SOFTWARE STANDARDS FOR THE MULTICORE ERA

SOFTWARE STANDARDS FOR THE MULTICORE ERA

SYSTEMS ARCHITECTS COMMONLY USE MULTIPLE CORES TO IMPROVE SYSTEM PERFORMANCE. UNFORTUNATELY, MULTICORE HARDWARE IS EVOLVING FASTER THAN SOFTWARE TECHNOLOGIES. NEW MULTICORE SOFTWARE STANDARDS ARE NECESSARY IN LIGHT OF THE NEW CHALLENGES AND CAPABILITIES THAT EMBEDDED MULTICORE SYSTEMS PROVIDE. THE NEWLY RELEASED MULTICORE COMMUNICATIONS API STANDARD TARGETS SMALL-FOOTPRINT, HIGHLY EFFICIENT INTERCORE AND INTERCHIP COMMUNICATIONS.

Jim Holt
Freescale Semiconductor

Anant Agarwal
Massachusetts Institute of Technology

Sven Brehmer
PolyCore Software

Max Domeika
Intel

Patrick Griffin
Tilera

Frank Schirrmeister
Synopsys

Pushing single-thread performance is no longer viable for further scaling of microprocessor performance, due to this approach's unacceptable power characteristics.[1-3] Current and future processor chips will comprise tens to thousands of cores and specialized hardware acceleration to achieve performance within power budgets.[4-7] Hardware designers find it relatively easy to design multicore systems. But multicore presents new challenges to programmers, including finding and implementing parallelism, dealing with debugging deadlocks and race conditions (that is, conditions that occur when one or more software tasks attempt to access shared program variables without proper synchronization), and eliminating performance bottlenecks.

It will take time for most programmers and enabling software technologies to catch up. Concurrent analysis, programming, debugging, and optimization differ significantly from their sequential counterparts. In addition, heterogeneous multicore programming is impractical using standards defined for symmetric multiprocessing (SMP) system contexts or for networked collections of computers (see the "Requirements for Successful Heterogeneous Programming" sidebar.

We define a *system context* as the set of external entities that may interact with an application, including hardware devices, middleware, libraries, and operating systems. A *multicore SMP context* would be a system context in which the application interacts with multicore hardware running an SMP operating system and its associated libraries. In SMP multicore-system contexts, programmers can use existing standards such as Posix threads (Pthreads) or OpenMP for their needs.[8,9] In other contexts, the programmer might be able to use existing standards for distributed or parallel computing such as sockets, CORBA, or message passing interface (MPI, see http://www.mpi-forum.org).[10] However, two factors make these standards unsuitable in many contexts:

- heterogeneity of hardware and software, and
- constraints on code size and execution time overhead.

Published by the IEEE Computer Society

Requirements for Successful Heterogeneous Programming

Successful heterogeneous programming requires adequately addressing the following three areas: a development process to exploit parallelism, a generalized multicore programming model, and debugging and optimization paradigms that support this model.

Development process

Multicore development models for scientific computing have existed for years. But applications outside this domain leave programmers with standards and development methods that don't necessarily apply. Furthermore, programmers have little guidance as to which standards and methods provide some benefit and which don't. Domeika has detailed some guidance on development techniques,[1] but there is no industry-wide perspective covering multiple embedded-system architectures. This problem invites an effort to formulate industry standard development processes for embedded systems that address current and future development models.

Programming models

A programming model comprises a set of software technologies that help programmers express algorithms and map applications to underlying computing systems. Sequential programming models require programmers to specify application functionality via serial-programming steps that occur in strict order with no notion of concurrency. Universities have taught this programming model for decades. Outside of a few specialized areas, such as distributed systems, scientific applications, and signal-processing applications, the sequential-programming model permeates the design and implementation of software.[2]

But there is no widely accepted multicore programming model outside those defined for symmetric shared-memory architectures exhibited by workstations and PCs[3] or those designed for specific embedded-application domains such as media processing (see http://www.khronos.org). Although these programming models work well for certain kinds of applications, many multicore applications can't take advantage of them. Standards that require system uniformity aren't always suitable, because multicore systems display a wide variety of nonuniform architectures, including combinations of general-purpose cores, digital-signal processors, and hardware accelerators. Also, domain-specific standards don't cover the breadth of application domains that multicore encompasses.

The lack of a flexible, general-purpose multicore-programming model limits the ability of programmers to transition from sequential to multicore programming. It forces companies to create custom software for their chips. It prevents tool chain providers from maximally leveraging their engineering, forcing them to repeatedly produce custom solutions. It requires users to craft custom infrastructure to support their programming needs. Furthermore, time-to-market pressures often force multicore solutions to target narrow vertical markets, thereby limiting the viable market to fewer application domains and preventing efficient reuse of software.

