

Development of a Fork-Join Dynamic Scheduling Middle-Layer for Automotive Powertrain Control Software

Waseem Sadeh and Osamah Rawashdeh
 Oakland University

Dona Burkard, Kelvin Dobbins, and Tony Lockwood
 Ford Motor Company

Atila Bulmus
 Infineon Technologies North America Corp.

ABSTRACT

Multicore microcontrollers are rapidly making their way into the automotive industry. We have adopted the Cilk approach (MIT 1994) to develop a pure ANSI C Fork-Join dynamic scheduling runtime middle-layer with a work-stealing scheduler targeted for automotive multicore embedded systems. This middle-layer could be running on top of any AUTOSAR compliant multicore RTOS. We recently have successfully integrated our runtime layer into parts of legacy Ford powertrain software at Ford Motor Company. We have used the 3-core AURIX multicore chip from Infineon and the multicore RTA-OS. For testing purposes, we have forked some parallelizable functions inside two periodic tasks in Ford legacy powertrain software to be dynamically scheduled and executed on the available cores. Our preliminary evaluation showed 1.3–1.4x speedups for these two forked tasks. It also showed that this runtime layer scales well to the available cores and that it abstracts away the details involved in load balancing and inter-core communications from programmers. This paper outlines our preliminary design and results of evaluating the dynamic scheduler approach for powertrain control software on multicore chips. This work is ongoing research and this paper presents challenges such as meeting deadlines, reliability, and safety that arise when adopting dynamic scheduling for such time and safety-critical software.

CITATION: Sadeh, W., Rawashdeh, O., Burkard, D., Dobbins, K. et al., "Development of a Fork-Join Dynamic Scheduling Middle-Layer for Automotive Powertrain Control Software," *SAE Int. J. Passeng. Cars – Electron. Electr. Syst.* 10(2):2017, doi:10.4271/2017-01-1620.

1. INTRODUCTION

Increasing the processor's clock frequency by hardware engineers was a key factor for delivering high performance microprocessor or microcontroller chips. Hardware engineers also used other important techniques to increase the number of executed instructions per clock cycle such as adding pipeline stages, out-of-order execution, branch predictions, caches, and speculative execution. Software engineers enjoyed this seemingly free performance gains for their applications with minimal changes in their source code or even with no changes at all [1].

Hardware engineers reached a point where they cannot deliver new processors with faster clock frequencies and acceptable power consumptions. They started with the new path of delivering new processors with more cores. The embedded systems industry started to move also toward the multicore microcontroller chips. The automotive industry in particular started to use microcontroller chips with three cores, and more cores are expected to arrive [2]. Simply running existing software applications which have been designed for single cores on the new multicore processors will not achieve performance gains. Software engineers should consider new ways to

design and migrate their applications for efficient utilization and for achieving high performance applications on multicore chips. Furthermore, migrating legacy applications (which were designed to run on single core chips) into multicore chips requires significant changes in their design to port them efficiently into the multicore chips. Exploiting parallel processing in the multicore processors is a key factor to decrease execution time and power consumptions. Design for performance is important goal for automotive industry in the presence of multicore processors. But software engineers should also utilize the available cores for many other important goals. For example, making redundant standby software components to be running in different cores will be a good design to achieve reliable and more fault-tolerant applications. The increased performance and the available number of cores in the future processors could also bring many benefits to the embedded systems. For instance, design techniques for building highly concurrent and parallel applications that have been used by the desktop and server computing could be leveraged and evaluated by the embedded systems world. This is the approach we have taken in this ongoing research. We have started this research with goal of evaluating the dynamic scheduling approach for building automotive controls software targeted for multicore

microcontrollers. Dynamic scheduling allows unit of computations to be executed on different cores each time they run. For example, when a unit of computation is about to be executed, intelligent decisions could be made at runtime to hand over this unit of computation to the available idle core to achieve effective load balancing. We have adopted the approach used by **Cilk** [3] to build a Fork-Join runtime middle-layer for automotive powertrain controls software running on multicore processors. This runtime layer is a pure **ANSI C** library and therefore no language or operating system changes are required. Ideally, dynamic scheduling could be a component integrated in the multicore real-time operating system (RTOS) but for our preliminary work we have taken the approach of making this runtime layer running on top of any available **AUTOSAR** compliant multicore RTOS. Making this runtime layer as a pure **ANSI C** library will play a major role in our goal of this research, which is to evaluate the dynamic scheduling approach. It can be easily ported into different **AUTOSAR** RTOSs and into different multicore processors. This paper is divided into six sections. [Section 2](#) provides an overview of the **Cilk** runtime system. [Section 3](#) describes our system design in detail. [Section 4](#) gives our initial evaluations and results. [Section 5](#) gives lessons learned, challenges and future work. Finally, [Section 6](#) concludes our paper.

