Planned Direct Transfers: A Programming Model for Real-time Applications

Gérard Vichniac, Barry Isenstein, Craig Lund, Arlan Pool
Mercury Computer Systems, Inc.

Abstract

This paper presents the “Planned Direct Transfer” programming model, developed by Mercury Computer Systems to meet the requirements of embedded high-performance computing applications. In this model, data transfers are “Planned” before they occur, resulting in low software overhead execution; they are also “Direct”—they do not require intermediate data copying. This paper locates the Planned Direct Transfer (PDT) model in the landscape of the standard approaches of Shared Memory and Message Passing.

1: PDT: Key Objectives and Assumptions

The Planned Direct Transfer (PDT) programming model strives to contain the software component of interprocessor communication management within the initialization phase of parallel programs, thus letting the hardware take care of the data movements during the main phase of the program execution. That way, all the computational and resource allocation overhead associated with communications can be incurred before the transfers actually take place.

The PDT model is not universal; it relies on two assumptions. First, PDT was developed and exercised on a specific class of applications, those requiring high-performance large-scale real-time computing. These real-time applications (RTAs) typically involve embedded sensor-based signal or image processing, and are exemplified by Synthetic Aperture Radar (SAR), Space-Time Adaptive Processing (STAP), Adaptive Beamforming, and security and medical imaging. Secondly, like all programming models, PDT makes specific assumptions on the architecture and hardware it runs on. In particular, PDT assumes high-bandwidth interconnect fabric as well as low-latency direct read and write capability between distributed memory modules, with no intermediate data copying. The PDT model was developed by Mercury Computer Systems on the RACE® architecture [1,2].

2: Specific Demands of Real-time Applications

RTAs are characterized by demands that are distinct from those of traditional scientific applications. Some of these demands are more exacting, some are more lenient. The more exacting demands require highly tuned resources and handles for optional programmer low-level fine tuning. The relative leniency of some RTA’s demands explain how PDT can exceed the performance of most supercomputers at a fraction of their cost.

2.1: Real-time Applications’ Exacting Demands

2.1.1: Real-time (Uninterruptible Sensors and Actuators)

RTAs involve a flow of data constantly entering the system from the external world via sensors, to be processed in real time and delivered to actuators, displays, or further processing. The hardware latencies (e.g., in routing data) and the software latencies (e.g., in servicing interrupt routines and in synchronizing threads) must not only be low, but also deterministic to avoid data overruns from sensors that do not have the option of turning off. The inherent urgency associated with real-time applications stems not only from sensors, but also from actuators. For example, a display requires the value of the next pixel be delivered within 30 microseconds.

2.1.2: High Utilization and Performance (Volume, Power Constraints)

The embedded nature of real-time applications (e.g., radar processing on an aircraft or image reconstruction on an ultrasound pushcart) places extraordinary demands on hardware utilization. To increase performance, one must achieve higher utilization, because there is simply no room or power for additional hardware. This particular requirement naturally leads to PDT’s demands for minimal software overhead and for handles for optional fine tuning.
2.1.3: Support for Heterogeneity

Embedded real-time multicomputing applications are best served by a heterogeneous mix of processor (DSP, RISC, and specialized) and I/O types, integrated in an architecture modular enough to accommodate changes in system design and advances in device technology [3]. Indeed, the computing involved in an entire DSP application is seldom “purely DSP.” Together with typical DSP vector tasks, such as image reconstruction, signal filtering, and signal classification, a whole DSP application generally includes scalar tasks, such as feature extraction, inversion problems, interpretation, global understanding, and decision. For these tasks, which are more scalar than vector in nature since they typically involve conditionals, table look-ups, and jumps, RISC processors are a better fit than DSP chips. [4]

The PDT programming model has been implemented on Mercury’s RACE architecture, characterized by:

- crossbar-based interconnect fabric
- adaptive, preemptable, source-based circuit switch routing
- link list DMA controllers with single-cycle start at every end point
- heterogeneity

2.2 Real-time Applications’ Lenient Demands

2.2.1: “Passive” (vs. “Control”) Data, and Separability between “Setup” and “Go” Phases

The confinement of data movement management within the initialization stage of a program is a key tenet of the PDT programming model. The quasi elimination of software overhead during actual data transfers is possible thanks to a property of RTAs. Sensor-based applications are typically separable between a “Setup” and a “Go” phase. In turn, this separability results from the “passive” nature of the data in RTAs.

