Adaptive Data Placement in an Embedded Multiprocessor Thread Library

Phillip Stanley-Marbell
Dept of ECE, Carnegie Mellon University
Pittsburgh, PA
pstanley@ece.cmu.edu

Kanishka Lahiri
NEC Laboratories America
Princeton, NJ
klahiri@nec-labs.com

Anand Raghunathan
NEC Laboratories America
Princeton, NJ
anand@nec-labs.com

***Abstract*— Embedded multiprocessors pose new challenges in the design and implementation of embedded software. This has led to the need for programming interfaces that expose the capabilities of the underlying hardware. In addition, for systems that implement applications consisting of multiple concurrent threads of computation, the optimized management of inter-thread communication is crucial for realizing high-performance.**

This paper presents the design of an application-adaptive thread library that conforms to the IEEE POSIX 1003.1c threading standard (Pthreads). The library adapts the placement of both explicitly marked application data objects, as well as implicitly created data objects, in a physically distributed on-chip memory architecture, based on the application's data access characteristics.

I. Introduction

Advances in semiconductor process technology are enabling integrated circuits consisting of multiple, loosely-coupled, programmable devices, integrated along with a variety of other components (e.g., on-chip memories, peripherals, domain-specific processors), to implement a specific application, or set of applications [1]. One challenge in the use of such embedded multiprocessors is the dearth of appropriate programming interfaces for effectively exploiting hardware-level concurrency. Another emerging challenge is the increasing performance-cost associated with on-chip data transfers, due to the combination of rapidly escalating system complexity and poor scaling trends associated with chip-level interconnects [2]. As a result, for multi-threaded applications mapped to embedded multiprocessors, optimizing inter-thread communication is a critical requirement for achieving high performance. In this paper, we illustrate the role of an *application-adaptive* thread library in addressing these challenges. We provide an overview of an IEEE POSIX 1003.1c [3] compliant thread library that we designed, highlighting the optimizations that were built in for efficient management of shared data objects in an embedded multiprocessor.

II. Illustrative Example

In this section we consider a multi-threaded implementation of the IEEE 802.11 wireless LAN MAC layer (Figure 1). For illustration, we focus on the sub-system enclosed by the dashed box, which contains the WEP (Wired Equivalent Privacy) encryption and the CRC (Cyclic Redundancy Check) tasks. As shown in the figure, each task is implemented as a thread that is mapped to a different processor. The memory architecture consists of a distributed set of memories that are

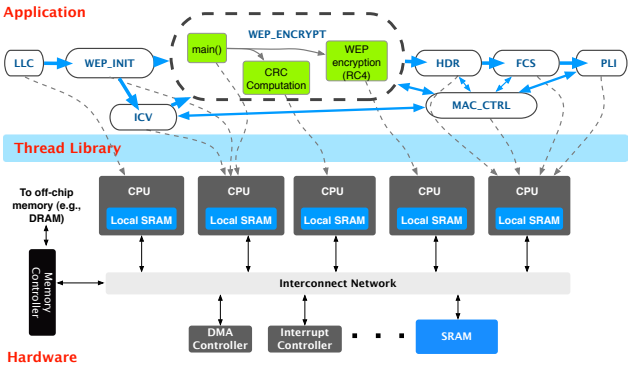

Fig. 1. A multi-threaded implementation of the IEEE 802.11 wireless LAN MAC protocol, and it's mapping to a multiprocessor based architecture.

local to each processor, as well as a globally shared memory. The threads comprising the illustrated sub-system share different data structures that may potentially be mapped to any of the three processor-local memories within the considered sub-system. We identified 36 data structures spanning two categories: *programmer-specified* data structures, such as CRC tables, encryption keys, and network packets, and *library-specific* data structures that are created and maintained by the thread library to facilitate communication and synchronization between threads (e.g., mutexes, condition variables).

The application was simulated with actual MAC layer data captured off a real wireless LAN. Based on an analysis of the complete input trace, we calculated the best mapping of the shared structures to the three processor-local memories, over 100 ms intervals. We observed that the optimum mapping *varied over time*, with significant "phase" behavior for many mappings (Figure 2). This motivates run-time adaptation of the mapping of shared data structures to different physical locations within the distributed memory architecture. In addition, to the extent possible, these optimizations should be transparent to the application developer.

III. Thread Library Overview

The proposed thread library implements the IEEE POSIX 1003.1c threading (Pthreads) interface, and takes advantage of inter-thread communication characteristics to intelligently manage the mapping of virtual addresses to physical locations within a distributed on-chip memory architecture. For applications with time-varying characteristics, the library is capable

3-9810801-0-6/DATE06 © 2006 EDAA

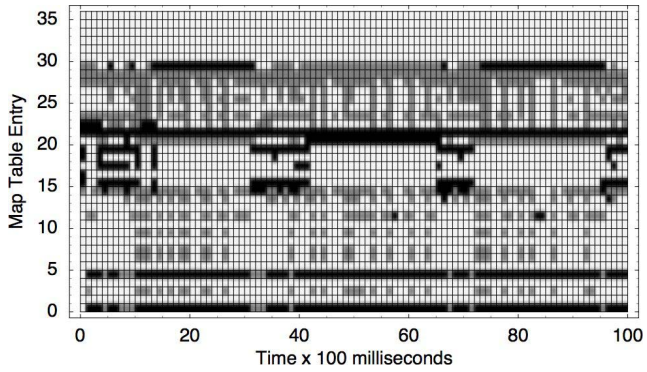

Fig. 2. Optimum mapping of each of 36 data structures in the multi-threaded implementation of WEP, over time, to one of three processor-local memories. (Black: processor 2, dark gray: processor 1, light gray: processor 0)

of performing these optimizations at run time, using minimal additional hardware support.

