MTAPI: Parallel Programming for Embedded Multicore Systems

Urs Gleim

Siemens AG, Corporate Technology
http://www.ct.siemens.com/

urs.gleim@siemens.com

Markus Levy

The Multicore Association
http://www.multicore-association.org/

markus.levy@multicore-association.org

Abstract

Using homogeneous and heterogeneous multicore processors requires the programmer to split a software program into tasks that cores can execute in parallel. Operating systems and runtime libraries for embedded systems lack standardized support for fine-grain parallelism. The coordination of many parallel tasks with today's standard APIs that support threads and synchronization mechanisms generates too much overhead relative to the computation time, requires complex low-level synchronization, and is limited to single operating systems running on single homogeneous multicore processors.

MTAPI™, the *Multicore Task Management API* [1] defined by The Multicore Association®, provides an open and standardized approach for system-wide task management in heterogeneous systems and includes runtime scheduling and mapping of tasks to processor cores. This approach, intended for optimizing multicore throughput, allows the programmer to improve the task scheduling strategy for latency and fairness and supports asymmetric multiprocessing at the hardware and software level.

Furthermore MTAPI is designed for very resource constrained devices. It allows minimal implementations in plain C, which can be run on top of an embedded operating system, or even in a bare metal environment. MTAPI is aligned with previously released specifications by The Multicore Association, MCAPI™ and MRAPI™. Together, these APIs provide a balanced infrastructure to support other multicore services and value-added functions.

This paper shows the results of the MTAPI standardization activities.

1 Introduction

The barriers that slow development of complex multicore applications arise when programmers attempt to split programmatic workloads into parallel tasks that can be executed in parallel on different processor cores. The Multicore Association has created an industry-standard specification for the Multicore Task Management API (MTAPI) that supports the coordination of tasks on embedded parallel systems.

Using homogeneous and/or heterogeneous multicore processors requires the programmer to develop software that splits a software program into tasks that can be executed in parallel on different processor cores. Today's operating systems and runtime libraries for embedded systems provide threads or thread-like mechanisms that are not suited for the fine-grain parallelism required by multicore architectures, typically because the coordination of hundreds or thousands of parallel tasks

[1] Application Programming Interface

generates too much overhead relative to the actual computation time. The current programming model requires complex, low-level synchronization and programming with threads is limited to single operating systems running on single homogeneous multicore processors. In heterogeneous embedded systems, however, a system-wide task management is needed.

The MTAPI specification eliminates these obstacles by providing an API that allows programmers to develop parallel embedded software in a straight-forward manner. Core features of MTAPI are runtime scheduling and mapping of tasks to processor cores. Due to its dynamic behavior, MTAPI is intended for optimizing throughput on multicore-systems. Furthermore, it provides the means for a software developer to adjust the task scheduling strategy for specific requirements; for example, to focus more on latency or fairness policies.

Unlike existing APIs that provide task management functionality (e.g. OpenMP [4], TBB [3], Cilk [1]), MTAPI allows implementations for resource-constrained embedded systems, such as those with a small memory footprint, deterministic behavior, and allow for hardware-specific optimizations. Furthermore, portability is essential for the implementation. Therefore, MTAPI supports different processor architectures and can be implemented on top of different operating systems or as a bare-metal solution. In short, MTAPI supports asymmetric multiprocessing at the hardware and software level.

2 Terms and Definitions

MTAPIs most important definitions are:

- **Job**
  A job is a piece of processing implemented by an action (see below). Several actions can implement the same job based on different hardware resources (for instance a job can be implemented on a DSP and a general purpose core, or a job can be both implemented in hardware and in software). Each job is represented by a job ID that is unique in the system (technically speaking there is the concept of *domains* that allows different namespaces in the system, so the pair <domain, ID> has to be unique).

- **Action**
  An action is the implementation of a job. An action can be realized in software or hardware. If an action is implemented in software, it consists of the implementation of a function with a defined signature. Software actions are registered at the MTAPI runtime system by an MTAPI function with the job ID representing the job implemented. Hardware implementations of actions must be known in the MTAPI runtime system implementation. There is no standardized way of registering hardware actions because hardware implementations are literally highly hardware dependent. Hardware actions are also referred to via job IDs.

- **Task**
  A task is the invocation of an action. When a task is started, it is associated with a job representing one or more actions. A task can be started and it is possible to wait for task completion from other parts of the program. Every task can run exactly once, i.e., the task cannot be started a second time. (In the context of this paper we refer to this generic definition of a task. In other contexts the term “task” has a different meaning. Some real-time operating systems use “task” for operating system threads, for example.)