Debugging and optimization

There are standards for the ''plumbing'' required in hardware to make multicore debugging work,[4] and debugging and optimization tools exist for multicore workstations and PCs.[5,6] However, as with the multicore-programming model, no high-level multicore debugging standard exists. Tool chain providers have created multicore debugging and optimization software tools, but they are custom-tailored to each specific chip.[7,8] This situation leaves many programmers lacking tool support.

It became evident decades ago that source-level debugging tied to the programming language provided efficient debugging techniques to the programmer, such as visually examining complex data structures and stepping through stack back-traces.[9] Raising the abstraction of multicore debugging is another natural evolutionary step. In a multicore-system context, the programmer creates a set of software tasks, which are then allocated to the various processor cores in the system. The programmer would benefit from monitoring task life cycles and from seeing how tasks interact via communications and resource sharing. However, the lack of standards means that debugging and optimization for most multicore chips is an art, not a science. This is especially troubling because concurrent programming introduces new functional and performance pitfalls, including deadlocks, race conditions, false sharing, and unbalanced task schedules.[10,11] Although it isn't this article's main focus, we believe that the standards work presented in the main text will enable new capabilities for multicore debug and optimization.

References

1. M. Domeika, *Software Development for Embedded Multi-core Systems: A Practical Guide Using Embedded Intel Architecture,* Newnes, 2008.
2. W.-M. Hwu, K. Keutzer, and T.G. Mattson, ''The Concurrency Challenge,'' *IEEE Design & Test,* vol. 25, no. 4, 2008, pp. 312-320.
3. *IEEE Std. 1003.1, The Open Group Base Specifications Issue 6,* IEEE and Open Group, 2004; http://www.opengroup.org/onlinepubs/009695399.
4. B. Vermeulen et al., ''Overview of Debug Standardization Activities,'' *IEEE Design & Test,* vol. 25, no. 3, 2008, pp. 258-267.
5. L. Adhianto and B. Chapman, ''Performance Modeling of Communications and Computation in Hybrid MPI and OpenMP Applications,'' *Proc. 12th Int'l Conf. Parallel and Distributed Systems* (ICPADS 06), IEEE CS Press, 2006, pp. 3-8.
6. J. Cownie, and W. Gropp, ''A Standard Interface for Debugger Access to Message Queue Information in MPI,'' *Recent Advances in Parallel Virtual Machine and Message Passing Interface,* J. Dongarra, E. Lugue, and T. Margalef, eds., 1999, Springer, pp. 51-58.
7. M. Biberstein et al., ''Trace-Based Performance Analysis on Cell BE,'' *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software* (ISPASS 08), IEEE Press, 2008, pp. 213-222.
8. I.-H. Chung et al., ''A Study of MPI Performance Analysis Tools on Blue Gene/L, *Proc. 20th Int'l Parallel and Distributed Processing Symp.* (IPDPS 06), IEEE CS Press, 2006.
9. S. Rojansky, ''DDD—Data Display Debugger,'' *Linux J.,* 1 Oct. 1997; http://www.linuxjournal.com/article/2315.
10. E.A. Lee, ''The Problem with Threads,'' *Computer,* vol. 39, no. 5, 2006, pp. 33-42.
11. *The OpenMP API Specification for Parallel Programming,* OpenMP Architecture Review Board, 2008; http://openmp.org/wp.

For heterogeneous multicore contexts, using any standard that makes an implicit assumption about underlying homogeneity is impractical. For example, Pthreads is insufficient for interactions with offload engines, with cores that don't share a memory domain, or with cores that aren't running a single SMP operating system instance. Furthermore, implementers of other standards such as OpenMP often use Pthreads as an underlying API. Until more suitable and widely applicable multicore APIs become available, these layered standards will also have limited applicability.

Systems with heterogeneous cores, instruction set architecture (ISAs), and memory architectures have programming characteristics similar to those of distributed or scientific computing. Various standards exist for this system context, including sockets, CORBA, and MPI. However, there are fundamental differences between interconnected computer systems (common in distributed and scientific computing) and multicore computers. This limits these standards' scalability into the embedded world. At issue is the overhead required to support features that aren't required in the multicore system context. For example, sockets are designed to support lossy packet transmission, which is unnecessary with reliable interconnects; CORBA requires data marshalling, which might not be optimal between any particular set of communicators; and MPI defines a process/group model, which isn't always appropriate for heterogeneous systems.

These concerns justify a set of complementary multicore standards that will

- embrace both homogeneous and heterogeneous multicore hardware and software,
- provide a widely applicable API suitable for both application-level programming and the layering of higher-level tools,
- allow implementations to scale efficiently within embedded-system contexts, and
- not preclude using other standards within a multicore system.

