Programming Models and Development Software for a Space-Based Many-Core Processor

Stephen P. Crago, Dong-In Kang, Mkyung Kang, Robert Kost, Karandeep Singh, Joseph Suh, John Paul Walters
University of Southern California / Information Sciences Institute
Arlington, Virginia, USA
crago@isi.edu

Abstract—The Maestro processor is a 49-core many-core processor for space based on the TILE64 architecture and implemented in rad-hard-by-design technology by Boeing. In this paper we discuss the programming models for Maestro, the implications of the programming model on fault tolerance and flight software, and the software development tools that have been developed for Maestro. The software described here is experimental development software that allows application and algorithm evaluation on the architecture, but we believe this software can be used as the basis for flight software. The software includes libraries, performance analysis and optimization tools, and compilers. While this work was done on the Maestro chip, the principles discussed can be applied to any multi-core or many-core processor.

Multi-core programming; space-based processing; parallel software

I. INTRODUCTION

Since at least 2004, commercial microprocessor companies have been adding cores rather than increasing the performance and clock speed of single cores or uniprocessors. This shift in strategy was forced on microprocessor developers because they could no longer increase single-core performance with reasonable efficiency. Multi-core architectures improve power efficiency and performance by exploiting more parallelism at lower clock rates. Many-core architectures extend this trend by combining dozens of cores on a single processor chip. The Maestro microprocessor, designed by Boeing, is the first general-purpose many-core microprocessor for space [1]. The Maestro processor achieves radiation hardness using rad-hard by design technology.

As the number of cores increases, software development becomes more challenging, especially for applications for which performance is critical. Programmers (or tools and libraries) optimizing for performance must consider interconnects, complex memory hierarchies, and limited I/O resources. The memory hierarchy becomes deeper as memory associated with remote cores is accessed and techniques are used to avoid consuming limited off-chip memory bandwidth. A variety of programming models that expose parallelism must be considered, and the ability of compilers and libraries to exploit parallelism will affect software productivity and costs significantly.

The authors of this paper have addressed these software challenges for Maestro through a variety of programming models, libraries and tools. Both sequential and parallel programming models can be used to develop Maestro software. For parallel programming, shared memory and message passing programming models are supported. The Linux and VxWorks operating systems both run on Maestro, and a variety of application libraries and performance analysis tools support software development. We also expect that new programming models will be developed that ease programming and provide important properties like fault tolerance. The current development software for Maestro is a robust environment for application development and evaluation and for performance analysis and optimization. The software is not flight-qualified.

While multi-core architectures do present programming challenges, they also provide opportunities for space processing, beyond the raw processing performance they provide. The inherent spatially separated resources and redundancy in multi-core have advantages for real-time and fault-tolerant processing. For real-time processing, cores can be dedicated to real-time tasks, avoiding the overhead and scheduling challenges associated with time multiplexing. For fault tolerance, reliability techniques can be exploited in software, with software partitioned across cores as appropriate to achieved required levels of reliability. This fault tolerance can be used to achieve robustness in the context of radiation effects, other hardware faults, or software faults. While these multi-core software-based techniques do not entirely replace other hardware-based reliability techniques, they are flexible, so that fault tolerance can be tailored to specific application needs or dynamic mission requirements.

In this paper, we present an overview of the prototype software that we have developed for the Maestro processor. Section II has an overview of the Maestro architecture. Section III summarizes the programming models available for Maestro, and Section IV describes the tools and libraries for Maestro with performance results. In Section V, we discuss flight software considerations, including real-time processing and fault tolerance. Section IV contains conclusions and future work.

II. MAESTRO ARCHITECTURE

The Maestro processor is a processor for space designed using rad-hard-by-design techniques and based on the
commercial TILE64 processor. The TILE64 from Tilera is the first commercial, general-purpose many-core microprocessor [2]. Maestro has 49 cores on a single die. Each Maestro core executes a 3-way Very Long Instruction Word (VLIW) Reduced Instruction Set Computer (RISC) instruction set, with Single-Instruction Multiple-Data (SIMD) operations for 16-bit and 8-bit operations and inorder execution. Each core has a memory management unit for virtual memory support and executes an independent control thread or process. A block diagram for Maestro is shown in Figure 1.

The processor has four DDR2 (double data rate) memory interfaces, four XAUI, and two Gigabit Ethernet MAC (Medium Access Control) interfaces, as well as serial and programmable interfaces.