- **Queue**
  A queue is a software or hardware entity in which tasks are enqueued in a

given order. The queue can ensure in-order execution of tasks. Furthermore queues might implement other scheduling policies that can be configured by setting queue attributes.

3 MTAPI Programming Models

MTAPI supports two programming modes derived from use cases of the working group members (Figure 1): tasks and queues.

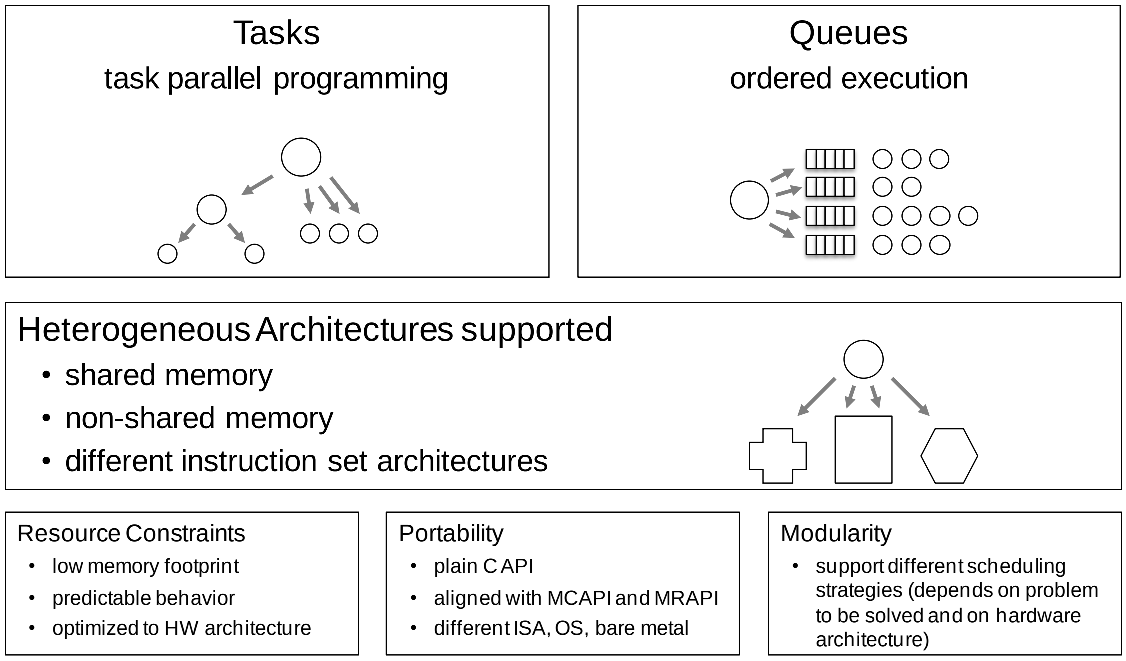
Tasks
task parallel programming
Queues
ordered execution
Heterogeneous Architectures supported
• shared memory
• non-shared memory
• different instruction set architectures
Resource Constraints
• low memory footprint
• predictable behavior
• optimized to HW architecture
Portability
• plain C API
• aligned with MCAPI and MRAPI
• different ISA, OS, bare metal
Modularity
• support different scheduling strategies (depends on problem to be solved and on hardware architecture)

Figure 1: MTAPI programming models

3.1 Tasks

MTAPI allows a programmer to start tasks and to synchronize on task completion. Tasks are executed by the runtime system, concurrently to other tasks that have been started and have not been completed at that point in time. A task can be implemented by software or by hardware. Tasks can be started from remote nodes, i.e. the implementation can be done on one node, but the starting and synchronization of corresponding tasks can be done on other nodes. The developer decides where to deploy a task implementation. On the executing node, the runtime system selects the cores that execute a particular task. This mapping can be influenced by application specific attributes. Tasks can start sub-tasks. MTAPI provides a basic mechanism to pass data to the node that executes a tasks, and back to the calling node.

MTAPI supports the following types of tasks:

- **Single tasks**
  Single tasks are the standard case: After a task is started, the application may wait for completion of the task at a later point in time. In some cases the application waits for completion of a group of tasks. In other cases waiting is not required at all. When a software-implemented task is started, the corresponding code (action) is executed once by the MTAPI runtime

environment. When a hardware-implemented task is started, the task execution is triggered once by the MTAPI runtime system.

