Deploying OpenMP on an Embedded Multicore Accelerator

Spiros N. Agathos* Vassilios V. Dimakopoulos Aggelos Mourelis Alexandros Papadogiannakis
Department of Computer Science
University of Ioannina
P.O. Box 1186, Ioannina, Greece, GR-45110
Email: {sagathos,dimako,cs102007,apapadog}@cs.uoi.gr

***Abstract*—Multiprocessor systems-on-chip (MPSoC) are now considered first-class citizens both in the embedded systems and in the high-performance computing arenas, in the form of specialized or general-purpose accelerators. Programming models for such systems is currently a hot research topic, and as a general rule require deep programmer knowledge of the underlying hardware architecture. In this paper we present the implementation of OpenMP, one of the most intuitive and productive programming models, on the STHORM accelerator. This particular platform provides a shared-memory substrate which OpenMP requires. An innovative feature of our design is the deployment of the OpenMP model both at the host and the fabric sides, in a seamless way, which provides the programmer with a simple but effective interface for offloading and executing OpenMP kernels on the MPSoC. The optimized runtime environment provides full OpenMP support despite its small footprint (less than 10KB for a 16-core cluster) and can sustain close-to-ideal speedups in computationally intensive applications. We detail on design issues we faced along with their solutions, given the limited available resources.**

I. Introduction

The current market trend is to launch devices that are small, portable, mobile and autonomous. They feature embedded components which have already adopted multicore processor architectures in order to provide the necessary processing capacity for demanding end-user applications and for the ever increasing need for multitasking, while also relying on limited energy resources. Their hardware usually consists of heterogeneous processing elements (PEs), e.g. a general-purpose CPU and one or more specialized devices such as graphics or digital signal processors. At the same time, accelerators in the form of multiple, small and rather weak processing elements (e.g. GPGPUs) have been employed to accelerate a broad range of applications. Even the high-performance computing (HPC) community has benefited from them both in terms of performance and power savings.

The real challenge in this era of multicore computing proliferation is to provide a programming model that enables extracting satisfactory performance while also keeping programmer productivity at high levels in application development. As multicore hardware enables the implementation of more functionality on the same device, the corresponding software will become more complex. OpenMP [1] is a very intuitive parallel programming model which can help in dealing with the above issues. OpenMP is the standard programming model for shared memory multiprocessors but is currently expanding its applicability beyond HPC, with proposals such as OpenACC [2]. It is a directive-based model whereby simple annotations added to a sequential C or Fortran program are enough to produce decent parallelism without significant effort, even by non-experienced programmers.

It is our belief that OpenMP can form the basis for an attractive programming model for multicore embedded accelerators. However its implementation for such devices is not straightforward because it was designed with homogeneous shared memory systems in mind. In contrast, embedded systems include diverse groups of weak PEs operating in SIMD or MIMD fashion, usually connected with a few powerful heterogeneous cores. In addition, it is highly likely that these systems come with a partitioned address space and private memories, leaving portions of memory management as a programmer responsibility.

In this work we present the design and implementation of an OpenMP infrastructure for the STHORM platform [3]. This is a multiprocessor system on chip (MPSoC), designed to operate as an embedded application accelerator. The platform consists of a set of low-power general-purpose PEs, working in full MIMD mode, interconnected by an asynchronous network-on-chip (NoC). In addition, the architecture allows the inclusion of special-purpose processing cores to fit particular application domains. In our view, this will be a reference architecture for future multicore accelerators as it combines the ability to perform general-purpose computations alongside with specialized hardware, while also offering a scalable interconnection structure that will allow large core counts.

Implementing OpenMP for such a platform is a non-trivial task. Our implementation is based on the OMPi [4] open-source OpenMP compiler for C. We discuss our experiences, the problems we faced and the design decisions we made in order to provide a full OpenMP implementation for this embedded MPSoC.

The the paper is organized as follows. In Section II we give an overview of programming models and tools available for the STHORM platform. We also survey related efforts for other platforms. In section III we give details on the architecture of our target MPSoC. Section IV describes the design and implementation of our OpenMP infrastructure. Some initial

This work has been supported in part by the General Secretariat for Research and Technology and the European Commission (ERDF) through the Artemisia SMECY project (grant 100230).

* S.N. Agathos is supported by the Greek State Scholarships Foundation (IKY).

experiments are reported in Section VI and finally, Section VII concludes this paper.

II. Related Work

A number of programming models and tools have been proposed for the STHORM architecture, which is presented in detail in Section III. Apart from the low-level system software which is rather inappropriate for general application development, STHORM comes with the Native Programming Model (NPM) which relies on the system runtime libraries and forms the base for component-based tools and applications. OpenCL is a major model implemented in STHORM, relying on the system runtime libraries, too. A number of high-level tools have also been ported, targeting the NPM or OpenCL layers. A prominent example is BIP/DOL [5] which generates platform-specific code using abstract application models based on communicating processes. Here we present the first OpenMP implementation for the STHORM accelerator.