The cores of the Maestro are interconnected by a set of 2D-mesh networks collectively called iMesh™. iMesh is a set of five inter-core networks, each of which perform different functions. Routing through intermediate tiles does not disrupt intermediate tile processors (cores). The Memory Dynamic Network (MDN) routes data between cores and between cores and main memory and is accessible only through the caches. The Tile Dynamic Network (TDN) routes memory data between tiles and is also accessible only through caches. The I/O Dynamic Network (IDN) routes data from cores to I/O interfaces and is accessible from the cores from operating system code. The last two networks are accessible by user-level code. The User Dynamic Network (UDN) and the Static Network (STN) are both accessible via a register-level first-in first-out interface from user code directly via assembly code or through a C library. The UDN requires a header for each packet that specifies the destination core. The STN requires routes to be pre-programmed, which eliminates the need for the header, at the cost of less flexible (static) routing. Packets on the dynamic network can be from 1 to 128 32-bit words, and each packet has one header word. Data in the static network is routed on a word-by-word basis. The cores can be divided into rectangular partitions to provide protection between independent programs.

As of this writing, Maestro hardware is still under test, but the cores are expected to run at up to 350 MHz, providing peak performance of 50 giga-operations per second and 25 giga-floating-point operations per second. Performance results in this paper are based on cycle-accurate simulations run at 260 MHz.

III. PROGRAMMING MODELS

The flexibility of the Maestro architecture, which it inherits from the TILE64 processor, allows a wide variety of programming models to be implemented. The Maestro processor runs Linux SMP and VxWorks Symmetric MultiProcessing (SMP) and Asymmetric MultiProcessing (AMP). Parallelism models supported include explicit thread-based and process-based parallelism, as well as Tilera’s iLib on Linux and VxWorks’ task-based parallelism model. Linux provides shared memory via pthreads and OpenMP [3], and VxWorks provides shared memory, MPI-based message passing is implemented for Linux, and Tilera’s iLib also provides several message passing abstractions. The R-Stream compiler from Reservoir Labs performs automatic parallelization, primarily for loop-based C code.

The TILE64 processor and its commercially available software support Linux SMP (based on the 2.6 kernel) and this operating system has been ported to the Maestro processor. Modifications to the commercial Tilera implementation mainly focused on support for the floating-point coprocessor included in each core, ensuring that context-switching included the state of the floating-point...
coprocessor and that exceptions generated by the floating-point coprocessor are handled correctly.

The Linux implementation for Maestro supports a robust development environment for American National Standards Institute (ANSI) standard C and C++. Maestro has an Eclipse-based graphical IDE (Integrated Development Environment). Standard libraries, such as libc and libm are supported, and the Tilera-specific iLib and Tilera Multicore Components (TMC) libraries are supported for efficient inter-core communication. Drivers are provided for the I/O interfaces of the Maestro processor, which include Gigabit Ethernet and 10-Gigabit Ethernet as well as serial interfaces such as Universal Asynchronous Receiver/Transmitter (UART). The Linux environment supports oporfile [4] and the Performance Application Programming Interface (PAPI) and perfmon2 for performance profiling [5][6].

The standard C/C++ compiler targets individual cores, with libraries used for shared memory access and synchronization and inter-core communication and is the primary development environment recommended for application development. The standard compiler also includes support for OpenMP 2.5, which supports automatic parallelization if user directives. The R-Stream compiler performs automatic parallelization for intensified C code that is primarily composed of regular loops.

For shared memory programming, the recommended interface is Posix threads, known as pthreads. Pthreads is a standard interface for lightweight threads that are, in the case of Maestro’s software environment, contained in Linux processes. Pthreads has support for creating and destroying threads and for mutex and condition variables. Individual threads can be pinned to specific cores if desired, and the iLib and TMC have functionality for accessing features specific to the Maestro (TILE64) processors, such as the user-dynamic network or neighborhood caching.

For message passing programs, Linux processes can be used to create parallel units of execution and then there are a number of options for passing messages between processes. Traditional system calls such as sockets can be used, and iLib and TMC provide functions for message passing. MPI has also been implemented for portable message passing programs. The programmer can also access the UDN and STN directly since they are register-mapped.

The TILE64 and Maestro processors also support VxWorks SMP and AMP. This support was added under the context of this work and it is not fully commercially supported at the time of this writing, but the ports are sufficient to allow code development and application evaluation. VxWorks SMP supports a single address space and operating system image running across all cores while VxWorks AMP supports an independent address space for each core. The VxWorks ports support C development using the Tilera Eclipse-based programming environment.