During the “Go” phase (embodied in a routine inner loop), fresh data is continuously fetched from sensors, processed, and delivered to displays or to further downstream processing. While the sensors can deliver data at very high rates, the processing requirements of each chunk of data are typically simple, repetitive, and most importantly, predictable. From the programmer’s viewpoint, the time spent in an inner loop iteration is predictable and it is the same for all trips through the loop. The data obtained from the sensors is in general “Passive” (as opposed to “Control.”) The system takes a planned action (e.g., it executes a fixed sequence of signal processing transformations such as filters and FFTs) on the data, regardless of its runtime value. The data’s constant renewal, its passive nature, and the uniformity of the processing characterize RTAs and contrast them with scientific applications. In scientific applications, which are often expressed mathematically as initial value problems, the data is given only once (it is the initial value) and is evolved according to a differential equation expressing a scientific model. Each evolution step requires varied and complex operations, and is susceptible to changes with respect to the previous step, because the runtime value of the data can affect the control flow of the program execution via conditionals, e.g., in checking convergence criteria, or in periodic recalculations of the discretization step sizes.

During a RTA “Setup” phase, which in most cases occurs only once, the software prepares the stage and then stays out of the way, letting the hardware manage the data movements during the Go phase. The Setup section creates shared memory buffers, hires processors, partitions tasks, and invokes and initializes parallel libraries. Thanks to the regularity of processing characteristics of RTAs, the data flow patterns are ascertainable throughout the execution and thus can be entirely planned during the Setup phase. RTAs, in contrast with many scientific applications, do not require dynamic task partitioning and scheduling.

The resource management is done using named resources, under “positionless programming.” Knowledge of the actual hardware resources and of their relative positions is acquired by reading a configuration file. This file, which can vary from execution to execution of the same executable, contains the inventory of resources and their topology, defines routes and, when appropriate, associates individual paths with priority rankings. Typically, paths servicing sensors have highest priority—again, sensors are uninterruptible. Reading a configuration file at runtime allows the use of a single executable for a different number of processors and different topologies. This level of abstraction, which hides the actual hardware configuration from the code, is in line with the separation, inherent to PDT between endpoints and transfer plans, which are described below.

2.2.2: Dedicated Applications

The embedded nature of applications allows the economy of multi-user operating system overhead. With only one user (actually, only one application) on a system, one need not worry about job priority arbitration, security, and memory protection. In an airplane or in a medical device, your application is not on the Internet!

Note finally that while RTAs enjoy more lenient demands than many scientific applications, they are by no means embarrassingly simple: data transposes (or
that require "maximum" interprocessor communications are pervasive in multidimensional signal and image processing.

3: Anatomy of the PDT Programming Model

It is useful to view PDT as made of three building blocks: processes, endpoints, and transfer plans.

3.1: Processes

These are the same as in the Shared Memory (SM) and Message Passing (MP) programming models. For example RACE multicomputers implement POSIX threads and processes.

3.2: Endpoints

These are source and destination terminations for transfers. Examples of endpoints are:
- arrays in local memory
- shared-memory buffers that are globally visible to all processes
- memory and FIFOs in the interfaces of I/O devices and specialized processors
- foreign bus address spaces

The most liberating idea of endpoints is that sources and destinations are not limited to an intelligent processor's memory nor bound to any specific process, as they are in the standard message passing approach. PDT acknowledges the fact that there is more to data movement than interprocessor communication. One important consequence is that endpoints elevate hardware interfaces into "primary objects" which can be directly manipulated. Furthermore, data distribution can be independent of process distribution; this is important since most specialized processors, I/O devices, and bulk memory are not capable of supporting processes.

3.3: Transfer Plans

These specify how data is moved between endpoints. A transfer plan determines the endpoints, the engine (e.g., CPU or DMA), and optional transfer policies (e.g., fabric routing policies). A rich set of user control features for real-time applications becomes possible once transfer plans become the subject of manipulation.

The transfer plan is a flexible way to migrate an application from a high-overhead model, to one with fewer runtime inefficiencies. As mentioned above, to reduce runtime overhead, the “Setup” phase of a transfer can be separated from transfer execution in typical RTAs. A programmer can create transfer plans in an outer loop, and then inner loop execution of any transfer plan is initiated with a single-word write operation.

In the RACE architecture, the transfer plan selects either the CPU or an independent DMA controller as the transfer engine. The advantage of using a DMA engine over the CPU is that it is possible to overlap processing with I/O. CPU action has the advantage of immediate lower latency transfers, particularly useful for short messages. While the RACE environment offers reasonable defaults, application subtleties often interact in unpredictable ways and the ability to override the defaults and experiment with CPU versus DMA data movements without rewriting the application has proven to be very useful.

