# Compiler-Aided Run-Time Performance Speed-Up in Super-Scalar Processor

***Moshe Pelleh***
***Holon Institute of Technology, Rehovot, Israel***

**mpelleh@yahoo.com**

## Abstract

In our world, where most systems become embedded systems, the approach of designing embedded systems is still frequently similar to the approach of designing organic systems (or not embedded systems). An organic system, like a personal computer or a work station, must be able to run any task submitted to it at any time (with certain constrains depending on the machine). Consequently, it must have a sophisticated general purpose Operating System (OS) to schedule, dispatch, maintain and monitor the tasks and assist them in special cases (particularly communication and synchronization between them and with external devices). These OSs require an overhead on the memory, on the cache and on the run time. Moreover, generally they are task oriented rather than machine oriented; therefore the processor's throughput is penalized.

On the other hand, an embedded system, like an Anti-lock Braking System (ABS), executes always the same software application. Frequently it is a small or medium size system, or made up of several such systems. Many small or medium size embedded systems, with limited number of tasks, can be scheduled by our proposed hardware architecture, based on the Motorola 500MHz MPC7410 processor, enhancing its throughput and avoiding the software OS overhead, complexity, maintenance and price. Encouraged by our experimental results, we shall develop a compiler to assist our method. In the meantime we will present here our proposal and the experimental results.

**Keywords**: Computer architecture, Operating systems, Embedded systems, Scheduling.

## Introduction

For the sake of better understanding our discussion let's remember some concepts.

Each software unit which is managed or executed by the operating system is called a job or an instance. The collection of related jobs (or instances) cooperating to execute a function is called a task. The tasks in the system can be presented by models. The model in which the jobs (instances) of a task are executed periodically, with a fixed time period, on a regular and continues basis in order to execute a function (or an application) is called periodic task. When it is needed to react to external random events, we use aperiodic tasks or sporadic tasks. The release time of their instances is random and can not be predicted. One way to execute correctly (on time) jobs in the system, is to assign each job a priority according to its period or its importance or its criticality, and then let the scheduler (of the operat-

Material published as part of this publication, either on-line or in print, is copyrighted by the Informing Science Institute. Permission to make digital or paper copy of part or all of these works for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage AND that copies 1) bear this notice in full and 2) give the full citation on the first page. It is permissible to abstract these works so long as credit is given. To copy in all other cases or to republish or to post on a server or to redistribute to lists requires specific permission and payment of a fee. Contact Publisher@InformingScience.org to request redistribution permission.

ing system), schedule them for execution based on their priorities (Liu, 2000).

A thread, sometimes called Light Weight Task (LWT), is a basic unit of CPU utilization. It comprises a Program Counter (PC), a register set, and a stack. It shares with other threads belonging to the same task its code section, data section, and other OS resources. A traditional task has a single thread of control. If a task has multiple threads of control, it can perform more than one mission at a time. By default, threads share the memory and the resources of the task to which they belong. The benefit of sharing code and data is that it allows a task to have several different threads of activity within the same address space. The benefit of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors. Since threads share resources of the task to which they belong, it is much more economical to create and context-switch threads (Silberschatz, 2005).

Switching the CPU from the running task to another task requires performing a state save of the current task and a state restore of a different task. This operation is known as a context switch. The context includes the CPU registers, the task state, and memory management information. The context switch is generally a pure heavy overhead.

An embedded system is a software system that is completely encapsulated by the hardware that it controls (Laplant, 1997).

HTT (Hyper-Threading Technology) works by duplicating certain sections of the processor—those that store the *architectural state*—but not duplicating the main execution resources. This allows a Hyper-Threading equipped processor to pretend to be two "logical" processors to the host operating system, allowing the operating system to schedule two threads or tasks simultaneously. Practical restrictions on chip complexity have limited the number to two logical processors for most implementations.