We are not the first to propose OpenMP as a suitable model for accelerators or multicore embedded systems. Other efforts include [6] where the authors propose OpenMP extensions to provide a high level API for executing code on FPGAs. They propose a hybrid computation model supported by a bitstream cache in order to hide the FPGA configuration time needed when a bitstream has to be loaded. Sato, Nakajima, Ojima and Hotta [7] implement OpenMP and report its performance on a dual M32R processor, which runs Linux and supports fully the POSIX execution model. Liu and Chaudhary [8] implement an OpenMP compiler for the 3SoC Cradle system, a heterogeneous system with multiple RISC and DSP-like cores. Additionally in [9] double buffering schemes and data prefetching are proposed for this system. In [10] the authors discuss an OpenMP implementation that targets MPSoCs with physically shared memories, hardware semaphores, and no operating system. However, they cannot use the fork/join model for the parallel directive due to the lack of threading primitives. In contrast the above works, we support the OpenMP programming model in both the host and the device sides seamlessly, alleviating, in part, programming heterogeneity issues.

Furthermore, extensions to OpenMP have been proposed to enable additional models of execution for embedded applications. Gonzàlez, Ayguad, Martorell and Labarta [11] extend OpenMP to facilitate expressing streaming applications through the specification of relationships between tasks generated from OpenMP worksharing constructs. The authors in [12] propose a set of OpenMP extensions that can be used to to convert conventional serial programs into streaming applications.

Chapman et al. [13] describe the goals of an OpenMP-based model for different types of MPSoCs that takes into account non-functional characteristics such as deadlines, priorities, power constrains etc. They also present the implementation of the worksharing part of OpenMP on a multicore DSP processor. In [14] the authors present an OpenMP task implementation for a simulated embedded multicore platform inspired by the STHORM architecture. Their system consists of doubly linked queues which store the tasks. They make use of task cut-off techniques and task descriptor recycling. Those works, however, do not address full OpenMP functionality, in contrast to what we present here.

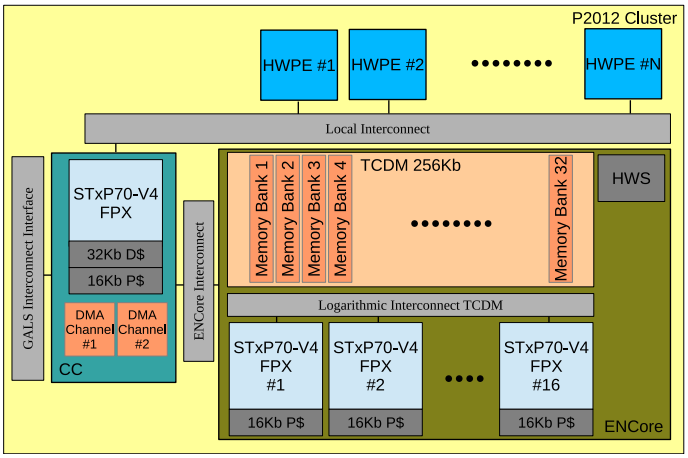

Fig. 1. STHORM cluster architecture

III. System Architecture

The system we target is STHORM [3], formerly known as the ST P2012 platform. The accelerator fabric consists of tiles, called clusters, connected through a globally asynchronous locally synchronous (GALS) network-on-chip where each cluster may operate in different clock speed. The architecture of a single cluster is depicted in Fig. 1 and is composed of a multicore computing engine, called ENCore, and a cluster controller (CC). Each ENCore can host from 1 to 16 processing elements (PEs). Each PE is an STxP70-V4 processor, a cost effective and customizable 32-bit RISC core. It has a 32-bit load/store architecture with variable-length instruction set encoding, allowing manipulation of 32-bit, 16-bit or 8-bit data. This particular version (V4) of the processor includes a floating point unit and corresponding instruction set extensions (FPX). Thus the ENCore PEs are full-fledged processors and follow the MIMD execution model, i.e. each one operates independently of the others. This is an important feature that simplifies the design of embedded applications.

Each PE has a 16KB private instruction cache and no data cache. Instead, the team shares a 256KB scratchpad memory called TCDM (tightly coupled data memory). The latter features single cycle accesses and is divided into 32 banks in order to reduce the probability of conflicts, when several PEs try to access it simultaneously. The cluster is coordinated by the cluster controller (CC) which is also an STxP70-V4 processor. The CC has a 16KB instruction cache, an additional 32KB of local data memory and also two DMA channels for memory copies.

Each ENCore cluster is provided with a Hardware Synchronizer (HWS) engine which provides hardware synchronization support for efficient implementation of semaphores, locks and barriers. It also includes an event and interrupt generator. The accelerator has also provisions for additional Hardware Processing Elements (HWPEs) to efficiently support applications that require specialized hardware (for example audio / video decoders).

Fig. 2 shows the configuration of a STHORM SoC which consists of four clusters. The system includes a 1MB of L2 (or fabric) memory used for instruction and data, also shared among the clusters. In addition there exists a special unit