- **Multi-instance tasks**
  Multi-instance tasks execute the same action multiple times in parallel (similar to parallel regions in OpenMP or parallel MPI processes).
- **Multiple-implementation tasks / load balancing**
  In heterogeneous systems there could be implementations of the same job for different types processor cores, e.g., one general purpose implementation and a second one for a hardware accelerator. MTAPI allows attaching multiple actions to a job. The runtime system shall decide dynamically during runtime, depending on the system load, which action to utilize. Only one of the alternative actions will be executed.

3.2 Queues

Explicit queues can be used to control the task scheduling policies for related tasks. Order preserving queues ensure that tasks are executed sequentially in queue order with no subsequent task starting until the previous one is complete. MTAPI also supports non-order-preserving queues. This allows to control the scheduling policies of tasks started via the same queue (queues may offer implementation specific scheduling policies controlled by implementation specific queue attributes). Even hardware-queues can be associated with queue objects.

4 MTAPI Reference Architecture

Figure 2 shows an overview of the MTAPI architecture. The application uses the MTAPI interface and optionally also MRAPI and MCAPI interfaces. For some application domains libraries providing high-level APIs are built on top of MTAPI. In this case the application might not use MTAPI directly.

The core of the MTAPI runtime system implementation is a scheduler that controls task execution. Optionally the implementation provides access to hardware-implemented actions and/or queues. MRAPI can be used as part of MTAPI's internal portability layer. The same applies for MCAPI that may be used for inter-node communication internally.

The lower layers provide the mapping to hardware and/or operating system resources (e.g., threads). MTAPI can be implemented on bare metal, on top of an operating system, and also directly on top of a hypervisor (which is not shown in the figure).

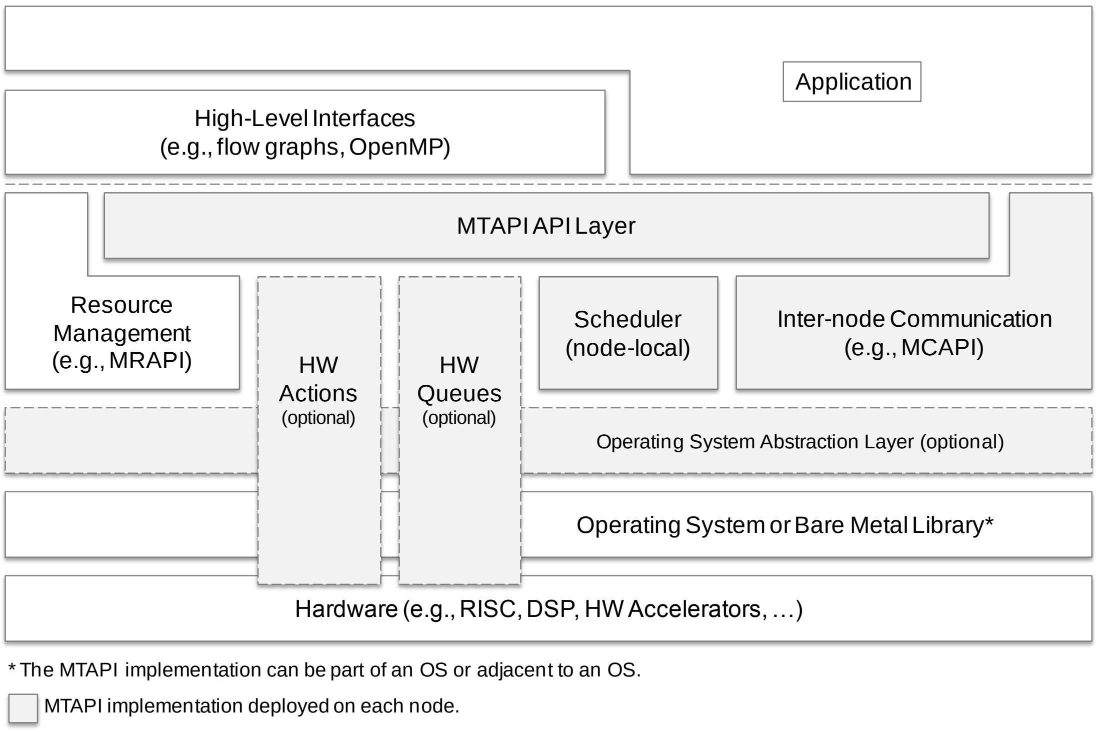
Application
High-Level Interfaces (e.g., flow graphs, OpenMP)
MTAPI API Layer
Resource Management (e.g., MRAPI)
HW Actions (optional)
HW Queues (optional)
Scheduler (node-local)
Inter-node Communication (e.g., MCAPI)
Operating System Abstraction Layer (optional)
Operating System or Bare Metal Library*
Hardware (e.g., RISC, DSP, HW Accelerators, …)
* The MTAPI implementation can be part of an OS or adjacent to an OS.
MTAPI implementation deployed on each node.