A barrel processor is a CPU that switches between threads of execution on every cycle. It generally does not allow execution of multiple instructions in one cycle. For example, certain CDC Cyber computers executed one instruction from each of 20 different threads before returning to the first thread. Each thread of execution is assigned its own program counter and other hardware registers (each thread's architectural state).

The SPARC processor usually contains as many as 128 *general purpose registers*. At any point in time, only 32 of them are immediately visible to software - 8 are global registers and the other 24 are from the *stack* of registers. These 24 registers form what is called a *register window*, and at function call/return, this window is moved up and down the register stack. Each window has 8 local registers and shares 8 registers with each of the adjacent windows. The shared registers are used for passing function parameters and returning values, and the local registers are used for retaining local values across function calls.

A superscalar processor executes more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to redundant functional units on the processor. Each functional unit is not a separate CPU core but an execution resource within a single CPU such as an arithmetic logic unit, a bit shifter, or a multiplier. A superscalar CPU is typically also pipelined. Instructions are issued from a sequential instruction stream. CPU hardware dynamically checks for data dependencies between instructions at run time.

## System Model

As we can see, all architectures are thread (or task) oriented by means of having a register set for each thread (or task).

Let's define any meaningful code instruction sequence as Ultra Light Task (ULT). A ULT has only a Program Counter (PC) and a Condition Code Register (CR). All its other resources are shared with other ULTs within the same task, which is the only single task in the system.

The Condition Code Register (CR) contains information about the last instruction's result, to be used by the following instruction (in the same thread) in case of a branch (or a jump) decision. Hence, each ULT must have its own CR.

The Motorola MPC7410 is a superscalar processor that can dispatch and complete two instructions simultaneously in each machine cycle, but has an average throughput of only one instruction per cycle.(RISC, 2000)

If we upgrade it to have 32 PCs, working in a round robin on the instruction level, 32 CRs, 256 General Purpose Registers (GPRs) and perhaps also 256 Floating Point Registers (FPRs) we shall be able to run up to 32 ULTs by the hardware, increasing throughput, and avoiding the use of an operating system. The penalty is 4KB of L1 cache dedicated to these registers. Alternatively, we can use a regular MPC7410 and spread our ULTs instructions to simulate an upgraded one. This is easily done: if we have 3 ULTs, 3 instructions each, a1 is the first instruction of ULT a and c3 is the third instruction of ULT c, we write the code a1,b1,c1,a2,b2,c2,a3,b3,c3 and so on, instead of writing it a1,a2,a3,b1,b2,b3,c1,c2,c3.

## First Series Experiments

We wrote a software application source code in C language, 40 C language instructions long, contained in a million times loop (see Appendix). We executed our experiments on Motorola MPC7410 superscalar processor, under the control of VxWorks 5.5 (Windriver, 2002), with clock rate of 60 ticks per second (an interrupt every 16.666 millisecond). We divided our experiments to 3 phases.

### *First Phase*

We compiled our source code and run it as 4 tasks, each with priority 200, synchronized to run sequentially by semaphores. 2 tasks were floating point intensive and the other 2 tasks were integer arithmetic intensive. This was the basic configuration.

### *Second Phase*

We attached the 4 task source codes to create a single task, in which the first instruction of the second task follows the last instruction of the first task, and so on. We compiled and run this resultant task. This configuration simulated execution without OS, or with a minimal OS intervention.

### *Third Phase*

We mixed the source code instruction stream of the resultant single task as described in paragraph 2 (System model) to get a new task, we compiled it and run it. This configuration simulated a round robin execution of 4 ULTs, without OS and without (or with reduced) dispatch unit stalls due to instruction data dependencies and without (or with reduced) cases in which two consecutive instructions require the same execution unit of the processor.

## Results of First Series Experiments

We observed that each C source code instruction was translated by the compiler to 7.5 assembly languge instructions in average. Hence, our 40 source code instructions were translated to 300

machine code instructions. Looping them million times, the processor executes 300 million machine instructions.