The companies working together in the Multicore Association (MCA, http://www.multicore-association.org) have published a roadmap for producing a generalized multicore-programming model. The MCA deemed intercore communications a top priority for this roadmap, because it is a fundamental capability for multicore programming. In March 2008, the consortium's working group completed the multicore communications API (MCAPI) specification,[11] the first in a set of complementary MCA standards addressing various aspects of using and managing multicore systems. MCAPI enables and simplifies multicore adoption and use. The example implementation in this article shows that MCAPI can meet its stated goals and requirements, and there is already a commercial version available.

MCAPI requirements and goals

The MCAPI working group's task was to specify a software API for intertask communication within multicore chips and between multiple processors on a board. The primary goals were source code portability and the ability to scale communication performance and memory footprint with respect to the increasing number of cores in future generations of multicore chips. While respecting these goals, the MCAPI working group also worked to design a specification satisfying a broad set of multicore-computing requirements, including:

- suitability for both homogenous and heterogeneous multicore;
- suitability for uniform, nonuniform, or distributed-memory architectures;
- compatibility with dedicated hardware acceleration;
- compatibility with SMP and non-SMP operating systems;
- compatibility with existing communications standards; and
- support for both blocking and nonblocking communications.

The working group first identified a set of use cases from the embedded domain to capture the desired semantics and application constraints. These use cases included automotive, multimedia, and networking domains. For due diligence, the working group applied these use cases to reviews of MPI, Berkeley

sockets, and Transparent Interprocess Protocol Communication (TIPC).[12] Although there are some overlaps, the working group found insufficient overlap between the MCAPI goals and requirements and these existing standards.

MPI is a message-passing API developed for scientific computing that can be useful for closely to widely distributed computers. It is powerful and supports many of the MCAPI requirements, but in its full form it is complex with a large memory footprint. Furthermore, the group communication concept of MPI isn't always amenable to the types of application architectures we found in our use cases.

Berkeley sockets are ubiquitous and well understood by many programmers. But they have a large footprint in terms of both memory and execution cycles. These complexities help sockets to handle unreliable transmission and support dynamic topologies, but in a closed system with a reliable and consistent interconnect such complexities are unnecessary.

TIPC serves as a scaled-down version of sockets for the embedded-telecommunications market. TIPC has a smaller footprint than sockets and MPI. But it is larger than the target for MCAPI: the TIPC 1.76 source code comprises approximately 21,000 lines of C code, whereas the MCAPI goal was one quarter of this size for an optimized implementation.

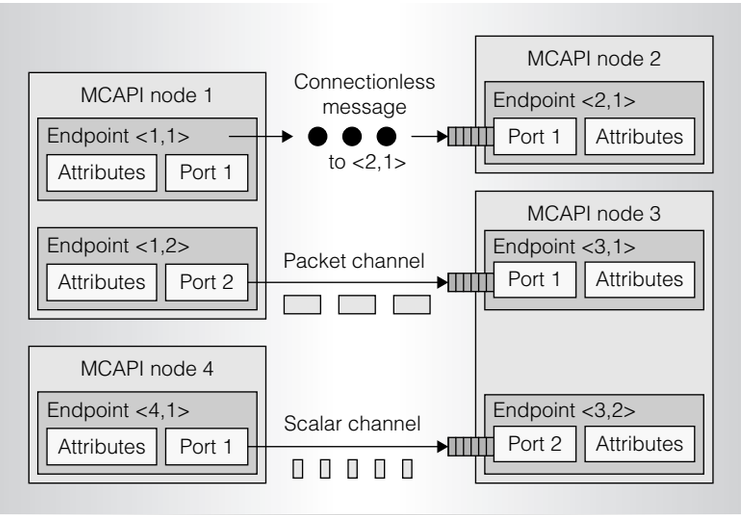

Figure 1. The three multicore communications API (MCAPI) communications types: connection-less messages, and connection-oriented packets and scalars.

MCAPI overview

To support the various embedded-application needs outlined in our use cases, the MCAPI specification defines three communication types (Figure 1), as follows:

- *messages* (connection-less datagrams);
- *packets* (connection-oriented, arbitrary size, unidirectional, FIFO streams); and
- *scalars* (connection-oriented, fixed size, unidirectional, FIFO streams).

Messages support flexible payloads and dynamically changing receivers and priorities, incurring only a slight performance penalty in return for these features. Packets also support flexible payloads, but they use connected channels to provide higher performance at the expense of slightly more setup code. Scalars promise even higher performance, exploiting connected channels and a set of fixed payload sizes. For programming flexibility and performance opportunities, MCAPI messages and packets also support nonblocking sends and receives to allow overlapping of communications and computations.

Communication in MCAPI occurs between *nodes*, which can be mapped to many entities, including a process, a thread, an operating system instance, a hardware accelerator, and a processor core. A given MCAPI implementation will specify what defines a node. MCAPI nodes communicate via socket-like communication termination points called *end points*, which a topology-global unique identifier (a tuple of `<node, port>`) identifies. An MCAPI node can have multiple end points. MCAPI channels provide point-to-point FIFO connections between a pair of end points.