Figure 2: MTAPI Reference Architecture

Figure 3 shows the communication of two MTAPI runtime system instances on different nodes of a system. The communication needed for task management is hidden by the MTAPI implementation. It is transparent to the application, if an action is implemented on the node where the corresponding task is started, or if it is executed on a remote node.

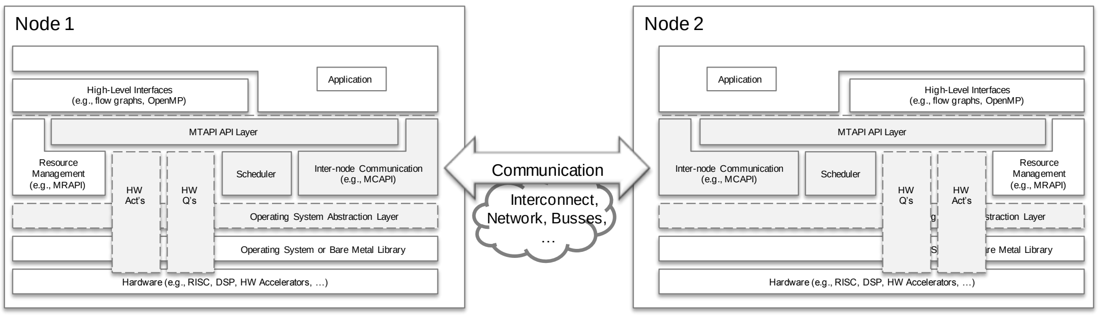
Node 1
Application
High-Level Interfaces (e.g., flow graphs, OpenMP)
MTAPI API Layer
Resource Management (e.g., MRAPI)
HW Act's
HW Q's
Scheduler
Inter-node Communication (e.g., MCAPI)
Operating System Abstraction Layer
Operating System or Bare Metal Library
Hardware (e.g., RISC, DSP, HW Accelerators, …)
Communication
Interconnect, Network, Busses, …
Node 2
Application
High-Level Interfaces (e.g., flow graphs, OpenMP)
MTAPI API Layer
Inter-node Communication (e.g., MCAPI)
Scheduler
HW Q's
HW Act's
Resource Management (e.g., MRAPI)
Hardware (e.g., RISC, DSP, HW Accelerators, …)

Figure 3: MTAPI inter-node communication

5 Tasks

A *task* represents the computation associated with the data to be processed. A task is executed concurrently to the code starting the task. The main API functions are `mtapi_task_start` and `mtapi_task_wait`. The semantics are similar to the corresponding thread functions (e.g. `pthread_create`/`pthread_join` in Pthreads). The lifetime of a task is limited; it can be started only once.

In order to cope with heterogeneous systems and computations implemented in hardware, a task is not directly associated with an entry function as it is done in other task-parallel APIs. Instead, it is associated with at least one *action object* representing the calculation. If the action is implemented in software this is either a function on the same node (which can represent the same processor or core) or a function implemented on a different node that does not share memory with the core starting the task (Figure 4).

Starting a task consists of three steps:

1. create the action object with a job ID (software-implemented actions only)
2. obtain an job reference
3. start the task using the job reference