The first phase run time duration was 320 ticks, or 5333 millisecond. Considering 4 task context switching, each with 20 stores and 20 loads, we have (20+20)*4=160 extra machine instructions in every loop. Adding kernel queues management process, running 4 times every loop, and using 8 times a loop system calls to handle semaphores, these huge numbers for run time duration can be explained.

The second phase run time duration was 35 ticks, or 583 millisecond, which yields approximately 515 million instructions per second, or approximately 1 instruction per processor cycle, according to Motorola's declaration about average throughput.(RISC)

The third phase run time duration was 25 ticks, or 417 millisecond, which yields approximately 720 million instructions per second, or approximately 1.44 instruction per processor cycle, about 44% throughput increase.

## Second Series Experiments

Same environment and software application source code as in paragraph 3 (first series). We divided our experiments to 3 phases.

### *Fourth Phase*

Since we carried out the second series few months later, we repeated the second phase in order to verify that our environment and application respond as in the first series. The fourth phase run time duration was 35 ticks, as it was in the second phase.

### *Fifth Phase*

We observed that the output assembly code from the fourth phase was using repeatedly the same 3 or 4 registers for all the calculations along the application. So, there is a real risk that if we mix the *assembly code* instruction stream (as described in paragraph 2 "System model") we shall cause data collisions on the registers. For sake of simplicity we defined all our 17 variables as register (instead of volatile or static as it was before). In other words, we allocated a different register for each variable. Changing the source code caused a dramatic change in the assembly output code. Instead of 7.5 assembly instructions per a C instruction, we got here about 3 assembly instructions per a C instruction (because most of the load and store instructions were dropped). It means that this time our 40 source code instructions were translated to 120 machine code instructions. Looping them million times, the processor executes 120 million machine instructions. We compiled and run this assembly language resultant task. This configuration simulated execution without OS, or with a minimal OS intervention, for the version with variables defined as registers.

### *Sixth Phase*

We mixed the source code instruction stream of the resultant single task as described in paragraph 2 (System model) to get a new task, we compiled it and run it. This configuration simulated a round robin execution of 4 ULTs, without OS and without (or with reduced) dispatch unit stalls due to instruction data dependencies and without (or with reduced) cases in which two consecutive instructions require the same execution unit of the processor.

# Results of Second Series Experiments

The fifth phase run time duration was 13 ticks, or 217 milliseconds, which yields approximately 553 million instructions per second, or approximately 1.1 instructions per processor cycle, similar to Motorola's declaration about average throughput (RISC, 2000).

The sixth phase run time duration was 8 ticks, or 133 milliseconds, which yields approximately 900 million instructions per second, or approximately 1.8 instructions per processor cycle, about 64% throughput increase.

# Conclusion and Future Work

Performing our experiments on C language level yields about 40% throughput increase. Performing our experiments on assembly language level yields about 64% throughput increase. Assuming the need of about 8 general registers per task (or thread, or ULT), having an embedded (real time) application that requires not more than 4 tasks and not more than 2MB of (private) memory, it is worth considering our method using the Motorola MPC7410 superscalar processor. You will gain more than 40% (or 60%) processor throughput and you will save the OS price, OS complexity, OS maintenance, OS memory overhead and OS run time overhead. Critical sections, mail boxes and semaphores can be easily implemented in the C application code,(Liu, 2000, Silberschatz, 2005) avoiding the need of an OS. Same consideration for interrupt handler routines - you can use an interrupt task instead of an interrupt subroutine. Prefer global variables to avoid stack inconvenience, and if possible, register variables. When and if Motorola produces our proposed architecture, we shall be able to execute up to 32 ULTs simultaneously.

# Epilogue

Let us remember some facts to better appreciate the results.

