achieved competitive performance for each of the OpenMP constructs on T4240RDB board while MCA-libGOMP still has more room to be optimized with larger thread pool.

Next we wanted to ensure the correctness of our implementation. For this step, we used our OpenMP validation suite [49] to identify if the enhancements made to the runtime did not cause a code to fail. The results helped determine some bugs,

and we fixed them such as tracing potential issues with a non-functional synchronization primitive in MCA-libGOMP that caused an OpenMP critical construct to fail.

B. Evaluating GNU OpenMP with MCA Library

We then evaluated our MCA-libGOMP implementation using NAS OpenMP benchmarks [50], the results are illustrate in Figure 4. The execution time is in seconds, and the performance comparison is between the MCA-libGOMP runtime library and the proprietary GNU OpenMP runtime library. The NAS OpenMP benchmarks have several different data sets available to choose from. Typically, size S and W, which are the smallest data sets available, could be used to validate the correctness of the compiler being tested and the runtime library. The larger data sets could be used to measure the real performance of the compiler and the OpenMP runtime library. In this project, we chose the size A of NAS benchmarks for our performance measurement. As seen in the Figure 4, runtime for a single thread is measured in several seconds.

We illustrate the performance from a single thread to 24 threads, which is the maximum amount of threads available on the T4 board. Besides the performance comparison, we also measure the speedup rate of both runtime libraries within the same graph.

As shown in Figure 4, the performance of MCA-libGOMP is very comparable to the proprietary GNU OpenMP runtime library. In CG, EP, and IS, the MCA-libGOMP runtime library shows better performance compared to proprietary libGOMP for a certain number of threads. With respect to the speedup rate, most of the benchmarks perform well. Both the OpenMP runtime libraries are close to the ideal speedup rate for benchmark EP. The rest of the benchmarks could achieve speedup around 15 times using 24 threads.

The performance and speedup rates shown in Figure 4 could prove that, by enhancing libGOMP with MCA libraries we did not incur any significant overheads, but provided a portable software toolchain that uses a high-level programming model, OpenMP to create a multithreaded application translated to industry-based standard MCA to target a multicore embedded platform.

7. Conclusion

Sophisticated portable toolchain is a dire necessity for embedded platforms. Such platforms are becoming more and more complicated while using parallel programming for such systems are becoming more and more challenging. The gap between emerging hardware and lack of appropriate software is widening. To address this primary concern, we have explored the usage of a high-level programming model, OpenMP that uses MCA, another industry-based standard that promotes multicore technology. Our work enhances a commonly used embedded industry-preferred OpenMP runtime with MRAPI to create a standardized application program interface for managing resources and providing source-code compatibility allowing applications to be ported to platforms. Adding MCA layer to the software toolchain did not incur any overhead as demonstrated in our design and evaluation discussions. As part of the future work, we will explore the challenges posted by a heterogeneous multicore system to the usability of MCAPI and MTAPI.

Acknowledgment

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

References

[1] C. Wang, S. Chandrasekaran, P. Sun, B. Chapman, and J. Holt, "Portable mapping of openmp to multicore embedded systems using mca apis," in *Proceedings of the 14th ACM SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, pp. 153–162, ACM, 2013.

[2] 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*, pp. 83–92, ACM, 2013.

[3] P. Sun, S. Chandrasekaran, and B. Chapman, "Targeting heterogeneous socs using mcapi," in *TECHCON 2014, in the GRC Research Category Section 29.1*, SRC, September 2014.

[4] "Facing nextgen multicore design challenges." http://www.embedded.com/electronics-blogs/cole-bin/4430047/Facing-nextgen-multicore-design-challenges.

[5] "Keystone dsp + arm guide." http://www.ti.com/lit/ds/sprs866e/sprs866e.pdf.

[6] N. Corpration, "Nvidia tegra mobile processor," *URL http://www.nvidia.com/object/tegra.html, year=2013*.

[7] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[8] "OpenMP application program interface version 4.0." http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, 2013.