2. RELATED WORK

Cilk is a language and runtime system that provides a dynamic concurrency platform for shared memory multiprocessor architectures [3]. In **Cilk**, programmers express the existing parallelism in their software regardless of the available number of processors and the runtime abstracts away all the details of load balancing and communication among processors. At runtime, the **Cilk** runtime system will map computations into the available processors using “work-stealing” scheduling algorithm. This dynamic concurrency model has many benefits. First, the same application scales well to the available number of processors without any change. When a new hardware platform becomes available with more processors, the same **Cilk** application would run again on the new hardware and application will gain efficient execution with no change. Second, the programmers will focus on expressing the parallelism in their applications. The runtime will handle all load balancing and communications among processors.

Cilk extended the **C** language by adding three keywords for parallelism and synchronization: **cilk**, **spawn**, and **sync**. [Fig. 1](#) lists a **Cilk** program that calculates the *n*th Fibonacci number.

The **cilk** keyword as used in Line 1 identifies that the **fib** function could be *spawn*. The **spawn** keyword as used in Line 5 allows **fib(n-1)** call to be executed on different processor. The work-stealing scheduler decides *at runtime* where this computation should run to achieve best load balancing. It could end up running on the same caller processor or mapped into other idle processor. The **sync** keyword as used in Line 7 is a synchronization point where the flow of control should not continue until all spawned work finish its execution. The work-stealing algorithm is a provably good scheduler [3].

```
#include <headers.h>
1  cilk int fib ( int n){
2      if( n < 2)
3          return n;
4      else{
5          int x = spawn fib (n-1)
6          int y = fib (n-2)
7          sync;
8          return x + y;
9      }
10 }
11 int main( ){
12     int n = .... ;
13     int results spawn fib(n);
14     sync;
15     printf(“Fib is %d\n”, results);
16     return 0;
17 }
```

Fig. 1. Cilk program that calculates the *n*th Fibonacci number

In the next section, we explain how we adopted the **Cilk** approach to design our Fork-Join runtime layer for multicore embedded systems.

3. FORK-JOIN DYNAMIC SCHEDULING RUNTIME LAYER DESIGN

Our first design decision on implementing our dynamic scheduling runtime system was whether to integrate it as a part of a multicore real time operating system or to make it as a layer on top of multicore RTOS. We made the decision to make it a layer that runs on top of any available multicore RTOS. This turns out to be a good decision because it will allow us to easily port our runtime layer into any multicore RTOS environment, which targets different multicore microcontroller chips. This will be important for our goal of this ongoing research for evaluating the dynamic scheduling for multicore embedded systems. Automotive industry is moving towards the **AUTOSAR** standard. So, we aimed for **AUTOSAR** multicore real time operating systems. Although the runtime layer sits on top of the **AUTOSAR** OS, the programmer could bypass it all together and just use the OS API. In other words, the runtime layer does not hide the operating system API from the programmer.

Cilk made an extension to the **C** language by adding keywords to the language. This approach requires either a front-end compiler that understands these keywords or at least a complete preprocessing step before invoking the **C** compiler. We made the second design decision of making the Fork-Join dynamic scheduling runtime layer as a pure **ANSI C** API library that could be linked into any **C** application. Effectively, no **C** compiler changes are required. And because this runtime layer runs on top of **AUTOSAR** OS, no changes are required for the RTOSs. [Fig. 2](#) shows the software layers on top of multicore microcontroller chip

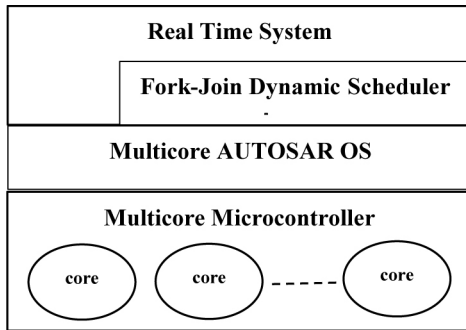


Fig. 2. Fork-Join Dynamic Scheduling Runtime Layer

AUTOSAR OS supports basic tasks and extended tasks. A basic task has 3 states: *Suspended*, *Ready*, and *Running*. A basic task has a simple model: starts, runs, and finally terminates. During its running, it could be preempted by higher priority tasks if configured as fully preemptive basic task. Once terminated, it could be rescheduled to run again by making a request to the OS to activate it again.