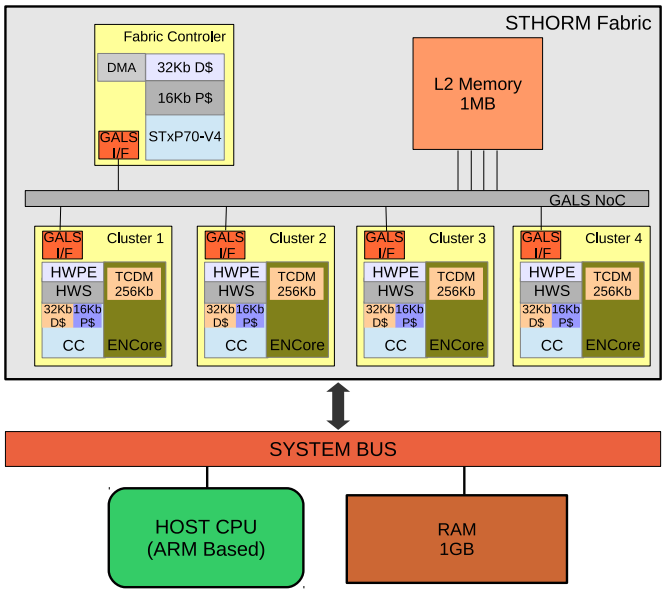

Fig. 2. STHORM SoC Configuration

called fabric controller (FC). The FC is similar to the CC processor and is responsible for interactions with the off-chip host and for instrumentation and coordination of the clusters. The overall memory organization follows the partitioned global address space (PGAS) model. This means that all processing elements can access all SoC memories but with varying delays depending on where the PE and the data are located. For example it is possible for a PE of one cluster to perform load and store instructions directly from/to the fabric memory or a remote TCDM memory of another cluster, albeit slower as compared to accessing its own local memory.

IV. Implementing OpenMP

In order to provide an implementation of the OpenMP model for the STHORM architecture which however is general enough to be ported to similar future MPSoCs, we relied on source-to-source compilation, In particular we based our implementation on the OMPi compiler [4]. OMPi is lightweight OpenMP C infrastructure, composed of a source to source compiler and a flexible runtime system. The compiler takes as input C code with OpenMP pragmas and outputs an intermediate multithreaded code augmented with calls to the runtime system. A native compiler is used to generate the final executable. OMPi is an open source project that adheres to OpenMP V3.0 and targets general purpose SMPs and multicore platforms.

An initial observation is that an accelerator is usually not a stand-alone device—it is rather a back-end system attached to a host, which could also be a multicore processor. There is a conceptual difficulty as to where and how the OpenMP programming model is to be applied: at the host side or at the accelerator side? That is, should OpenMP threads live in the host processor and generate simultaneous (sequential) computational requests towards the fabric, or should the parallelism (OpenMP threads) be created and executed within the accelerator? Our design decision was to be as general as possible, so as to have the greatest flexibility in supporting general MPSoC architectures, and as programmer-friendly as possible, so as to let the programmer express parallelism in any way convenient. Consequently the goal of our compilation chain was to support OpenMP at both the host and the accelerator levels.

OMPi's compilation chain for MPSoC accelerated systems is shown in Fig. 3 and is composed of three phases. During the first phase and after preprocessing the source file, the compiler divides it into two new OpenMP files. The first file contains the code to be executed by the host processor and the second one has the kernel functions to be offloaded and executed on the MPSoC fabric. This target separation phase analyzes the function call graph and packs all dependent functions into the fabric code. During the second phase the actual transformations take place; the separated files are transformed into intermediate pure C code. The intermediate file for the host embeds calls to the standard OMPi runtime designed for the support of shared memory systems, which has been extended to provide the necessary primitives for communication with the accelerator. The intermediate file for the fabric is augmented with calls to a new runtime that supports OpenMP execution within the accelerator. During the final phase, system back-end compilers are used to link the intermediate object files with the appropriate runtime libraries and to create the two executables. The executable for the fabric side is delivered as a shared library.

A. Programming Model

From a programmer's point of view the application consists of two parts to be executed, respectively, by the host and the accelerator, which however are presented in a unified code. The accelerator part is a collection of C functions that are appropriately annotated. Function annotation occurs at the call site. The annotation model we follow is based on the SME-C [15] representation that has been proposed by the SMECY project consortium. In particular, a function call preceded by the following pragma:

#pragma smecy map(HWunit) [arg[[,]arg]...]
<function call>

causes the called function (or kernel in OpenCL terms) to be offloaded to the accelerator and executed (mapped) at a specific hardware unit. Valid hardware units are the PEs, the cluster controller and the fabric controller. Execution by a PE is described by the pair (PE, n) where n is the id of the PE that should execute the offloaded function. The optional 'arg' clauses describe the size and direction of function arguments (input/output/both). In the offloaded function code, OpenMP directives are allowed which dynamically spawn parallelism within the fabric. In addition OpenMP in the rest of the user code is translated as parallelism to be generated at the host. Multiple host threads may offload multiple functions for simultaneous execution on the fabric.

An example is shown in Fig. 4. The execution of this code begins at the host where a team of four threads is created (line 7), and is visualized in figure 5. The thread with id 3 that meets the offload directive, forces the accelerator to execute the kernel function *foo* and suspends its execution until the kernel is finished (lines 10-11). After the accelerator is enabled, the PE with id 0 begins executing the kernel. When this PE