[9] E. Stotzer, A. Jayaraj, M. Ali, A. Friedmann, G. Mitra, A. P. Rendell, and I. Lintault, "Openmp on the low-power ti keystone ii arm/dsp system-on-chip," in *OpenMP in the Era of Low Power Devices and Accelerators*, pp. 114–127, Springer, 2013.

[10] "Multicore Association Website." http://www.multicore-association.org.

[11] P. Athanas, D. Pnevmatikatos, and N. Sklavos, *Embedded Systems Design with FPGAs*. Springer, 2013.

[12] "Polycore implementing a standard." http://polycoresoftware.com/news/implementing-a-standard.

[13] "An open source implementation of the mcapi standard." https://bitbucket.org/hollisb/openmcapi/wiki/Home.

[14] "Siemens produces open-source code for multicore acceleration." http://www.techdesignforums.com/blog/2014/10/31/siemens-produces-open-source-code-multicore-acceleration/.

[15] "Embedded multicore building blocks (embb)." https://github.com/siemens/embb.

[16] 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*, pp. 95–104, ACM, 2008.

[17] 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*, pp. 337–352, Springer-Verlag, 2010.

[18] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, p. 66, 2010.

[19] D. A. Augusto and H. J. Barbosa, "Accelerated parallel genetic programming tree evaluation with opencl," *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 86–100, 2013.

[20] T. Suganuma, R. B. Krishnamurthy, M. Ohara, and T. Nakatani, "Scaling analytics applications with opencl for loosely coupled heterogeneous clusters," in *Proceedings of the ACM International Conference on Computing Frontiers*, p. 35, ACM, 2013.

[21] NVIDIA, "Cuda C Programming Guide." http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