Additional features include the ability to test or wait for completion of a nonblocking communications operation; the ability to cancel in-progress operations; and support for buffer management, priorities, and back-pressure mechanisms to help manage data between producers and consumers.

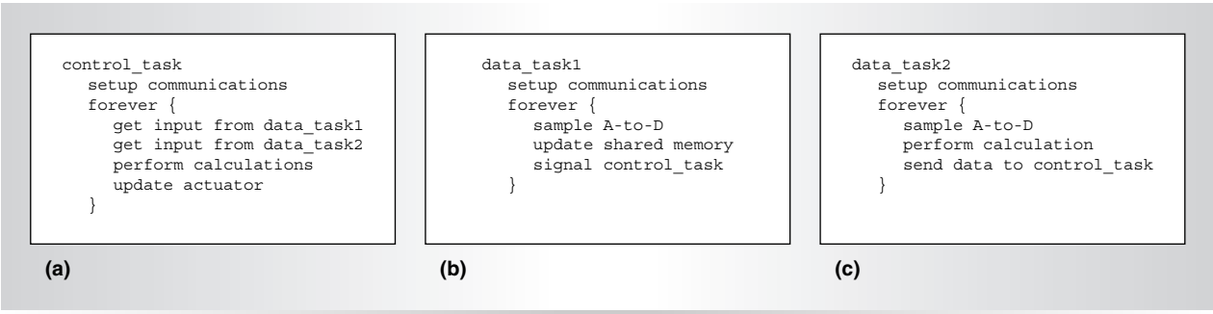

Figure 2. Pseudocode for industrial-automation example. The three tasks involved in this implementation are a control task that receives data from two data tasks, analyzes the data, and adjusts the actuator (a); a data task that collects data from the first sensor (b); and a data task that collects data from the second sensor (c).

Industrial-Automation Application

To illustrate the use of MCAPI, we present a simple example of an industrial-automation application that must read two sensors on a continuous basis, perform some calculation using data from the sensors, and then update a physical device's setting using an actuator. A multicore implementation for this might comprise three tasks: data collection for the first sensor; data collection for the second sensor; and a control task to receive data from the data tasks, analyze the data, and then adjust the actuator. Figure 2 shows pseudocode for the three tasks. This example can highlight the three MCAPI communication modes.

Figure 3 shows an MCAPI implementation of `control_task`. (For brevity, we removed variable declarations and error checking.) In step 1, `control_task` initializes the MCAPI library, indicating its node number via an application-defined constant that is well-known to all tasks. In step 2, `control_task` creates local end points for communications with `data_task1` and `data_task2`. In step 3, `control_task` uses nonblocking API calls to retrieve remote end points created by `data_task1` and `data_task2`, eliminating possible deadlock conditions during the bootstrapping phase caused by a nondeterministic execution order of the tasks. The nonblocking calls (indicated by the `_i` suffix in the function name) return an `mcapi_request_t` handle, which step 4 uses to await completion of the connections. In step 5, `control_task` allocates shared memory and sends the address via an MCAPI message to `data_task1`. In step 6, `control_task` connects end points to create channels to the data tasks, again avoiding potential deadlock by using nonblocking calls. In step 7, `control_task` waits for the connection operations to complete. MCAPI requires two steps to create channels: first, creating the actual channels, which any node can do; second, opening the channels, which the nodes owning each end point must do. Steps 8 and 9 illustrate channel creation for end points owned by `control_task`. In step 10, `control_task` waits for `data_task1` to signal that it has updated the shared memory, and in step 11 `control_task` reads the shared memory. This rendezvous approach provides lock-free access to the critical section for both tasks. In step 12, `control_task` receives a data structure from `data_task2`. In step 13, `control_task` computes a new value and adjusts the actuator. Finally, in step 14, `control_task` releases the buffer that MCAPI supplied for the packet that was received.