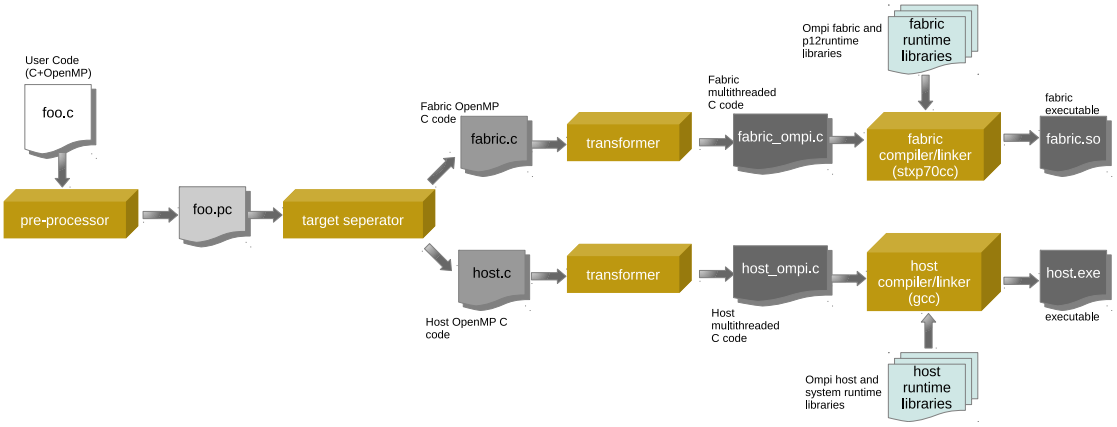

Fig. 3. OMPi compilation chain

```
1  void foo(int A[256]) {
2    #pragma omp parallel
3    {...}
4  }
5
6  int main(void) {
7    #pragma omp parallel num_threads(4)
8    {
9      if (omp_get_thread_num()==3) {
10       #pragma smecy map(PE,0) arg(1,inout,[256])
11       foo(A);
12     }
13     else {
14       ...
15     }
16   }
17 }
```

Fig. 4. Example of kernel (foo) offloading

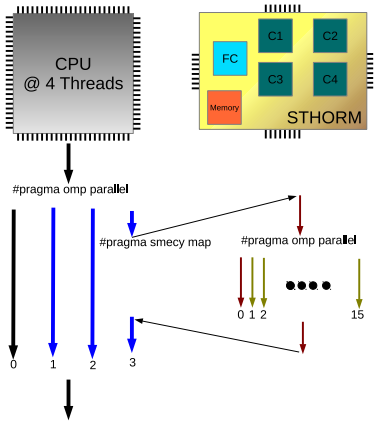

Fig. 5. STHORM execution model

encounters the parallel directive in line 2 it creates a team of 16 PEs to execute the code in line 3. After the kernel's execution, control goes back to the host thread with id 3 in order to resume its work. Notice that the programmer doesn't have to deal with special glue code for the accelerator management (i.e. discover devices, enable/disable units, load binary etc.) because this functionality is provided in the compiler generated code and its runtime library.

B. Data Management

From the programmer's perspective the accelerator memory hierarchy consists of three types of memory:

1) a *scratchpad* memory area (implemented by the TCDM in Fig. 1), which is fast and serves to store data used repeatedly by the PEs. For the STHORM platform its size is 256KB per cluster.
2) a slower *fabric* memory, outside of but shared by all clusters, with a size of 1Mb and
3) a *shared* memory area, accessible by both the host and the accelerator. This is part of system's RAM used for communication between the host and the fabric. Its size can be large, and so is its access time from any PE.

The arguments of the offloaded function are stored in the shared memory area. In the example of Fig. 4 the compiler transparently generates code to i) allocate space in shared memory for a copy of the 256 elements of array A, ii) copy A to the allocated shared memory area iii) execute the kernel iv) copy the data back from shared memory to A (since it is both an input and an output of the function) and v) free the shared memory space.

Because of the need to utilize variables by multiple kernels, instead of transferring them back and forth multiple times, we provide a new directive that allocates such variables on the SoC and stores them there for the whole program execution:

#pragma device_global(var [,var [, ...]])

The enlisted variables will be stored in fabric memory. Finally, in order to allow programs exploit the full memory hierarchy, calls for allocating/freeing memory as well as calls for copying memory areas using the underlying DMA mechanisms are provided:

- ***ompi_local_malloc()***
- ***ompi_local_free()***
- ***ompi_sthorm_dma_ext2loc_memcpy()***
- ***ompi_sthorm_dma_loc2ext_memcpy()***

The first two functions are used to allocate/deallocate space within the scratchpad memory of a cluster, while the other

two are used for DMA transfers between shared memory and scratchpad memory.

V. Runtime Support

Runtime support is provided on top of native libraries which provide basic operations such as feeding jobs to PEs, CCs and FC, memory management, DMA transfer primitives and synchronization facilities.

The two types of processing units, CC and PE have a discrete role in program execution. PEs execute code in a MIMD manner and have limited access to the cluster hardware. PEs can execute reads and writes in memory, request DMA transfers, increment/decrement atomic counters and send/receive signals. In contrast, the CC has full access to all hardware and can additionally allocate/deallocate space in the local, the fabric and the scratchpad memory, allocate/deallocate atomic counters, events, feed itself and PEs with computations, communicate through mailboxes with other CCs and with the FC. Therefore, from a programmer point of view a CC is a master unit that sends/receives requests to/from other CCs, prepares hardware, distributes work to the PEs, and supervises program execution. On the other hand the PEs are 'slave' units that receive job requests and execute them in a preallocated data environment.

The only way a PE can have an active role is by instructing the CC to perform a job. This way, when a PE wants to execute a privileged operation, it prepares a request, sends it to CC and waits until the CC satisfies it. These privileged operations include: allocate/deallocate memory, give jobs to other PEs and allocate/deallocate DMA requests.