The VxWorks ports to TILE64 and Maestro support tasks and C language programming, but not real-time processes or C++. Inter-task communication mechanisms include shared memory, semaphores (SMP only), mutexes and condition variables, message queues and pipes, VxWorks events, message channels, sockets and remote procedure calls, and signals. All communication mechanisms in VxWorks use shared memory rather than the UDN network of Maestro.

As part of the implementation of VxWorks AMP, a multi-client hypervisor, which provides a layer of software between the hardware and the operating system was developed. This multi-client hypervisor allows multiple instances of operating systems to run on different sub-sets of cores on the Maestro (and TILE64) architectures. These instances can include any combination of Linux and VxWorks.

IV. TOOLS AND LIBRARIES

The primary compiler for Maestro is based on Tilera’s tile-cc compiler, which has been modified to support the floating-point coprocessor included in each Maestro core. The floating-point coprocessor is accessed through special-purpose registers, although C/C++ programmers need not explicitly reference this interface. We measured the performance of Maestro’s floating-point operations using four different micro-kernels: 64-point FFT, 12-tap FIR filter, matrix multiplication (MM), and vector addition (VADD). The result is summarized in Figure 2.

![Figure 2. Single-core performance](image)

For true automatic parallelism, the R-Stream prototype compiler from Reservoir Labs targets TILE64 (and TILEPro64) as well as Maestro. The R-Stream compiler takes stylized C (standard C written in a style that is amenable to compiler analysis) and uses the polyhedral model to analyze the code, parallelize, and optimize for locality and appropriate granularity. The R-Stream compiler emits parallel C code, which is then run through the tile-cc compiler to target the Maestro processor. The R-Stream compiler has been found to achieve reasonable speedup on the 49 cores of Maestro. Detailed performance characterization will be performed as future work.

The VSIPL Vector Signal Image Processing Library is a sequential application programming interface (API) that provides portability for embedded platforms [7]. VSIPL is an open standard that was initially supported by the Defense Advanced Research Project Agency (DARPA). The implementation for Maestro supports the VSIPL 1.3 Core and Core-Lite profiles. The University of Maryland developed the implementation by starting with the Tactical Advanced Signal Processing (TASP) Plus Plus Op86
reference implementation and performing automatic optimizations such as loop unrolling and prefetching.

FFTW (Fastest Fourier Transform in the West) [8] is a portable, self-adaptive implementation of the Fast Fourier Transform that has been ported to Maestro. FFTW is composed of codelets that implement transforms that can be assembled to perform FFTs. FFTW performs a planning stage, which can be run off-line and determines which combinations of codelets perform best on a target architecture. Results of planning can be stored as wisdom for future use. The execution stage then takes the selected codelets and uses them to perform the user-specified FFT. FFTW has a sequential implementation and a parallel implementation that targets pthreads; both the sequential and parallel implementations run on Maestro. We were able to obtain 145 MFLOPS (0.36 FLOPs/Cycle) on a single Maestro core with double-precision computations and 202 MFLOPS (0.50 FLOPs/Cycle) on a single Maestro core with single-precision computations running at 260 MHz. FFTW performance for three FFT sizes on Maestro when independent FFTs are computed in parallel on multiple tiles is shown in Figure 3.

![Figure 3. FFTW performance scaling with FFT size](image)

The FFTW library was used extensively in implementing the Complex Ambiguity Function (CAF) [9], which is a reference space application. CAF computes the time and frequency delay of arrival between two input signals. We used FFTW wisdom to speed up FFT computations and DMA engines in the tiles to improve the I/O performance. Performance numbers for CAF are presented in Figure 4.

![Figure 4. CAF single-iteration execution time](image)

Maestro also supports MPI (Message Passing Interface) for parallel programming. MPI is portable and commonly used for high-performance applications on clusters and distributed memory architectures, and the implementation on Maestro provides portability as well as a familiar, straightforward message-passing interface. The results for send/receive pair communication using two cores on TILE64 are shown in Figure 5. The results assume cold instruction caches and warm data caches, which corresponds to a scenario when other functions are occupying the instruction cache, but the data to be sent has been recently generated and is in the data cache. The latency is about 50 µsec at 260MHz for small message sizes. As the message size gets longer, the initial processing time is amortized and the data transfer time per word is reduced as the message size increases. When the message size is 64K words, communication takes only 12 cycles per word. The components of execution time for message passing for different data sizes are shown in Figure 6. For large message sizes, the overhead costs are amortized over more words, but also more data cache misses are also incurred.