Figure 4 shows the MCAPI implementation for `data_task1`. Step 1 shows the MCAPI initialization, creation of a local end point, and lookup of the remote `control_task` end point using nonblocking semantics. In step 2, `data_task1` receives the pointer to the shared-memory region from `control_task`. Step 3 opens the channel between `data_task1` and `control_task`. Now, the

```
void control_task(void) {

   // (1) init the system
   mcapi_initialize(CNTRL_NODE, &err);

   // (2) create two local endpoints
   t1_endpt = mcapi_create_endpoint(CNTRL_PORT_T1, &err);
   t2_endpt = mcapi_create_endpoint(CNTRL_PORT_T2, &err);

   // (3) get two remote endpoints
   mcapi_get_endpoint_i(T1_NODE, T1_PORT_CNTRL, &t1_remote_endpt, &r1, &err);
   mcapi_get_endpoint(T2_NODE, T2_PORT_CNTRL, &t2_remote_endpt, &r2, &err);

   // (4) wait on the endpoints
   while (!((mcapi_test(&r1,NULL,&err)) && (mcapi_test(&r2,NULL,&err))) { }

   // (5) allocate shared memory and send the ptr to data_task1
   sMem = shmemget(32);
   tmp_endpt = mcapi_create_endpoint(MCAPI_PORT ANY, &err);
   mcapi_msg_send(tmp_endpt, t1_remote_endpt, sMem, sizeof(sMem), &err);

   // (6) connect the channels
   mcapi_connect_sclchan_i(t1_endpt, t1_remote_endpt, &r1, &err);
   mcapi_connect_pktchan_i(t2_endpt, t2_remote_endpt, &r2, &err);

   // (7) wait on the connections
   while (!((mcapi_test(&r1,NULL,&err)) && (mcapi_test(&r2,NULL,&err))) { }

   // (8) now open the channels
   mcapi_open_sclchan_recv_i(&t1_chan, t1_endpt, &r1, &err);
   mcapi_open_pktchan_recv_i(&t2_chan, t2_endpt, &r2, &err);

   // (9) wait on the opens
   while (!((mcapi_test(&r1,NULL,&err)) && (mcapi_test(&r2,NULL,&err))) { }

   // begin the processing phase below
   while (1) {
       // (10) wait for data_task1 to indicate it has updated shared memory
       t1Flag = mcapi_sclchan_recv_uint8(t1_chan, &err);

       // (11) read the shared memory
       t1Dat = sPtr[0];

       // (12) now get data from data_task2
       mcapi_pktchan_recv(t2_chan, (void **) &sDat, &tSize, &err);

       // (13) perform computations and control functions …

       // (14) free the packet channel buffer
       mcapi_pktchan_free(sDat, &err);
   }
}
```

Figure 3. MCAPI implementation of `control_task`. The work in this task is divided into an initialization phase (steps 1–9) and a control loop (steps 10–14).

bootstrap phase is complete, and `data_task1` enters its main loop: reading the sensor (step 4); updating the shared-memory region (step 5); and sending a flag using an MCAPI scalar call to `control_task`, indicating the shared memory has been updated (step 6).

Figure 5 shows the MCAPI implementation of `data_task2`. Steps 1 through 4 are essentially the same as for `data_task1`, except that `data_task2` communicates with `control_task` using an MCAPI packet channel. The configuration of the packet and scalar channels is virtually identical. In step 5, `data_task2` populates the data structure to be sent to `control_task`, and in step 6 `data_task2` sends the data.

Figure 6 shows two candidate multicore systems for this application: one comprising two cores of equal size, and one comprising three

```
void data_task1() {

   // (1) init the system
   mcapi_initialize(T1_NODE, &err);
   cntrl_endpt = mcapi_create_endpoint(T1_PORT_CNTRL, &err);
   mcapi_get_endpoint_i(CNTRL_NODE, CNTRL_PORT_T1, &cntrl_remote_endpt, &r1, &err);
   mcapi_wait(&r1,NULL,&err);

   // (2) now get the shared mem ptr
   mcapi_msg_recv(cntrl_endpt, &sMem, sizeof(sMem), &msgSize, &err);

   // (3) open the channel; NOTE – connection handled by control_task
   mcapi_open_sclchan_send_i(&cntrl_chan, cntrl_endpt, &r1, &err);
   mcapi_wait(&r1,NULL,&err);

   // all bootstrapping is finished, begin processing
   while (1) {
       // (4) read the sensor
       // (5) update shared mem with value from sensor
       sMem[0] = sensor_data;
       // (6) send a flag to control_task indicating sMem has been updated
       mcapi_sclchan_send_uint8(cntrl_chan, (uint8_t) 1, &err);
   }
}
```

Figure 4. MCAPI implementation of `data_task1`. This task first performs initialization (steps 1–3), and then continuously reads a sensor, updates a data structure in shared memory, and notifies the control task using an MCAPI scalar channel (steps 4–6).

```
void data_task2() {

   // (1) init the system
   mcapi_initialize(T2_NODE, &err);
   cntrl_endpt = mcapi_create_endpoint(T2_PORT_CNTRL, &err);
   mcapi_get_endpoint_i(CNTRL_NODE, CNTRL_PORT_T2, &cntrl_remote_endpt, &r1, &err);

   // (2) wait on the remote endpoint
   mcapi_wait(&r1,NULL,&err);

   // (3) open the channel; NOTE - connection handled by control_task
   mcapi_open_pktchan_send_i(&cntrl_chan, cntrl_endpt, &r1, &err);
   mcapi_wait(&r1,NULL,&err);

   // all boostrap is finished, now begin processing
   while (1) {
       // (4) read sensor
       // (5) prepare data for control_task
       struct T2_DATA sDat;  sDat.data1 = 0xfea0; sDat.data2 = 0;
       // (6) send the data to the control_task
       mcapi_pktchan_send(cntrl_port, &sDat, sizeof(sDat), &err);
   }
}
```

Figure 5. MCAPI implementation of `data_task2`. After initialization (steps 1–3) this task continuously reads a sensor and then sends updated data to the control task using an MCAPI packet channel.

cores of various sizes. The MCAPI implementation can be easily ported to either of these candidate systems. The main concern would be ensuring that code for tasks is compiled for and run on the correct processors for each of these hardware configurations.