A. EECB *Management*

Throughout the execution of an OpenMP application OMPi associates a block of special data called Execution Entity Control Block (EECB) with every OpenMP thread it manages. The EECB contains all the information needed by the runtime in order to schedule the thread, including the size of its team, the thread's ID, its nested parallel level, a pointer to its parent thread EECB (thus a tree of EECBs is formed at runtime) etc. Whenever a thread starts the execution of a parallel region, a new EECB is assigned to it, which is later freed when the team is disbanded.

An important design decision was the placement of EECBs. These structures are constantly accessed during program execution, so their placement in scratchpad memory was the only solution for guaranteed performance. On the other hand, EECBs may occupy significant space in some execution scenarios, for example in applications that perform nested parallelism in great depths or have a certain number of nested parallel teams which are constantly formed and deformed. To avoid the successivePE requests to the CC and possible overflow of scratchpad memory we designed an EECB placement strategy that uses the minimum memory possible in common OpenMP application cases.

Our strategy uses the following scheme: in the scratchpad memory we use a preallocated table of 16 EECBs and a dynamic list that will be used for all active EECBs. We also use a dynamic list placed in the fabric memory that will be used

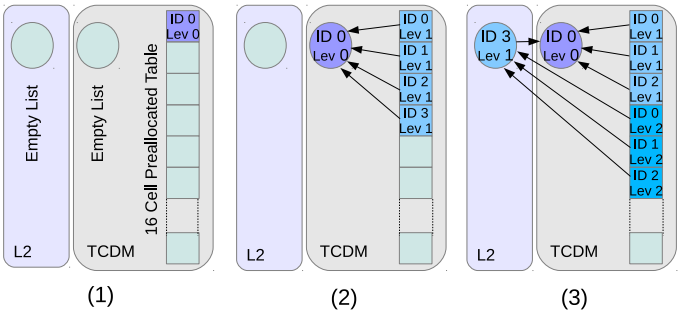

Fig. 6. EECB placement. 1) When a single PE executes a kernel, 2) when this PE forms a team of 4 OpenMP threads and 3) when a PE of this team creates a new nested team of 3 OpenMP threads.

in case of nested parallelism. The preallocated table and the two lists mentioned are also used for recycling EECB data. The core idea is to always maintain the EECBs of active threads in scratchpad memory in every execution scenario. In non-nested parallelism cases the parent data is stored in the scratchpad list. In contrast, during the execution of nested teams, the parent data is placed in the fabric memory list.

This hybrid scheme is shown in Fig. 6. Here we show the EECB placements during the execution of a kernel in three cases: 1) when a single PE executes this kernel, 2) when this PE forms a team of 4 OpenMP threads and 3) when a PE of this team creates a new nested team of 3 OpenMP threads. In case 1), when a PE starts the execution of a kernel it is assigned an EECB from the preallocated table in scratchpad memory. In case 2), this PE suspends the execution of its current job and participates in the new team of threads. To do that it becomes a parent, acquires a new EECB from the list in the scratchpad memory (to use as child), it copies the parent EECB data there and reuses its old EECB. The other threads in the team use EECBs taken from the table. All EECBs point to the parent in scratchpad memory list. Therefore in case of non-nested parallelism all data (parent and child) is placed in scratchpad memory. Finally in case 3), a thread of this team wants to form a nested parallel team. To achieve this, it acquires an EECB from the fabric memory, copies the parent EECB data there and reuses its old EECB. Other threads in the team use EECBs taken from the table in the scratchpad memory. Now the EECBs of child threads point to the EECB in the fabric memory list. The accesses in fabric memory are slower but this solution can efficiently support nested parallelism without wasting memory resources. During the deformation of parallel teams, the reverse procedure is followed and EECBs from the lists are copied back to the EECBs in the table. This way active EECBs are always placed in the scratchpad memory.

B. *Parallel Regions*

Our runtime treats an OpenMP parallel region as a group of implicit tasks that are executed in parallel by PEs. When a PE executing a kernel meets a parallel directive, it suspends the execution of its current job and sends a request to the CC in order to supply other PEs with the appropriate implicit tasks. After that it allocates a new EECB with the procedure mentioned above and becomes the master PE of the newly created OpenMP team, executing its implicit task (job). Simultaneously, other PEs receive the request of the CC and start

executing their jobs. When a PE finishes its job (i.e. exits the parallel region) it notifies the CC and then falls to sleep mode. When all PEs have finished their jobs, the CC informs the master PE so as to safely do its bookkeeping, return to its old EECB and resume its suspended job. We chose to utilize the CC in order to avoid the use of locks and to efficiently exploit CC's idle time.

The PEs need memory to use as stack in order to execute a job. OMPi allocates space for these stacks in the scratch-pad memory and then uses a recycling mechanism to avoid unnecessary allocations and deallocations. The default stack size is 4KB for the master thread (PE 0) and 512 bytes for all other threads, occupying 11.5KB for a team of 16 concurrent threads. Depending on the application, larger stack sizes may be required; the actual amount of stack space is user-controllable by a standard OpenMP environmental variable (`OMP_STACKSIZE`).

