Application of Advanced Multi-Core Processor Technologies to Oceanographic Research

Mark R. Abbott
104 CEOAS Administration Building
College of Earth, Ocean, and Atmospheric Sciences
Oregon State University
Corvallis, OR 97331-5503
phone: (541) 737-5195 fax: (541) 737-2064 email: mark@coas.oregonstate.edu

Grant Number N000141110104
Grant Number N000141210298
http://www.ceoas.oregonstate.edu

LONG-TERM GOALS

Improve our ability to sense and predict ocean processes, utilizing state-of-the-art information processing architectures.

OBJECTIVES

Next-generation processor architectures (multi-core, multi-threaded) hold the promise of delivering enormous amounts of compute power in a small form factor and with low power requirements. However, new programming models are required to realize this potential. Our objectives are to deploy data processing and vehicle control systems onto a variety of "systems on a chip" (SOC) to provide more autonomous functionality.

APPROACH

The overarching theme of this work relates to the application of advanced heterogeneous processors in an embedded environment to high bandwidth signal processing and vehicle autonomy. Our previous work included the development of a task dispatcher model for rapid development of signal processing applications, scheduling tasks, and a graphical programming language. We have focused on a range of power-constrained systems and development environments, including an inexpensive consumer-grade drone. We are also integrating new approaches to user interfaces to these environments.

WORK COMPLETED

In the last year, we have continued development on the compute device scheduling system. The in-progress and planned work includes porting the agent application to Linux, enhancing the local scheduling facilities, and supporting embedded applications. The result of this work will enable greater support for compute hardware and an increased ability to embed the scheduler in vehicles. The arc of our previous work has been to generalize the first generation scheduler from local scheduling on the Cell/B.E. to heterogeneous cluster scheduling and to improve ease of use through a graphical user interface. In the last year we started the work of bringing these enhancements back to locally

scheduled workflows and custom platforms (in their current state, most embedded systems are more like the Cell/B.E. than commodity compute systems). Furthermore, we are addressing fault tolerance, low power, and latency management to improve vehicle-readiness.

By porting the scheduler agent to the Linux operating system we were able to dramatically increase the reach of our scheduling system. The vast majority of computation-oriented co-processors are designed to work on Linux systems, and Linux is the most common OS for embedded systems. In the past year, we have been systematically redesigning the major portions of the compute scheduling agent to not only support Linux, but to be as cross-platform as practical. Our choice to use industry standards such as XML, OpenCL and Bonjour proved to be a good decision, as it has reduced the number of modules that we have had to restructure. We are aware of the delicacy of the intellectual property landscape surrounding Linux, and have taken pains to ensure that our product is unencumbered by "viral" licenses that would require reciprocal code sharing.

In addition to supporting a greater breadth of platforms, we are improving the handling of real-time sensor data on power-constrained systems. The existing implementation of the scheduler is optimized for dynamically changing user applications using line power. An example of the consequences of these optimizations is that the scheduler will benchmark a new compute kernel implementation immediately, and on every compute element available. This is a necessary action to have complete and up to date compute cost metrics, but can be time and power consuming. Furthermore, the existing scheduler attempts to maximize the computational throughput of the cluster. There are cases where the most important metric of a system is latency, and even cases where minimizing power draw is the most important goal. In the past year, we have been working on designing the models and algorithms that have the flexibility to schedule using the metrics outlined above, including whether, and to what extent, these metrics can be combined and prioritized automatically.

We tested our concepts on two control platforms for a Parrot AR Drone. The first platform is based on a Netduino microcontroller and coded in the .NET Micro Framework. This controller is capable of communicating with the Parrot AR Drone via the serial port on the bottom of the drone. Using this port, the controller can take off the quadricopter, land it, and have general control over the drone while it is in operation. This microcontroller is also using sensor input (GPS and compass) to determine the location and heading of the drone and relay that information over a wireless network with a wireless communication device or over a wired network if plugged in.

The second platform for controlling the drone is based on a Raspberry PI microcontroller running a version of the Linux operating system. This controlled was coded in C++ and is also capable of performing all the functions of the .NET microcontroller, with the added benefit of being portable between other Linux based microcontrollers (for example, the Beagle Bone platform). All the sensor libraries are general and could be reused for other projects as needed. Another benefit of this Linux based platform is that the microcontrollers are generally faster than the .NET version, allowing for more computation to be performed on the microcontroller itself, but with the drawback that it does consume more power. There are some additional features included in the C++ version of the code that are not available in the .NET version. The first allows for remote control in which users can give the microcontroller a set heading and using the built in compass, the controller will have the drone maintain that heading. There are also additional TCP communications stack capabilities included with this microcontroller as the full TCP stack is a large overhead for the slower .NET microcontroller.