Benchmarking MCAPI

To evaluate how well an implementation of the specification could meet the MCAPI goals of a small footprint and high performance, we created an example implementation of MCAPI and a set of benchmarks.

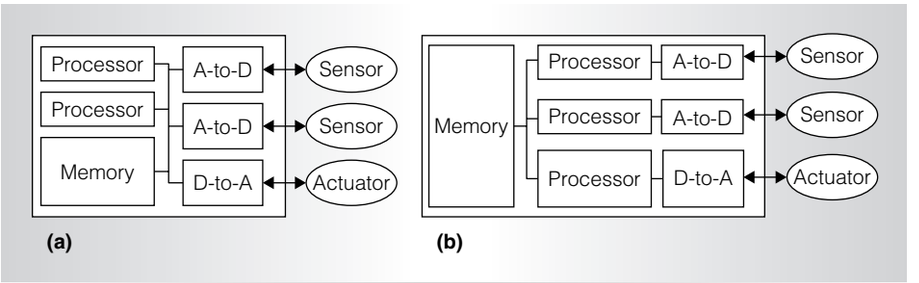

Figure 6. Candidate multicore architectures for industrial-automation example. The first comprises two cores of equal size (a); the other has cores of various sizes (b).

(The implementation and benchmarks are also available on the Multicore Association Web site.)

MCAPI example implementation

We created the example MCAPI implementation to

- determine any MCAPI specification issues that would hinder implementers,
- understand an implementation's code footprint,
- provide a concrete basis for programmers to evaluate their level of interest in MCAPI, and
- establish a baseline for benchmarking optimized implementations.

The example implementation (Figure 7) is publicly available and can be compiled on computers running Linux, or Windows with Cygwin (http://www.cygwin.com) installed. The example implementation's architecture is layered with a high-level MCAPI common-code layer that implements MCAPI interfaces and policies, and a low-level transport layer with a separate API. This architecture allows for reimplementing the transport layer, as desired. The current transport layer uses Posix shared memory and semaphores to create a highly portable implementation.

MCAPI benchmarks

The example implementation shows that it's possible to implement MCAPI with a small footprint. To characterize performance, we also created two synthetic benchmarks (*echo* and *fury*) and characterized the example implementation on several multicore devices. The echo benchmark measures the roundtrip latency of communications through a variable number of hops, with each one being an MCAPI node. A master node initiates each send transaction, which is forwarded in a ring over the desired number of intermediate hops until it returns to the master. When the master node receives the sent data, it reports each transaction's roundtrip latency. The fury benchmark measures throughput between a producer node and a consumer node, sending data as quickly as possible. The consumer node reports the total time and the number of receive transactions along with the amount of data received.

Summary benchmark results

For these results, we didn't tune the example MCAPI implementation for any platform, and we chose a fixed set of implementation parameters (number of nodes, end points, channels, buffer sizes, internal queue depths, and so on).

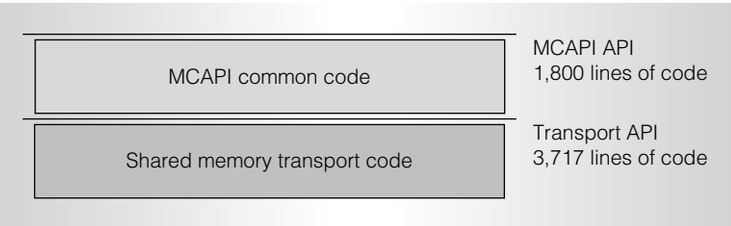

Figure 7. Architecture and code size of example implementation. The MCAPI common-code layer implements MCAPI interfaces and policies. The transport layer has a separate API and can be reimplemented as desired.