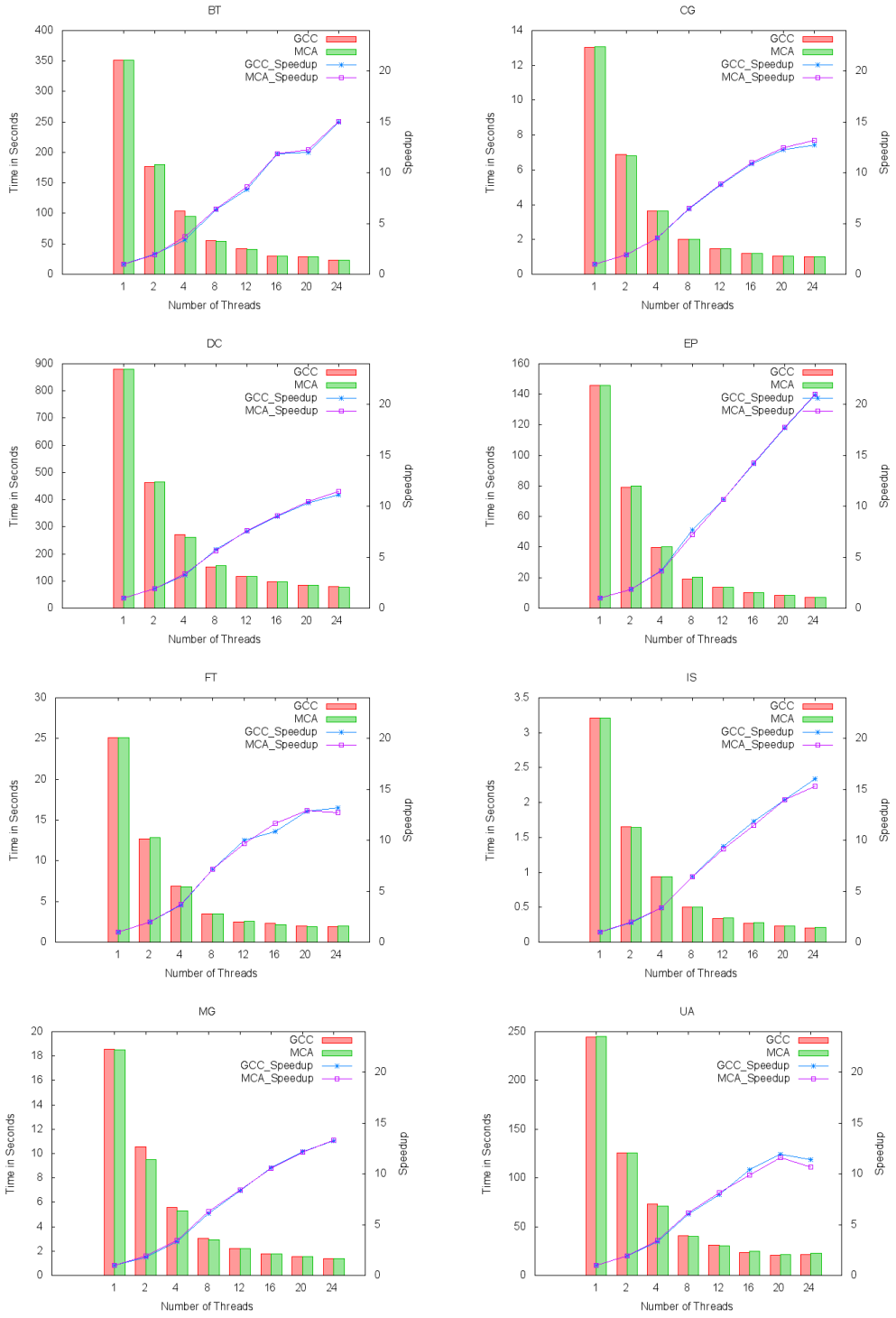

Fig. 4: Evaluating MCA-libGOMP runtime library using NAS benchmarks

[22] Y.-C. Wang, B. Donyanavard, and K.-T. T. Cheng, "Energy-aware real-time face recognition system on mobile cpu-gpu platform," in *Trends and Topics in Computer Vision*, pp. 411–422, Springer, 2012.

[23] K.-T. Cheng and Y.-C. Wang, "Using mobile gpu for general-purpose computing–a case study of face recognition on smartphones," in *VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on*, pp. 1–4, IEEE, 2011.

[24] N. Rajovic, A. Rico, J. Vipond, I. Gelado, N. Puzovic, and A. Ramirez, "Experiences with mobile processors for energy efficient hpc," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 464–468, EDA Consortium, 2013.

[25] P. Rogers and A. C. FELLOW, "Heterogeneous system architecture overview," in *Hot Chips*, 2013.

[26] D. F. Bacon, R. Rabbah, and S. Shukla, "Fpga programming for the masses," *Communications of the ACM*, vol. 56, no. 4, pp. 56–63, 2013.

[27] J. Sérot, F. Berry, and S. Ahmed, "Caph: A language for implementing stream-processing applications on fpgas," in *Embedded Systems Design with FPGAs*, pp. 201–224, Springer, 2013.

[28] A. Kamppi, L. Matilainen, J. Maatta, E. Salminen, T. D. Hamalainen, and M. Hannikainen, "Kactus2: Environment for embedded product development using ip-xact and mcapi," in *Digital System Design (DSD), 2011 14th Euromicro Conference on*, pp. 262–265, IEEE, 2011.

[29] L. Matilainen, E. Salminen, T. Hamalainen, and M. Hannikainen, "Multicore communications api (mcapi) implementation on an fpga multiprocessor," in *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pp. 286–293, IEEE, 2011.

[30] L. Gantel, M. Benkhelifa, F. Verdier, and F. Lemonnier, "Mrapi implementation for heterogeneous reconfigurable systems-on-chip," in *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pp. 239–239, IEEE, 2014.

[31] A. Nicolas, H. Posadas, P. Peñil, and E. Villar, "Automatic deployment of component-based embedded systems from uml/marte models using mcapi," *XXIX Conference on Design of Circuits and Integrated Systems,DCIS 2014.*, 2014.