![Figure 5. Number of cycles per word for MPI send/receive](image)
A run-time monitor (RTM) implements instrumented versions of pthreads and MPI to provide profiling for shared memory and message passing programs, respectively. In the case of pthreads shared memory programs, the run-time monitor collects information about which threads access individual synchronization variables, how often the synchronization variables are accessed, how long threads spend waiting for those variables, and other similar statistics. In the case of message passing programs, the run-time monitor collects profiling information about communication topology, average message size, total data sent, etc. The overhead for performance monitoring is very large – roughly 10x the execution time of the base function call of iLib message passing for small messages. This is expected since iLib is intended to allow access to the UDN with little overhead, and the performance monitoring code can be much more complex. For larger message sizes, overhead is much more reasonable/acceptable. For example, for a message size of 64KW, overhead is less than 30%.

The information collected by PAPI, perfmon2, and the run-time monitor is integrated with the Parallel Performance Wizard (PPW) tool [10]. PPW is a performance analysis tool that provides load balancing analysis, statistical data tables, and communication/data transfer analysis. It has a graphical display for parallel performance profiling. Typical overhead is roughly 10,000 cycles per function call for a large number of profiled function calls, compared to 50,000 cycles per function call for a small number of function calls. The Maestro port of PPW includes the base-profiling interface similar to what is supported for other programming languages and extends support for iLib and TMC. In addition, Eclipse plugins were created to aid in using the tools more seamlessly, as well as easily showcasing the results.

V. FLIGHT SOFTWARE CONSIDERATIONS

While multi-core has potential performance advantages and programming challenges, it also leads to some unique opportunities for flight software. First, having many cores per processing chip means that cores can be dedicated to individual processing tasks, improving real-time responsiveness and reducing the need for time-sharing of processing resources. This spatial partitioning of processing tasks also means that processing tasks can run simultaneously, with real-time tasks running on dedicated cores, without the overhead of context switching, leading to tighter coupling of sensor processing and control, as shown in Figure 7.

Real-time processing capabilities have been provided by RTOS (real-time operating system) software for decades. A RTOS is different from a general-purpose operation system because the RTOS can make guarantees about task completion. The overhead for these guarantees may cause some loss in efficiency, programmability, and portability, but these guarantees are necessary in many processing modules. A RTOS can provide guarantees about task completion primarily by the way it handles task exemption, timers, and interrupts. If a task must complete by a certain time, it cannot be interrupted and kept on an inactive work queue while its deadline passes. At the same time, some interrupt handling routines must be made active within certain time bounds when there are real-time requirements for interrupt handling. While a RTOS cannot make unreasonable time constraints possible, it can ensure that the processing scheduling is done by a protocol that can be understood and controlled by the programmer, making real-time processing possible when programmed properly. On traditional uniprocessors, a RTOS manages context switching to implement the necessary properties.

Multi-core architectures have been developed primarily for performance and efficiency reasons, but, as a side effect, they have properties that are amenable to real-time processing, although these characteristics have not yet been fully exploited. As discussed in the preceding paragraph, uniprocessor resources are shared over time (time multiplexing), and context switching must be carefully
managed to provide real-time processing. The same is true for multi-core architectures when processing cores are time-
multiplexed, which is the common way multi-core
architectures are used. VxWorks for Maestro allows
processing cores to be multiplexed between multiple tasks in
a way that supports real-time processing. However, in multi-
core architectures, it is also possible to dedicate processing
cores to real-time tasks. When cores can be dedicated to real-
time tasks, those processing resources can be available to
satisfy real-time requests without interrupting other tasks. In
order to completely dedicate cores, the number of available
cores must be larger than the number of real-time tasks;
when the number of real-time tasks can exceed the number
of cores available, hybrid techniques can be used. Having a
core dedicated to a task does not guarantee that other tasks
will not interfere with the dedicated cores ability to make
real-time guarantees. A processing core may depend on
memory hierarchy performance and behavior, on-chip
interconnect, and external IO. However, having dedicated
cores does make scheduling easier and makes it easier to
meet real-time constraints with less context-switching
overhead.