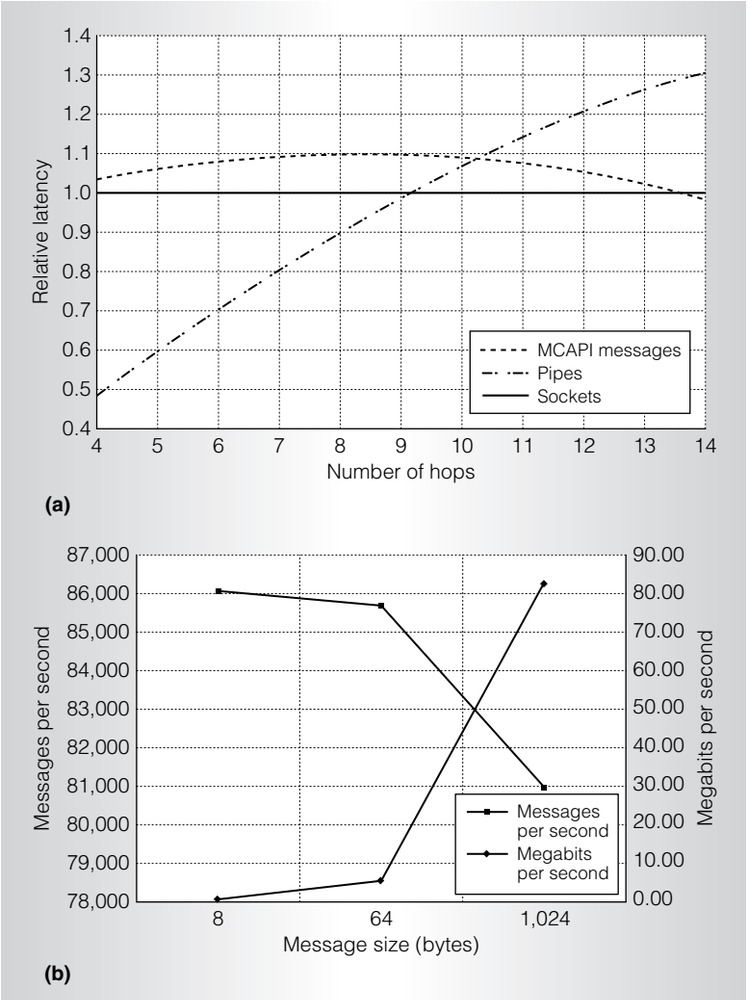

Figure 8. Benchmark results: latencies for the echo benchmark (a) and throughput for the fury benchmark (b).

Figure 8a illustrates the echo benchmark for 64-byte data. We collected this data from a dual-core Freescale evaluation system running SMP Linux. The results are normalized to the performance of a sockets-based version of echo. The other data series in the graph compares the sockets baseline to a Unix pipes version of echo and an MCAPI message-based version of echo. For distances of eight or fewer hops, MCAPI outperformed both pipes and sockets. It is important to emphasize that these results were collected on a dual-core processor with a user-level implementation of MCAPI. Thus, sockets and pipes had kernel support for task preemption on blocking calls, whereas the example MCAPI implementation used polling. Despite these disadvantages, the example MCAPI implementation performed quite well, and we expect optimized versions to exhibit better performance characteristics.

Figure 8b shows fury throughput measured on a high-end quad-core Unix workstation running SMP Linux. Because fury involves only two MCAPI nodes, only two of the four CPUs were necessary for this data. We used scheduler affinity to associate each node with a different CPU. This graph shows that the throughput in messages per second (around 86,000 for 8-byte messages) didn't achieve the highest throughput in bits per second (for example, fury is limited by API overhead for 8-byte messages). But for larger message sizes, the throughput in bits per second was almost two orders of magnitude greater, with a small degradation in the number of messages per second. Thus, increasing message size by approximately two orders of magnitude increases bits per second by approximately two orders of magnitude. Between 64 bytes and 1,024 bytes lies an equilibrium point at which the number of messages per second and the number of bits per second are optimally matched. It remains to be seen whether tuning can push the crossover of these two curves toward the larger message sizes, indicating that the API overhead is minimal with respect to message size. But the initial results are encouraging.

The MCAPI specification doesn't complete the multicore-programming model, but it provides an important piece. Programmers can find enough capability in MCAPI to implement significant multicore applications. We know of two MCAPI implementations that should help foster adoption of the standard.

Three active workgroups—Multicore Programming Practices (MPP), Multicore Resource Management API (MRAPI), and Hypervisor—continue to pursue the Multicore Association roadmap. MPP seeks to address the challenges described in the sidebar. This best-practices effort focuses on methods of analyzing code and implementing, debugging, and tuning that are specific to embedded multicore applications. These include current

techniques using SMP and threads, and forward-looking techniques such as MCAPI.

MRAPI will be a standard for synchronization primitives, memory management, and metadata. It will complement MCAPI in both form and goals. Although operating systems typically provide MRAPI's targeted features, a standard is needed that can provide a unified API to these capabilities in a wider variety of multicore-system contexts.

The Hypervisor working group seeks to unify the software interface for paravirtualized operating systems (that is, operating systems that have been explicitly modified to interact with a hypervisor kernel). This standard will allow operating system vendors to better support multiple hypervisors and thus more multicore chips with a faster time to market.