## *Maximum Utilization in Multi-Tasking Systems*

***(Liu & Layland, 1973)***

A sufficient schedulability condition for any system of n independent preemptable tasks (that have relative deadlines equal to their respective periods) is that the schedulable utilization $U(n) \leq n(2^{1/n} - 1) \approx 0.7$.

In other words, using regular software engineering methods for building an application consisting of fixed priority tasks under a commercially existing real time operating system, we can use up to 70% of the processor throughput capability in most of the cases. Otherwise, we may miss deadlines. Hence, if our application requires 500 or 600 million instructions per second, we should couple two processors on a (VME) bus to run it without risk of deadline miss, while using the regular task approach under existing real time operating system. Using our ULT approach, we get 900 million instructions per second on the same processor, avoiding the need of buying, coupling and synchronizing two processors.

## *Utilization Bounds for Multi-Processor*

***(Oh & Baker, 1996)***

The schedulable utilization of the first fit assignment algorithm for a fixed priority system containing m processors is $U(m) = m(2^{1/2} - 1) = 0.414m$. If the total utilization of the periodic tasks exceeds $(m+1)(1+2^{1/(m+1)})$, the first fit algorithm may not find a feasible assignment on m processors. For m equal to 2, the lower bound 0.414m is about 60% of this upper bound, which is 0.414*2*100/60=1.38, and for large m, the lower bound is about 80% of the upper bound.

In other words, using 2 processors coupled together on a (VME) bus, for running fixed priority tasks under existing real time operating system, will add about mere 40% to the total throughput of the system.

## *My Experience*

About ten years ago, in Elta Electronic Industry, we had an embedded real time multi tasking project, running on a Motorola MC68040 processor (Motorola, 1993), under MTOS-UX real time operating system (MTOS, 1994). In order to enhance performance, we coupled together 2 MC68040 processors on a VME bus and run it under the same MTOS-UX real time operating system. Throughput enhancement was 42%.

# References


Laplant, P. A. (1997). *Real-time systems design and analysis* (2nd ed.). IEEE Computer Society Press.

Liu, C. L., & Layland, J. W. (1973). Scheduling algorithms for multiprogramming in hard real time environment. *Journal of the ACM, 20*, 46-61.

Liu, J. W. S. (2000). *Real Time Systems*. Prentice Hall.

Motorola. (1993). *M68040 Microprocessor User's Manual, Rev. 1*. Motorola Inc.

MTOS. (1994). *MTOS UX User's Manual, Version 3.1, I.P.I.* Jericho, NY.

Oh, D. I., & Baker T .P. (1996). *Utilization bounds for n-processor rate monotone scheduling with static processor assignment*. Technical Report, Department of Computer Science, Florida State University

RISC. (2000). *Motorola, MPC7410 RISC Microprocessor User's Manual, Rev. 0*.

Silberschatz, A. (2005). *Operating system concepts* (7th ed.). John Wiley & Sons.

Windriver. (2002, July 25). *VxWorks Programmer's Guide, Ver. 5.5*.


# Appendix

Our initial pseudo code used to develop our application. The final C code is less understandable and contains long statements which do not fit this double column format.