In order to ensure that dedicated (or allocated) tasks
handle interrupts that signal external events, which are
critical to real-time systems, the architecture must support
the capability to route interrupts to individual cores. The
Maestro architecture supports this capability. If an
architecture did not support this capability directly in hard-
ware, the core that does handle interrupts could be
responsible for sending messages to those cores that are
allocated by software for real-time processing of interrupts.
This layer of indirection would deliver a similar capability,
but with additional overhead.

Hardware-managed memory hierarchy (caching) can also
interfere with real-time processing. The latency difference
between an on-chip memory reference and an off-chip
memory reference can be two orders of magnitude, and with
a general purpose operating system, e.g. Linux, the
programmer cannot ensure that data is kept in a cache when
another task may run and pollute the cache. In the Maestro
architecture, local memory can be set to operate as an
explicitly software managed local buffer, so in combination
with RTOS task management or dedicating processing cores,
we can make sure the programmer can control which data is
kept in local memory and therefore can control memory
latency and task completion times.

The on-chip interconnect is another potential source of
processing time uncertainty. If a task depends on
communication with processing assigned to other cores, that
communication would go through the on-chip network. Even
if two cores have been dedicated to real-time tasks and have
coordinated to ensure communication will be done within
constraints consistent with the real-time requirements, the
possibility exists that communication between other cores
will interfere with the communication between real-time
tasks. Fortunately, there are two properties in Maestro that
we can use to ensure the network does not prevent real-time
constraints from being met. The first is that Maestro supports
hardwalling, which allows partitioning in the architecture to
prevent interference between partitions in the UDN. If a set
of communicating real-time tasks is kept within a hardwalled
partition, it can be guaranteed that UDN communication
from other partitions will not interfere. The second property,
which is common to most scalable on-chip networks, is that
dimension-ordered routing is used in the on-chip network.
This routing technique is used primarily because it allows the
architecture to avoid deadlock, but it has the side effect that
communication routes can be statically determined, and so
the programmer or system software can allocate cores with
known communication patterns to avoid interference
between unrelated groups of tasks when necessary, such as
when real-time guarantees are important.

Multi-core architectures also have some natural
advantages for fault tolerance. Fault tolerance is often (but
not exclusively) achieved through redundancy, and multi-
core processors have replicated resources that can be used
for redundancy. In the case of the Maestro processor, there
are opportunities for redundancy at many levels. In the
hardware for Maestro, many techniques have been used to
reduce the probability of single-event upsets, but software
must still be used to increase reliability and to protect from
other kinds of errors (e.g. software bugs). Within a
processing core, opportunities for replication in software are
available by exploiting SIMD instructions and between
independent instructions packed within a very-long
instruction word VLIW. Tasks can also be replicated
between cores, and data can be replicated between
memories. The on-chip networks and input/output (IO) ports
also have redundancy that can be exploited for fault
tolerance. All of these sources of redundancy can be
exploited statically, when processing and redundancy needs
are static and pre-defined, which is consistent with the
present way of handling such issues, or dynamically when
processing needs and fault tolerance requirements are
expected to change over the course of a mission.

Multi-core architectures have the unique ability to
dedicate on-chip processing resources to redundant
processing. Exploiting such redundancy is similar to the
traditional technique of triple modular redundancy.
However, there are some important differences. First,
because the redundant resources are on the same chip, the
processing is not protected from a chip failure, and this
vulnerability must be considered in mission system
engineering (and is beyond the scope of this project).
Second, the on-chip redundancy of multi-core processors can
be controlled by software, which can be an advantage. The
amount and type of redundancy can be tailored to a specific
application. For example, there may be times when an
answer can be checked for correctness or reasonableness
much more cheaply than completely replicating an entire
processing chain. There may be times when certain kinds of
errors can be tolerated, e.g. minor errors in inherently noisy
sensor processing. This flexibility can be exploited statically
at design time, when a system engineer can determine
through analysis and experimentation and testing what kind
of redundancy is needed for a specific mission and
processing function. Alternatively, it can be determined
dynamically when a system may react to dynamic processing
needs and changing processing environments (e.g. increased radiation upsets).

Processing resources are not the only resources that are used for on-board processing in multi-core architectures, and these other resources must also be managed for fault tolerance. On-chip networks are used to support inter-core communication, memory references, and interrupts. In the case of the Maestro architecture, and most others, these networks have natural redundancy. For example, there are multiple networks that are available to the user to send data between cores. I/O resources also have redundancy. For example, the Maestro architecture has four XAU1 interfaces and four memory (DDR2) controllers and interfaces.