We believe that multicore-programming model standards can provide a foundation for higher levels of functionality. For example, we can envision OpenMP residing atop a generalized multicore-programming model, language extensions for C and C++ to express concurrency, and compilers that target multicore standards to implement that concurrency. Another natural evolution to programming models, languages, and compilers would be debugging and optimization tools that provide higher abstraction levels than are prevalent today. With MCAPI, it could also be possible for creators of debugging and optimization tools to consider how to exploit the standard. MICRO

References

1. M. Creeger, ''Multicore CPUs for the Masses,'' *ACM Queue,* vol. 3, no. 7, 2005, pp. 63-64.
2. J. Donald and M. Martonosi, ''Techniques for Multicore Thermal Management: Classification and New Exploration,'' *Proc. 33rd Int'l Symp. Computer Architecture* (ISCA 06), IEEE CS Press, 2006, pp. 78-88.
3. D. Geer, ''Chip Makers Turn to Multicore Processors,'' *Computer,* vol. 38, no. 5, 2005, pp. 11-13.
4. S. Bell et al., ''TILE64 Processor: A 64-Core SoC with Mesh Interconnect,'' *Proc. Int'l Solid-State Circuits Conf.* (ISSCC 08), IEEE Press, 2008, pp. 88-89, 598.
5. ''P4080: QorIQ P4080 Communications Processor,'' Freescale Semiconductor; http://www.freescale.com/webapp/sps/site/prod_summary.jsp?fastpreview=1&code=P4080.
6. ''Intel Microarchitecture (Nehalem),'' Intel 2008; http://www.intel.com/technology/architecture-silicon/next-gen/index.htm.
7. D.C. Pham et al., ''Overview of the Architecture, Circuit Design, and Physical Implementation of a First-Generation Cell Processor,'' *IEEE J. Solid-State Circuits,* vol. 41, no. 1, 2006, pp. 179-196.
8. *IEEE Std. 1003.1, The Open Group Base Specifications Issue 6,* IEEE and Open Group, 2004; http://www.opengroup.org/onlinepubs/009695399.
9. *OpenMP API Specification for Parallel Programming,* OpenMP Architecture Review Board, May 2008; http://openmp.org/wp.
10. *CORBA 3.1 Specification,* Object Management Group, 2008; http://www.omg.org/spec/CORBA/3.1.
11. *Multicore Communications API Specification,* Multicore Association, Mar. 2008; http://www.multicore-association.org/request_mcapi.php?what=MCAPI.
12. *TIPC 1.5/1.6 Protocol Specification,* TIPC Working Group, May 2006; http://tipc.sourceforge.net.

Jim Holt manages the Multicore Design Evaluation team in the Networking Systems Division at Freescale Semiconductor. His research interests include multicore-programming models and software architecture. He has a PhD in computer engineering from the University of Texas at Austin. He is a senior member of the IEEE.

Anant Agarwal is a professor in the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology, where he is also an associate director of the Computer Science and Artificial Intelligence Laboratory. In addition, he is a cofounder and CTO of Tilera. His research interests include computer architecture, VLSI, and software systems. He has a PhD in electrical engineering from Stanford University. He is an ACM Fellow and a member of IEEE.

Sven Brehmer is president and CEO of PolyCore Software, a company providing

multicore software solutions. He has many different technical interests, including embedded systems software, multiprocessing, and alternative energy, especially multicore. He has an MS in electrical engineering from the Royal Institute of Technology, Sweden. He is the founding member of the Multicore Association and the MCAPI working group chair. He is a member of the Multicore Association.

Max Domeika is an embedded-tools consultant in the Developer Products Division at Intel, where he builds software products targeting the Embedded Intel Architecture. His technical interests include software development practices for multicore processors. He has a master's degree in computer science from Clemson University and a master's degree in management in science and technology from the Oregon Graduate Institute. He co-chairs the Multicore Programming Practices working group within the Multicore Association. He is the author of *Software Development for Embedded Multicore Systems.*

Patrick Griffin is a senior design engineer on the software team at Tilera. His research interests include parallel-programming models and high-speed I/O for embedded applications. He has a MEng in electrical engineering and computer science from the Massachusetts Institute of Technology.

Frank Schirrmeister is director of Product Management in the Solutions Group at Synopsys. His research interests include electronic system-level design and verification, particularly presilicon virtual platforms for software development, architecture exploration, and verification. Schirrmeister has an MS in electrical engineering from the Technical University of Berlin with a focus on microelectronics and computer science. He is a member of the IEEE and the Association of German Engineers (VDI).

Direct questions and comments about this article to Jim Holt, Freescale Semiconductor, 7700 West Parmer Lane, MD:PL51, Austin, TX 78729; jim.holt@freescale.com.