A. Application-Specific Data Management

To facilitate programmer-directed mapping, the library extends the Pthreads interface with a facility to enable programs to register variables or data structures with the library, either relinquishing control of where (which physical memory location) it is to be allocated, or explicitly specifying in which processor's local memory the datum should reside. The interface and its use are illustrated in Figure 3.

Interface:

```
pthread_register(addr_t b, size_t s, char *t, int where);
```

Example: Register datum **buffer** with library, requesting placement in local memory of processor 0

```
pthread_register(&buffer, sizeof(buffer), "buffer", 0);
```

Example: Register datum **buffer** with library, relinquishing control of placement to library

```
pthread_register(&somestruct, sizeof(somestruct), "", -1);
```

Fig. 3. An extension to the Pthreads interface.

The facilitator for both static and dynamic optimizations is a structure referred to as the *map table*. The map table contains entries for all data structures which are either explicitly registered with the library by a programmer (using the `pthread_register` interface), or implicitly so. Implicit registration in the map table occurs when, as a result of standard Pthreads interface functions, the library allocates a data structure that is visible within multiple threads. Each map table entry maintains information about the mapping to physical local memory for a data structure. Static optimizations involve generating this map table once. Dynamic optimizations on the other hand, involve updating this map table at run time.

B. Hardware Support

Dynamic map table updates require a means of tracking the number of accesses to fine-grained memory ranges (e.g., a small data structure in memory), and being able to monitor the accesses to these memory ranges across multiple processors. This is achieved by means of a hardware device called a *Map Table Support Peripheral (MTSP)* (Figure 4). When a thread executes a `pthread_register` library call, the library creates an entry for the registered structure in the map table, and also creates a corresponding entry in the MTSP. MTSP updates occur on a per-processor basis each time a processor generates an address on the system bus that matches an address range associated with an already registered object. A dynamic re-mapping mechanism (details omitted for brevity) that re-maps data structures to optimized physical memory locations is triggered every time a certain window of time goes by. The window size is a parameter of the library implementation, and can be appropriately specified to trade-off mapping optimality for re-mapping overhead.

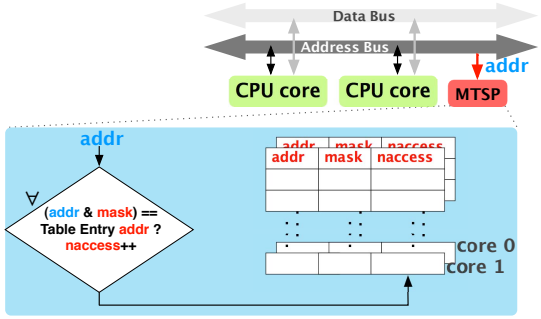

Fig. 4. Architecture of a *Map Table Support Peripheral (MTSP)*.

IV. Evaluation

To evaluate the performance of our thread library, we used cycle-accurate simulation of a multi-processor system based on the Hitachi SH embedded processor [4]. The system includes the MTSP hardware, and timers that generate interrupts for triggering dynamic map table updates. The thread library was implemented using a combination of C and assembly language for the target hardware. The benchmarks used were the IEEE 802.11 MAC Layer sub-system described in Section II, and a multi-threaded implementation of a Software Radio (SDR) application, both of which were implemented using the standard Pthreads interface.

In the case of the SDR application, the maximum sustainable throughput was improved by 60% over a realistic worst case, and 40% over a typical memory object placement. For the IEEE 802.11 MAC Layer application, which exhibits time-varying characteristics, the dynamic application-adaptive techniques provided a further 6–8% improvement in maximum throughput over static optimization approaches.

References

[1] A. Jerraya and W. Wolf, *Multiprocessor System-on-Chips*. Morgan Kaufmann, 2004.

[2] R. Ho, K. W. Mai, and M. A. Horowitz, "The Future of Wires," *Proc. IEEE*, vol. 89, pp. 490–504, Apr. 2001.

[3] The Institute of Electrical and Electronics Engineers (IEEE), "Chapter 16: Thread Management," in *Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language]*, July 1996.

[4] Atsushi Hasegawa and Ikuya Kawasaki and Kouji Yamada and Shinichi Yoshioka and Shumpei Kawasaki and Prasenjit Biswas, "SH3: High Code Density, Low Power," *IEEE Micro*, vol. 15, pp. 11–19, Dec. 1995.