[32] K. O'Brien, K. O'Brien, Z. Sura, T. Chen, and T. Zhang, "Supporting OpenMP on Cell," *Int. J. Parallel Program.*, vol. 36, pp. 289–311, June 2008.

[33] L. Huang, E. Stotzer, H. Yi, B. Chapman, and S. Chandrasekaran, "Parallelizing ultrasound image processing using openmp on multicore embedded systems," in *Global High Tech Congress on Electronics (GHTCE), 2012 IEEE*, pp. 131–138, IEEE, 2012.

[34] A. Marongiu, P. Burgio, and L. Benini, "Supporting openmp on a multi-cluster embedded mpsoc," *Microprocessors and Microsystems*, vol. 35, no. 8, pp. 668–682, 2011.

[35] J.-Z. He, W.-G. Chen, G.-R. Chen, W.-M. Zheng, Z.-Z. Tang, and H.-D. Ye, "OpenMDSP: Extending Openmp to Program Multi-Core DSPs," *Journal of Computer Science and Technology*, vol. 29, no. 2, pp. 316–331, 2014.

[36] M. Wu, W. Wu, N. Tai, H. Zhao, J. Fan, and N. Yuan, "Research on openmp model of the parallel programming technology for homogeneous multicore dsp," in *Software Engineering and Service Science (ICSESS), 2014 5th IEEE International Conference on*, pp. 921–924, IEEE, 2014.

[37] A. Marongiu and L. Benini, "An openmp compiler for efficient use of distributed scratchpad memory in mpsocs," *Computers, IEEE Transactions on*, vol. 61, no. 2, pp. 222–236, 2012.

[38] C.-T. Yang, T.-C. Chang, H.-Y. Wang, W. C.-C. Chu, and C.-H. Chang, "Performance comparison with openmp parallelization for multi-core systems," in *Parallel and Distributed Processing with Applications (ISPA), 2011 IEEE 9th International Symposium on*, pp. 232–237, IEEE, 2011.

[39] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[40] F. Sainz, S. Mateo, V. Beltran, J. L. Bosque, X. Martorell, and E. Ayguadé, "Extending ompss to support cuda and opencl in c, c++ and fortran applications," *Barcelona Supercomputing Center–Technical University of Catalonia, Computer Architecture Department, Tech. Rep*, 2014.

[41] NVIDIA, "The openacc specification, version 2.0, august 2013," *URL http://www.openacc-standard.org/, year=2013*.

[42] R. Xu, X. Tian, S. Chandrasekaran, Y. Yan, and B. Chapman, "Nas parallel benchmarks on gpgpus using a directive-based programming model," in *Intl. workshop on LCPC 2014*, 2014.

[43] X. Tian, R. Xu, Y. Yan, Z. Yun, S. Chandrasekaran, and B. Chapman, "Compiling a high-level directive-based programming model for gpgpus," in *Languages and Compilers for Parallel Computing*, pp. 105–120, Springer, 2014.

[44] D. B. Kirk and W. H. Wen-mei, *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.

[45] K. Sollins, "The tftp protocol (revision 2)." https://www.ietf.org/rfc/rfc1350.txt, 1992.

[46] S. Shepler, M. Eisler, D. Robinson, B. Callaghan, R. Thurlow, D. Noveck, and C. Beame, "Network file system (nfs) version 4 protocol," *Network*, 2003.

[47] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng, "Openuh: An optimizing, portable openmp compiler," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 18, pp. 2317–2332, 2007.

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

[49] C. Wang, S. Chandrasekaran, and B. Chapman, "An openmp 3.1 validation testsuite," in *OpenMP in a Heterogeneous World*, pp. 237–249, Springer, 2012.

[50] H. Jin, M. Frumkin, and J. Yan, "The openmp implementation of nas parallel benchmarks and its performance," tech. rep., Technical Report NAS-99-011, NASA Ames Research Center, 1999.