In the current implementation, the total number of threads that can co-exist in a cluster is limited to 16. Therefore, we can have one team of up to 16 threads or two concurrent teams of up to 8 threads each etc. These parallel teams can result from the execution of one or multiple offloaded kernels and at different levels of parallelism. Our runtime can support the concurrent execution of up to 16 kernels where each kernel utilizes only one PE. In case all PEs are busy and a PE wants to create a new OpenMP team, then CC will deny its request and the PE will execute the code of the parallel region serially.

C. Tasking Infrastructure

Efficient tasking support for the OpenMP model requires a sophisticated runtime and a rather generous amount of memory. On the other hand it is uncommon for embedded applications to have deeply nested tasking behaviors or create large numbers of nested parallel teams with large memory requirements. Given the limited memory resources of an MPSoC, we designed a lightweight tasking subsystem. The original implementation of tasking in the OMPi infrastructure [16] is targeting general purpose SMPs and multicore systems with abundant resources, and is clearly unsuitable for an MPSoC like the one under consideration here.

In OMPi all task bookkeeping information, i.e. task status, task id, number of children etc, are stored in structures called task nodes. For better utilization of the limited memory we use a preallocated table of task nodes that it is used as a recycling bin. This table is protected by a lock and is located in the scratchpad memory. At task creation the PEs use this table to allocate space for new tasks. After the task execution the corresponding node is recycled. The procedure of allocating and deallocating nodes includes only the altering of a task node's field.

Our tasking subsystem uses two important structures both located in a cluster scratchpad memory. The first one is a shared FIFO queue with a fixed size which is used to store pending tasks of all parallel teams that are executed within a cluster. This queue is protected by a single lock. The second structure is composed of private queues, one for each PE. These queues have a particularly small size and are also protected by a single lock.

We have extended the OpenMP task directive with an extra optional clause so one can request explicitly by what PE and on which cluster a task should be executed:

#pragma omp task on(cluster_id, pe_id)

This extension gives the user the ability to control task placement explicitly, since by default OpenMP tasks may be executed by any thread. This may also prove useful in increasing code locality. In case a PE meets this new task directive it enqueues the new job to the appropriate private queue.

There are three scheduling points in our tasking implementation. The first one is the aforementioned new clause. The other two are the `taskwait` and `barrier` clauses where PEs search for pending tasks in the global and their private queues and execute them. At `taskwait` a PE executes child tasks defined in the context of its current task, while at a `barrier` a PE will execute all tasks generated by its current OpenMP team.

D. Thread Synchronization & Locks

The HWS (hardware synchronizer) of the accelerator provides a small number of atomic counters (ACs) and an even smaller number of events. However, the tasking infrastructure is in great need for fine grain synchronization and a straightforward implementation of locks (using 1 AC per lock) is not feasible, because the atomic counters provided are not enough to cover the needs of typical task-based OpenMP applications. To solve this problem we present a novel locking scheme that provides an unlimited number of locks, using minimal hardware resources.

The basic idea is to use a small fixed number of ACs and map all program locks to them. Locks are implemented by plain integers. Access to a block of those integers is then protected by a block lock implemented by an AC. All locks share a global event. Thus a PE first access the block AC and then set/unset the actual (integer) lock. The lock handling mechanism is shown in Fig. 7.

For locking, a thread first tries to get access to the lock by constantly increasing the value of the AC. When an AC value is 0 a thread can access the lock data, otherwise some other thread is making changes. After getting access it checks the value of the actual integer lock (`locked`). If `locked` has a value of 0, then it locks it by setting its value to 1 and releases the AC by setting its value to 0. In case `locked` is 1, the thread goes to sleep and waits for the event. For unlocking, a thread again gets access through the AC, unlocks the actual lock by setting `locked` to 0, releases the AC and signals the event to wake up any sleeping threads.

An interesting part is the initialization procedure of the lock data. The AC field is initialized through the assign_AC function. This function uses a preallocated table of ACs and returns the next available AC. If all ACs are being used then the same ACs are used again, forming thus blocks of locks protected by the same atomic counter. In order to equalize the load on the block ACs, we allocate the actual integer locks in a round robin manner among the blocks.

OpenMP barriers are also implemented using ACs. An atomic counter is used to count the number of threads that

```
void Initialize(lock l) {
  l.lockAc = assign_AC();
  l.lockEvt = globalEvt;
  l.loked = 0;
}

void Unlock(lock l) {
  /* Grant access to lock */
  old = 0;
  while (old != 1)
   old = increase(l.lockAc);

  /* Unlock, wake up PEs */
  l.locked = 0;
  set_value(l.lockAc, 0);
  raise_evt(lock.lockEvt);
}

void Lock(lock l) {
  while(1) {
    /* Grant access to lock */
    old = 0;
    while (old != 1)
      old = increase(l.lockAc);

    if (l.locked == 0 ) {  /* Try to lock */
      l.locked = 1;
      set_value(l.lockAc, 0);
      break;
    }
    else {                 /* Go to sleep */
      set_value(l.lockAc, 0);
      wait_event(l.lockEvt);
    }
  }
}
```

Fig. 7. Code for lock initialization, locking and unlocking

have reached the barrier. At the barrier all waiting threads keep looking for pending tasks to execute until the last one releases them.