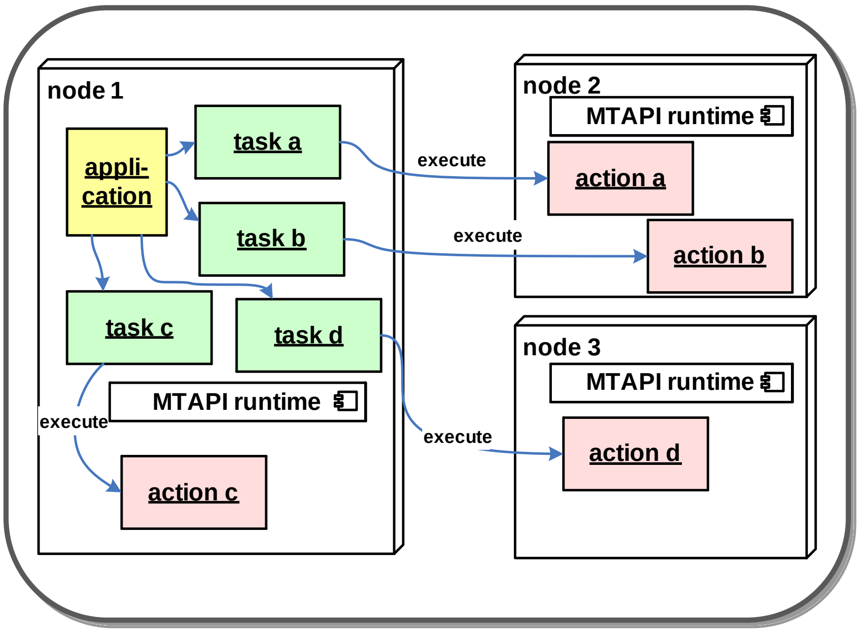

Figure 4: tasks and actions

The MTAPI function for creating an action object is[2]:

```
   mtapi_action_hndl_t mtapi_action_create(
     MTAPI_IN mtapi_job_id_t job_id,
     MTAPI_IN mtapi_action_function_t function,
     MTAPI_IN void* node_local_data,
     MTAPI_IN mtapi_size_t node_local_data_size,
     MTAPI_IN mtapi_action_attributes_t* attributes,
     MTAPI_OUT mtapi_status_t* status
  );
```

The function creates an action, i.e., it registers a function with the runtime environment. It is called on the node where the action function is implemented. Actions are accessible by any node in the system via the job ID. The job ID is unique in the sense that is unique for the job implemented by the action. However several actions may implement the same job for load balancing purposes. For non-default behavior, attributes must be set before the call to `mtapi_action_create()`.

Shared data (`node_local_data`) can be used to set a pointer to node-local data shared by tasks executed on the same node. `node_local_data_size` can be used by the runtime for cache coherency operations.

`mtapi_action_create()` is only called for software-defined actions. Hardware-defined actions and their signature are supposed to be known by the MTAPI runtime implementation.

Before staring a task, the job has to be obtained using the job ID:

```
   mtapi_job_hndl_t mtapi_job_get(
     MTAPI_IN mtapi_job_id_t job_id,
     MTAPI_IN mtapi_domain_t domain_id,
     MTAPI_OUT mtapi_status_t* status
  );
```

The job is uniquely identified by the pair job ID and domain ID. The job ID was assigned by `mtapi_action_create` while domain ID implements the addressing scheme from MCAPI and MRAPI. In MTAPI a node usually represents a processor, a core, or a set of cores (e.g., one node can represent the two ARM Cortex-A15 SMP cores in a TI OMAP 5 processor which contains additional general purpose cores and special purpose accelerators).

[2] At the time of writing this paper, all function signatures in this document are preliminary and subject to change for the final specification.

Once the right job handle is available we can start a task using:

```
mtapi_task_hndl_t mtapi_task_start(
    MTAPI_IN mtapi_task_id_t task_id,
    MTAPI_IN mtapi_job_hndl_t job,
    MTAPI_IN void* arguments,
    MTAPI_IN mtapi_size_t arguments_size,
    MTAPI_OUT void* result_buffer,
    MTAPI_IN mtapi_size_t result_size,
    MTAPI_IN mtapi_task_attributes_t* attributes,
    MTAPI_IN mtapi_group_hndl_t group,
    MTAPI_OUT mtapi_status_t* status
);
```

The task Id may be set for debugging purposes only. If more than one action is associated with a particular job, the MTAPI implementation at this point can decide which action to execute. This, for example, can depend on the system load when starting a task.

Arguments are transferred to the executing node if necessary. Actions and tasks are highly configurable by attributes that specify scheduling parameters such as task to core affinities (`mtapi_action_attributes_t`, `mtapi_task_attributes_t`). The standard attributes defined by the MTAPI specification can be extended by implementation specific attributes.

We can wait for the completion of a dedicated task by calling:

```
void mtapi_task_wait(
  mtapi_task_hndl_t task,          /* task handle */
  MTAPI_IN mtapi_timeout_t timeout,
  MTAPI_OUT mtapi_status_t* mtapi_status
);
```

Task groups (`mtapi_group_hndl` passed at `mtapi_task_start`) can be used to wait for completion of a group of tasks.