Extended task extends the basic task by providing another state: *Waiting*. A running extended task could enter the *Waiting* state to wait for some event. Once that event is signaled, the extended task wakes up again and becomes ready. The **AUTOSAR OS** provides APIs to set events (signal them) and to wait for them to be signaled. This model is important, for example, to design producer-consumer applications, where one extended task could produce some items and other extended tasks could consume them. If the consumer extended task has no items to consume or process, then it would go to the *Waiting* state to save CPU cycles. Extended tasks are key component in our runtime layer.

We will configure the system to have some cores running only basic tasks and the other available cores to run only extended tasks. If one core is running some basic tasks, then these basic tasks will preempt each other on the running core according to their defined priorities. But on the other hand, an extended task will be configured to run all the time on its dedicated core and no other tasks will run on this dedicated core. At run time, any basic task could fork some work computation to be executed on other cores. One available extended task will execute that forked unit of work on its dedicated core. The units of works are mapped into cores using the work-stealing algorithm which will be described later. If there is no work to do, each extended task will enter the *Waiting* state. The decision of having one extended task and no other tasks running on an available core will simplify our layer design at this early stage. For instance, the extended tasks will be always running and would not be preempted by other higher priority tasks on the same core. This will allow extended tasks to execute the forked works as quickly as possible. This design decision assumes that we will have many cores in future that would compensate for reserving some cores for such extended tasks.

The dynamic scheduling runtime layer provides the following Fork-Join (FJ) API functions: **FJ_init**, **FJ_Take**, **FJ_Fork**, and **FJ_Join**.

FJ_init() will initialize the runtime layer. It will associate extended tasks (threads) for each core. Each core will have its own priority queue where the unit of works can be buffered to be executed. Fig. 3 shows the runtime layer as initialized.

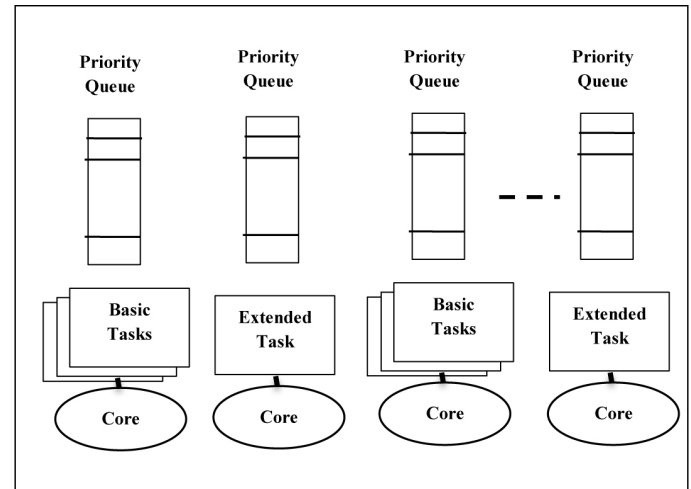


Fig. 3. Runtime Layer components

If a basic or extended task forked some work, then that work will be queued on the queue dedicated for the core that is running this forking task. A forked work will be queued according to its priority which is the priority of its forking task. This will allow higher priority forking tasks to get their forked work finished before lower priority tasks. Any idle extended task will “steal” work from the available queues including the queues allocated for other extended tasks. A forked work, when executed by extended task, could also fork other works which will be queued on the queue allocated for this core. This “work-stealing” scheduling algorithm will achieve automatic load balancing across the cores.

Fig. 4 lists partial C code of an extended task running on a dedicated core. The code uses macros and API defined by the multicore RTA-OS.

```

1  TASK(ExtendedTask) {
2  //initializations
3  int thisCoreId = GetCoreID();
4  while(1) {
5      do{
6          Work_t* w = FJ_Take (thisCoreId);
7          if ( w != NULL){
8              w -> runnable( );
9              w -> done =1;
10         }
11         }while( w != NULL );
12         WaitEvent( WorkExistsEvent );
13         ClearEvent( WorkExistsEvent );
14     }
15 }

```

Fig. 4. Extended Task Running on Dedicated Core