E. Summary

The runtime infrastructure we presented has been highly optimized in order to provide full and efficient OpenMP support with a minimal footprint. The total memory requirements are approximately 9.5KB for a team of 16 threads; this includes everything (i.e. storage for EECBs, task pools, locks, etc) except the thread stacks which by default occupy 11.5KB as discussed above. Consequently, in a typical run our library consumes only about 21KB (≈8%) of the TCDM memory, leaving a large portion of it available for the application.

The current implementation can dedicate only one STHORM cluster per offloaded kernel. This means that all four clusters can be utilized albeit by four different kernels. A single kernel may thus only utilize up to 16 PEs of a single cluster. We currently extend our framework to allow a kernel to exploit more than one cluster.

VI. Preliminary Experimental Results

We have conducted a number of experiments in order to test the efficiency of our OpenMP platform in the context of the STHORM accelerator. An actual implementation of the STHORM platform is not currently available. Consequently, for our experiments we utilized a cycle-accurate simulator provided by ST Microelectronics. The simulator is accompanied by a software development kit. Finally the host processor consists of a dual core ARM Cortex CPU. After the simulation of an application a detailed trace file is produced for the execution steps within the MPSoC. We utilized a provided performance analyzer tool to extract information from these trace files.

TABLE I. Overheads for various operations (16 threads)

Operation	# Cycles
kernel offload	6283
omp parallel	37750
omp for	8427
omp barrier	11941

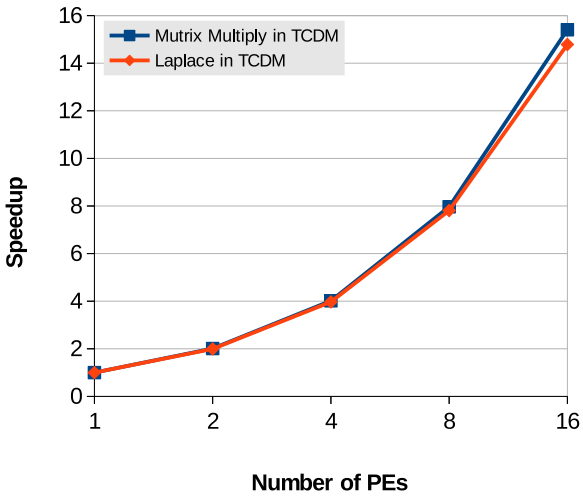

Fig. 8. Performance of matrix multiplication and the Laplace equation solver

First we present the cost for some crucial operations of the runtime system. In particular, in Table I we provide the overheads (in cycles) for offloading a kernel to the fabric and for three important OpenMP constructs: creation of a parallel team, loop worksharing and team barrier. The offloading overhead is measured by repeatedly submitting empty kernels for execution through a `smecy map` directive and counting the average number of cycles. For the last three we considered a team of 16 threads and followed the method of the EPCC benchmarks [17]. The larger overhead of the parallel construct is justified because of the extensive communication required among the PE that encounters the parallel region, the CC and the rest of the PEs.

Next, we present performance results for three applications: matrix multiplication, Laplace equation solver and calculation of the Mandelbrot set. We parallelized these kernels using OpenMP within the offloaded code and executed them in a cluster of the accelerator. The default stack sizes were used (512 bytes per worker and 4KB for the master thread) and the data sets were chosen so as to fit within the scratchpad memory.

In Fig. 8 we plot the speedup curves for the first two applications. Matrix multiplication uses the standard triple loop computation for matrices of 64×64 floats. All three

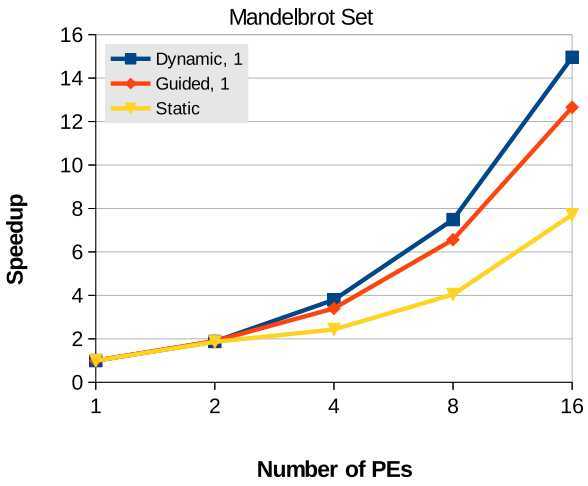

Fig. 9. Speedup for the Mandelbrot set with a variety of loop schedules

matrices (including the result) fit in the scratchpad memory. It is worth mentioning that the only code modification during the parallelization process was the addition of one extra line of a `#pragma omp parallel for` directive. This greatly shows the simplicity and high level parallelization possible by OpenMP. The same figure includes the speedup curve of a simple Laplace equation solver with a 4-point stencil operation and 100 iterations. The parallelization technique utilizes two matrices of 162×162 floats, totaling 205KB which fits with the available TCDM space. In both cases, we observe a close to ideal behavior.

Finally, Fig. 9 shows the performance of of the Mandelbrot set calculation for an image of 362×208 pixels (occupying $\approx$220KB). We plot results for different scheduling policies of the OpenMP `for` construct. Due to the highly unbalanced iteration load, the worst behavior is exhibited by the `static` schedule while the `dynamic` schedule proves to be the best policy, achieving almost linear speedup. The above results demonstrate the efficiency of our runtime infrastructure.