Both of these platforms utilize a program running on the Parrot AR Drone to relay messages to the built in control system of the drone. This program takes a command packet from the microcontroller through the serial port of the drone, and translates it to a message the drone will understand. Once this conversion is done, it sends the message to the specific socket on the drone for that command. This allows us to not have to write our own control software for the drone and continue to use all the built in features of the drone. As manufacturer updates their control software, our implementation will receive those same benefits. This also allows us to use the sensors in the drone in our own implementations.

RESULTS

In support of the task of developing an embedded scheduler, and making it appropriate for use in vehicles, we have collected a representative sample of development kits across a variety of architectures, brands, and models. Each instruction set architecture (ISA) and processor is designed within the constraints set by physics, computation theory, and often, economics. The best solutions to the problems of chip selection vary according to the target application. It is vital that we understand how the compromises chosen by vendors affect our goals of high-performance on-vehicle computation. We have included a table of platforms that we have acquired according to their rough power envelope.

Power Envelope	Architecture	Name	OpenCL
< 500 mW	ARM M series	TI Stellaris ST Micro STM32 NXP LPC series	No
	Proprietary	Microchip PIC32/DSPIC	No
> 500 mW; < 5 W	ARM Cortex	TI OMAP TI Sitara Broadcom BCM2835	Varies
	FPGA	Xilinx Spartan Altera Cyclone	Possible

The C++ implementation of the drone controller program has a Haskell-based library allowing remote control of the Parrot AR Drone using a functional programming language. This library allows a user to describe a general path (Straight line, or curved path, from latitude/longitude to latitude/longitude) and time (ex. Start time 0:01:03 end 0:02:03 would be a 1 minute flight path) of the waypoints that the user would like the drone to use. With this information, the program will calculate a path and speed for the drone to follow to reach the waypoints at the given times. In the event the drone is off course (due to wind or other problem), this library will attempt to make course corrections and return the Parrot to the correct course.

In order to accomplish the above parts of the Parrot AR Drone, we developed a robust communications protocol allowing the drone to receive commands and relay information from a base station or other device. This protocol needed to be able to detect errors in transmission as well as be expandable in the event we needed to add commands or attach new sensors. The protocol we developed uses a simple checksum to ensure correctness of data, and the library in place allows us to easily encapsulate the data

with the given checksum and header information. In the event we need to expand the protocol, we can simply add a new header tag and then create the new data packet and the rest of the communication (checksum creation and data transmission is done with the current libraries).

On the data communication portion of the controller, there is an API in place enabling users to subscribe to sensor data or query individual sensors. Due to limitations in processing capacity of the microcontroller and not wanting to take away from compute cycles, only one person can subscribe to the data at a given time, but that data could be sent to a relay station which could allow for more subscriptions or push updates as needed. There are two ways to retrieve data from the microcontroller, either through subscriptions or with direct queries. The direct query is simply a request for data from a specific sensor and the control will respond with the data. The subscription portion allows for subscriptions at given time intervals, or whenever the data is updated (not all sensors allow for subscribing to updates since they are updated often and would create a large overhead on the device for transmission).

For central data storage, a memcached instance running a high performance parallel version of memcached allowing for over one million requests per second has been developed, as well as a ZFS file system for archiving the data. We are currently developing communication, caching, and archiving procedures. A protocol will also need to be developed to allow many users to locate and utilize this data quickly and simultaneously.

A user interface was developed showing how data could be subscribed to or queried as needed from the controller. This interface is simple, subscribing to a GPS feed from the controller and displaying the drone location on a Bing maps interface, or allowing the user to see the current heading and acceleration of the Parrot AR Drone on request. This implementation is done in .NET and implements all the libraries needed to communicate with the controllers (checksum and packet decoding). The current implementation is being developed for use on large touchscreen devices.

In addition to the previous sensors used in the controller implementation, other libraries have been developed to use a barometric pressure sensor, a temperature and humidity sensor, an accelerometer, gyroscope, and magnetometer. Currently, these sensors can be read using a Beagle Bone and stored off to a log file. Work with these sensors is being done to create a low cost IMU for the microcontrollers on the drone using these sensors.

IMPACT/APPLICATIONS

We believe that the best chance of success for long duration and computationally intensive missions is by developing a heterogeneous system with a suite of processors spanning the power consumption spectrum. The expectation is that many tasks are not computationally intensive most of the time, but can peak dramatically from time to time. It would be advantageous to power a relatively small processor to handle only what must be processed in real time. When a burst of demand occurs, the system could power-on the more expensive and powerful processor to handle the increased demand, powering it down when complete. Furthermore, in the case where sensor data is collected, but real-time response isn't required and if the throughput is somewhat less than the capability of the processor, it would be better to keep the processor in a low power state most of the time and process backlogged data in a burst, rather than as a trickle.

RELATED PROJECTS

None