It is up to the runtime system how tasks are executed. On SMP systems in most cases a work stealing task scheduler [2] will be used. There may be no preemption between tasks. However, if the MTAPI runtime runs on top an operating system, threads having a higher priority than the threads executing tasks will preempt the tasks.

6 Queues

Queues were made explicit in MTAPI. This allows mapping of queues onto hardware queues, if available. One MTAPI queue is associated with one particular action. In order to support load balancing when using queues, a queue may be associated with several actions implementing the same job on different nodes.

Queues are used to control the scheduling policy of tasks. The default scheduling policy for queues is ordered task execution. Tasks that have to be executed sequentially are enqueued into the same queue. In this case every queue is associated with exactly one action. Tasks started via different queues can be executed in parallel. This is needed for packet processing applications, for example: each stream is processed by one queue. This ensures sequential processing of packets belonging to the same stream. Different streams are processed in parallel.

In some systems queues are implemented in hardware, otherwise MTAPI implements software queues. The MTAPI implementation must be able to handle thousands of queues that are processed in parallel.

The process of setting up a queue and using it is the following:

1. Create at least one action object for a particular job.
2. Obtain a job handle.
3. Create a queue object and attach the job to the queue.
4. Use the queue: enqueue the work using the queue.

For the sake of brevity we will not look at all of the function signatures creating the queue and attaching the job.

Enqueuing a task is similar to `mtapi_task_start`:

```
mtapi_task_hndl_t mtapi_task_enqueue(
      MTAPI_IN mtapi_task_id_t task_id,
      MTAPI_IN mtapi_queue_hndl_t queue,
      MTAPI_IN void* arguments,
      MTAPI_IN mtapi_size_t arguments_size,
      MTAPI_OUT void* result_buffer,
      MTAPI_IN mtapi_size_t result_size,
      MTAPI_IN mtapi_task_attributes_t* attributes,
      MTAPI_IN mtapi_group_hndl_t group,
      MTAPI_OUT mtapi_status_t* status
);
```

7 Discussion and Next Steps

The parts of the API discussed above show the core of MTAPI. Many details of the specification were not covered, including

– standard attributes for action objects, tasks, and queues
– setting attributes
– using task groups
– memory management

– multi-instance tasks (execute the same function several times in parallel)

The primary goals were to provide a small low level API and also support common use cases as good as possible. The task part is essential, the queues are easy to realize since task scheduler internally work with queue anyway. Currently the final specification is released by the consortium after an intensive review phase (working group internally and externally). In parallel some members started a prototype implementation of MTAPI.

The specification also contains source code examples showing the usage of MTAPI.

8 About MCA

The Multicore Association (http://www.multicore-association.org/) provides a neutral forum for vendors who are interested in, working with, and/or proliferating multicore-related products, including processors, infrastructure, devices, software, and applications. The consortium has made available its Multicore Communications API (MCAPI) and Multicore Resource Management API (MRAPI) specifications through its website. Currently, the organization has active working groups focused on: Multicore Virtualization, Multicore Communications (Version 2.x), Multicore Programming Practices (MPP), Multicore Task Management (MTAPI) and Tools Infrastructure (TIWG).

Members include Abo Akademi University, AMD, Argon Design, CAPS entreprise, Carnegie Mellon University, Cavium Networks, Codeplay, CriticalBlue, Delft University of Technology, EADS North America, Ecole Polytechnique de Montreal, EfficiOS, Enea, eSOL, Freescale Semiconductor, IMEC, Intel, LG Electronics Co, LSI , Mentor Graphics, MIPS Technologies, National Instruments, nCore Design LLC, NetLogic Microsystems, Netronome, Nokia Siemens Networks, OneAccess, PolyCore Software, Qualcomm, RadiSys, Sage Electronic Engineering, Samsung Electronics, Siemens, Texas Instruments, Tilera, UAS Technikum Wien, UltraSoC Technologies, University of Houston, University of Tsukuba, and Wind River. Further information is available at www.multicore-association.org.

9 References

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall und Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In: Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 207–216. ACM, 1995.

[2] R. D. Blumofe und C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. In: Symposium on Foundations of Computer Science (FOCS), pages 356–368. IEEE, 1994.

[3] Intel Corporation. Intel Threading Building Blocks: Reference Manual, 2011. Document No. 315415-015US.

[4] OpenMP Architecture Review Board. OpenMP Application Program Interface (version 3.1), 2011.