VII. Conclusion

We presented the design and implementation of the OpenMP programming model for the STHORM platform. The architecture of our infrastructure is general enough to allow relatively straightforward porting to other MPSoC accelerators that feature general-purpose processing elements. In addition, to the best of our knowledge, this is the first work that supports OpenMP both at the host and on the fabric side, and in a seamless manner.

The efficiency of an OpenMP runtime depends on how well it manages the limited available resources of the MPSoC. OpenMP runtime libraries targeting general-purpose multiprocessor and multicore machines are largely unsuitable. A simplified design is called for in order to provide only essential capabilities without sacrificing performance while at the same time saving the majority of the resources for the user programs.

We currently work in two directions: first, to provide full fabric (all clusters) support for a single offloaded kernel function and second, to assess and optimize the power requirements of our implementation. Finally, it is in our plans to make our implementation freely accessible when the STHORM platform becomes available in retail.

References

[1] OpenMP ARB, "OpenMP Application Program Interface V3.1," July 2011. [Online]. Available: http://www.openmp.org

[2] OpenACC Home, "OpenACC." [Online]. Available: http://www.openacc.org

[3] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," in *Proc. of DATE'12, Design Automation & Test in Europe*, Dresden, Germany, Mar. 2012, pp. 983–987.

[4] V. V. Dimakopoulos, E. Leontiadis, and G. Tzoumas, "A portable C compiler for OpenMP V.2.0," in *Proc. EWOMP 2003, 5th European Workshop on OpenMP*, Aachen, Germany, Sept. 2003, pp. 5–11.

[5] A. Basu and et al, "System-Level Modeling, Analysis and Code Generation: Object Recognition Case Study," in *Proc. of Embedded World Conference 2012*, Nuremberg, Germany, Feb. 2012.

[6] D. Cabrera, X. Martorell, G. Gaydadjiev, E. Ayguade, and D. Jiménez-González, "OpenMP extensions for FPGA accelerators," in *Proc. of SAMOS'09, 9th international conference on Systems, architectures, modeling and simulation*. Samos, Greece: IEEE Press, Jul. 2009, pp. 17–24.

[7] M. Sato, Y. Nakajima, Y. Ojima, and Y. Hotta, "OpenMP Implementation and Performance on Embedded Renesas M32R Chip Multiprocessor," in *Proc. of 6th European Workshop on OpenMP (EWOMP'04)*, 2004, pp. 37–42.

[8] F. Liu and V. Chaudhary, "A Practical OpenMP Compiler for System on Chips," in *Proc. of WOMPAT, Workshop on OpenMP Applications and Tools*, Toronto, Ontario, CANADA, Jun. 2003, pp. 54–68.

[9] ——, "Extending OpenMP for Heterogeneous Chip Multiprocessors," in *Proc. of ICPP 03, 32nd International Conference on Parallel Processing*, Kaohsiung, Taiwan, Oct. 2003, pp. 161–168.

[10] W.-C. Jeun and S. Ha, "Effective OpenMP Implementation and Translation For Multiprocessor System-On-Chip without Using OS," in *Proc. of ASP-DAC '07, 12th Asia and South Pacific Design Automation Conference*. Yokohama, Japan: IEEE Computer Society, 2007, pp. 44–49.

[11] M. Gonzlez, E. Ayguad, X. Martorell, and J. Labarta, "Exploiting pipelined executions in OpenMP," in *Proc of ICPP'03, 32nd Annual International Conference on Parallel Processing*, Boston, MA USA, Oct. 2003, pp. 153–160.

[12] P. Carpenter, D. Rodenas, X. Martorell, A. Ramirez, and E. Ayguadé, "A streaming machine description and programming model," in *Proc of SAMOS '07, 7th international conference on Embedded computer systems: architectures, modeling, and simulation*. Samos, Greece: Springer-Verlag, Jul. 2007, pp. 107–116.

[13] B. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, and A. Gatherer, "Implementing OpenMP on a high performance embedded multicore MPSoC," in *Proc. of IPDPS '09, IEEE International Symposium on Parallel&Distributed Processing*. Rome, Italy: IEEE Computer Society, May 2009, pp. 1–8.

[14] P. Burgio, G. Tagliavini, A. Marongiu, and L. Benini, "Enabling Fine-Grained OpenMP Tasking on Tightly-Coupled Shared Memory Clusters," in *Proc. of DATE 13, Design Automation and Testing in Europe - DATE (to appear on)*, Grenoble, France, Mar. 2013.

[15] M. Torquati, M. Vanneschi, M. Amini, and G. et. al, "An innovative compilation tool-chain for embedded multi-core architectures," in *Proc. of Embedded World Conference 2012*, Nuremberg, Germany, Feb. 2012.

[16] S. N. Agathos, P. E. Hadjidoukas, and V. V. Dimakopoulos, "Design and Implementation of OpenMP Tasks in the OMPi Compiler." in *Proc. PCI '11, 15th Panhellenic Conference on Informatics*, Kastoria, Greece, Sept. 2011, pp. 265–269.

[17] M. J. Bull, "Measuring Synchronisation and Scheduling Overheads in OpenMP," in *Proc. of 1st EWOMP, European Workshop on OpenMP*, Lund, Sweden, Sept. 1999, pp. 99–105.