DMA engines abound in RACE hardware. Each compute environment (CE) has at least one DMA engine associated with it. In addition, I/O devices, memory-only nodes, and specialized processor interfaces have their own DMA engines. The transfer plan facility can command any DMA engine in the system. So for example, a process running on a CE can effect a transfer using a DMA engine on a "distant" I/O device.

Transfer plans allow the interconnection fabric properties to be readily expressed and controlled. In the RACE architecture, path routes and priorities are selectable on a per-transfer basis. Path routes select between multiple paths or adaptive routing between endpoints. Priorities control real-time I/O determinism. Routes and paths can be fixed in runtime code, but more importantly overridden by configuration tools. With configuration tools manipulating the routes and priorities of transfer plans, complex applications are quickly tuned without recompilation.

Another benefit of transfer plans is that broadcast and multicast do not require special treatment. Broadcast and multicast use the same transfer plans, but with broadcast or multicast endpoints.

Separating endpoints from transfer plans provide for explicit scheduling, routing, and allocation of bandwidth resources (i.e., paths, CPUs, and DMA engines). Flexible assignment of bandwidth resources allows for the avoidance of bandwidth contention necessary for sensor I/O and deterministic environments.

4: Comparison with Standard Parallel Programming Models

4.1: Comparison of the PDT Model to Shared Memory

As in the Shared-Memory model, defining a region of physical memory as a shared-memory buffer, makes it
directly accessible to all processors. With PDT, programmers can dynamically set up an arbitrary number of globally shared memory buffers, even though RACE memory is physically distributed. Data in the shared-memory buffers is directly read or written by any CPU or DMA engine.

PDT shared-memory buffers differ from SM in two important ways: exposed transfer plans and coherency. In SM, the transfer plan, i.e., how the data is moved, is hidden from the user. Hidden transfer plans are good for general purpose programming but unacceptable for demanding RTAs. Implicit transfer plans in SM do not allow for the coordination of hardware resources that is critical in achieving the typical utilization required by sensor-based computing.

Coherency concerns in SM occur when an SM implementation deviates from its purest form. In its purest implementation (e.g., in Cray supercomputers and in IBM 3090 mainframes) the SM model is implemented with a physical central memory accessed by cacheless processors. Under this form, the SM model is both elegant and effective. It is, however, not scalable beyond a few dozen processors because of saturation of the central memory bandwidth. Any departure from the original implementation, e.g., using caches or physically distributed memory modules inevitably leads to the perennial cache coherence problem. This problem arises any time an address is realized in more than one physical location in the system. The RACE architecture addresses the problem by disallowing the caching of addresses in several nodes, reducing the problem to that of maintaining I/O coherency. Maintaining I/O coherency requires only snooping buses local to a given node, as opposed to snooping over the entire interconnect (and impacting utilization), as required by maintaining full cache coherency.

4.2: Comparison of the PDT Model to MP

A message is really a copy of application data from a processes' local address space (plus message information) to some address determined by the receiving process. In most MP implementations, the concept of messages is a fixed transfer plan where the buffers used are "hidden." That is, messages are formulated and sent from process to process; the actual buffers used and how the data is transferred are removed from the user's concern. Also, the software that directs message formulation and resolution of the implied transfer plan occurs at execution time for all messages, allowing for more dynamic scheduling, but at the cost of higher runtime overhead. Finally, in the send-receive paradigm, data transfer can be viewed as "legal only between consenting processors and in private." In contrast, under PDT, a transfer engine (CPU or DMA) can read to and write from a remote endpoint "without its consent," and in "public view" of other processors when the address subspace involved is declared a shared buffer. These one-sided transfers supported by PDT under RACE are quite similar to the put and get functions as implemented in the Cray T3E machine.

5: Summary

The key ideas of the Planned Direct Transfer programming model are:
- minimization of software overhead associated with data movements by planning transfers in advance
- elimination of intermediate data copying by making the transfers direct
- allowance for optional user control to maximize performance within fixed volume and power constraints
- allowance of non-intelligent compute and I/O devices to participate directly in the programming model

Since 1994, PDT has been deployed over 10,000 processors under Mercury's RACE architecture. It has proven be to meet the requirements of a variety of high-performance realtime applications.

References


URL: http://www.mc.com