**4 tasks execution**

```
unsigned long start_time, end_time,
elapsed_time
long int counter
int n
float s, v, t, a
int u0, u1, u2, u3, u4, u5, u6, u7, u8, u9
float w

task1 {   /*  n!   */

start_time = TICKGET( )
counter = 0
goto lab2
lab1:
wait (semaphor4)
lab2:
n = 1
n = n*2
n = n*3
n = n*4
n = n*5
n = n*6
n = n*7
n = n*8
n = n*9
send (semaphor1)
goto lab1
}

task2 {   /*  s = s0 + v0t + 0.5at2   */
lab3:
wait (semaphor1)
s = 10.0
v = 10.0
a = 9.82
t = 100.0
v = v*t
a = a/2.0
a = a*t
a = a*t
s = s + a
s = s + v
send (semaphor2)
goto lab3
}

task3 {   /*  u(n + 1) = u(n) + u(n – 1)   */
lab4:
```

```
                wait (semaphor2)
                u0 = 0
                u1 = 1
                u2 = u0 + u1
                u3 = u1 + u2
                u4 = u2 + u3
                u5 = u3 + u4
                u6 = u4 + u5
                u7 = u5 + u6
                u8 = u6 + u7
                u9 = u7 + u8
                send (semaphor3)
                goto lab4
                }

                task4 {   /*  w!   */
lab5:
                wait (semaphor3)
                w = 1
                w = w*2
                w = w*3
                w = w*4
                w = w*5
                w = w*6
                w = w*7
                w = w*8
                w = w*9
                counter = counter + 1
                if (counter >= 1000000) goto lab6
                send (semaphor4)
                goto lab5
lab6:
                end_time = TICKGET( )
                elapsed_time = end_time - start_time
                printf (elapsed_time)
                }
```

**Execution with minimal Operating System**

```
                unsigned long  start_time, end_time,
                elapsed_time
                long int counter
                int n
                float s, v, t, a
                int u0, u1, u2, u3, u4, u5, u6, u7, u8, u9
                float w

                start_time = TICKGET( )
                counter = 0
lab1:
                n = 1
                n = n*2
                n = n*3
                n = n*4
                n = n*5
                n = n*6
                n = n*7
                n = n*8
                n = n*9

                s = 10.0
                v = 10.0
                a = 9.82
                t = 100.0
                v = v*t
                a = a/2.0
                a = a*t
                a = a*t
                s = s + t
                s = s + v

                u0 = 0
                u1 = 1
                u2 = u0 + u1
                u3 = u1 + u2
                u4 = u2 + u3
                u5 = u3 + u4
                u6 = u4 + u5
                u7 = u5 + u6
                u8 = u6 + u7
                u9 = u7 + u8

                w = 1
                w = w*2
                w = w*3
                w = w*4
                w = w*5
                w = w*6
                w = w*7
                w = w*8
                w = w*9

                counter = counter + 1
                if (counter <= 1000000) goto lab1
                end_time = TICKGET( )
                elapsed_time = end_time - start_time
                printf (elapsed_time)
```

**Parallel execution**

```
                unsigned long  start_time, end_time,
                elapsed_time
                long int counter
                int n
                float s, v, t, a
                int u0, u1, u2, u3, u4, u5, u6, u7, u8, u9
                float w
                start_time = TICKGET( )
                counter = 0
lab1:
                n = 1
                s = 10.0
                v = 10.0
                u0 = 0
                u1 = 1
                w = 1

                n = n*2
                a = 9.82
                u2 = u0 + u1
                w = w*2

                n = n*3
                t = 100.0
                u3 = u1 + u2
                w = w*3

                n = n*4
                v = v*t
                u4 = u2 + u3
                w = w*4

                n = n*5
                a = a/2.0
                u5 = u3 + u4
                w = w*5

                n = n*6
                a = a*t
```

```
u6 = u4 + u5
w = w*6

n = n*7
a = a*t
u7 = u5 + u6
w = w*7

n = n*8
s = s + a
u8 = u6 + u7
w = w*8

n = n*9
s = s + v
u9 = u7 + u8
w = w*9

counter = counter + 1
if (counter <= 1000000) goto lab1
end_time = TICKGET( )
elapsed_time = end_time - start_time
printf (elapsed_time)
```

## Biography

After completing his doctoral studies at the University of Pisa, **Moshe Pelleh** developed several advanced projects in the communication and aerospace industries involving real time embedded systems. In the last years Dr. Pelleh has been a senior lecturer in Holon Institute of Technology, teaching courses in Network Communications, Operating Systems, Real Time Systems and Embedded Systems. He is the founder and head of the Real Time Embedded Systems Laboratory. His research is centred in the field of Real Time Embedded Systems.