While fault tolerance and real-time constraints impose orthogonal sets of requirements on a processing system, the underlying mechanisms share some commonality and the implementations will affect each other. Resource management techniques, such as the allocation of processing cores to processing tasks, must consider both sets of requirements. A task must be assigned the appropriate resources to satisfy real-time requirements while implementing redundancy for fault tolerance.

In order to demonstrate core-level fault tolerance techniques, we developed a rudimentary fault tolerance library. This library is not intended as a complete system-level fault-tolerance solution, but as a demonstration of some of the software-based fault tolerance capabilities of Maestro. For replication-based fault tolerance, we implemented process-level and thread-level replication. Process-level replication (PLR) provides redundancy without programmer intervention for error that can be detected in file output). This technique could be extended to other kinds of output, including inter-process messages and I/O. The overhead incurred by our PLR library for an image compression application (compressing a 904k image using OpenJPEG2K [11]) is shown in TABLE I. The overhead reported is the extra time incurred for the execution of the process to create and start the replicated process(ies) and to compare results. The replication also consumes additional core resources to execute the replicated process(ies).

<table>
<thead>
<tr>
<th>Phases</th>
<th>Cycles</th>
<th>Total Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>17,939,566</td>
<td>0.42%</td>
</tr>
<tr>
<td>Interpolation</td>
<td>13,395,244</td>
<td>0.61%</td>
</tr>
<tr>
<td>Fini</td>
<td>363,815,968</td>
<td>16.63%</td>
</tr>
</tbody>
</table>

TABLE I. PLR RUN-TIME OVERHEAD FOR IMAGE COMPRESSION.

Our thread-level replication (TLR) requires the programmer to modify code to implement fault tolerance, but it allows arbitrary error checking. Our library can automatically check contiguously stored data output and allows user-specified checking of more complex data outputs. The overhead incurred by our TLR library is shown in TABLE II. Like with TRL, we report the additional execution time, not computing resources required for the replicated threads.

<table>
<thead>
<tr>
<th></th>
<th>TLR OVERHEAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average time for creating 3 threads</td>
<td>4,850,853 cycles</td>
</tr>
<tr>
<td>Overall replication and joining costs for an empty function</td>
<td>7,066,721 cycles</td>
</tr>
<tr>
<td>Output comparison costs for an empty function</td>
<td>103,696 cycles</td>
</tr>
</tbody>
</table>

TABLE II. TLR OVERHEAD

Our process-level and thread-level replication techniques use voting to detect and correct errors at the application level. Both of these techniques, as currently implemented, detect and correct program-level data errors, but depend on reliable program execution and control.

In order to detect control errors, we implemented a distributed watchdog technique. This technique uses a combination of timer-based interrupts and programmed checks to determine whether each application process is alive. If a process dies, our checkpoint and rollback library can be used to recover from faults. Our checkpoint and rollback software is based on the Linux-CR v1.5 library that is publicly available [12]. The time taken to checkpoint a parallel matrix multiplication application (matrix sizes 1024x1024) is shown in Figure 9.

![Parallel Matrix Multiply Checkpoint Times](image)

Figure 9. Matrix multiply checkpoint times as a function of number of cores

VI. CONCLUSIONS AND FUTURE WORK

We have presented an overview of programming models and implementations for the many-core Maestro microprocessor for space. The tools described include a simulator, compilers, integrated development environment, and performance analysis tools. The programming environment described also includes application libraries, including VSIPL and FFTW, and system libraries including pthreads, MPI, TMC, and iLib. We described support for both Linux and VxWorks.

Future work includes detailed system-level performance characterization. This performance characterization is difficult to do at the system level in simulation, because of simulation time, and will be performed once the Maestro board support package is functional and software can be run on hardware. Additional support for the VxWorks operating system, equivalent to what is supported for commercial architectures is also left for future work. Parallel performance analysis and debug are still challenges, and additional tool support would be valuable. Finally, the
development of flight software, including qualification, is left as future work. This development should not be limited to testing and hardening of previously developed software, but should include the development of programming models that improve productivity and limit fault propagation and improve verification and validation and real-time capabilities while supporting the needs of flight missions.

ACKNOWLEDGMENT

The authors would like to thank our collaborators at the University of Maryland, Reservoir Labs, MIT, Wind River, and Boeing.

REFERENCES


