Improving SMP Deficiencies Regarding Real-Time Application Characteristics

David Akre, Jon Anderson, Miguel Martinez-Violante,
Nwachukwu Okwudirichi, Greg Romero

Electrical and Computer Engineering, University of Arizona
Tucson, Arizona 85721

dakre@email.arizona.edu, jonanderson@email.arizona.edu, mmartin01@email.arizona.edu,
oknwachukwu@email.arizona.edu, gregrom@email.arizona.edu

Abstract— Improvements to the symmetric multiprocessing architecture regarding real-time applications was mainly done in the user space layer utilizing the Linux operating system for testing improvements; there was however some research done inspecting the kernel and analysis of ARM and x86 processor architectures. The computer systems utilizing ARM and x86 architectures used during this research have multiple processors, and the Linux operating system takes advantage of these hardware characteristics by employing a symmetric multiprocessing (i.e. SMP) environment where multiple applications can run simultaneously on multiple cores sharing the system resources (e.g. IO, memory, and processor). Improvements tested in the user space layer consist of locking a process to a specific processor core, as well as making a process run at a high priority utilizing the Linux scheduler's policy of SCHED_FIFO (this allows processes to pre-empt all other processes running at lower priorities in first in, first out order). Both of these improvements gave processes more predictable response times (e.g. execution times in the research), but overall are not considered to be hard real time improvements due to response times still being somewhat unpredictable. Overall these improvements would be considered soft real time where response times can exceed or undershoot a deadline by a reasonable margin (millisecond precision).

Key Terms: SMP, AMP, RTOS, FPGA, QoS, IPC, CPI, pthread, PID, PMUs, WCET

I. Introduction

Real time systems or applications inherently require processes to produce responses in a deterministic time frame. There's a wide range of these systems that range from time-based feedback controllers in hardware, to hardware state-machines in field programmable gate arrays (i.e. FPGAs), cyclic executives running directly on microcontrollers (typically done utilizing assembly subroutines), to higher level software applications running on an operating system. As the application goes further away from the hardware resources, it becomes much harder to achieve deterministic responses for real time applications. Processors and systems on a chip (i.e. SoCs) are becoming more powerful and complex; so it's much easier to offload the management of these hardware resources to the operating system level. Nowadays, there are operating systems that specialize specifically in real time capabilities (e.g. VxWorks and Green Hills). Unfortunately, these software systems are expensive to use, and many of the free options out on the market have not been proven to be real time certified. Thus, there is motivation to improve Linux's SMP architecture to become more suitable for real time applications.

A typical computer operating system runs in three states: user space, kernel, and the idle state. Since most applications are executed in the user space this felt like a natural area to focus the research in; though there was some research done in the kernel level by inspection and analysis (as well as inspection of the hardware targets used in this research—Intel i7 x86 and ARM A57 Cortex A8 multiprocessors used on a Lenovo workstation and NVIDIA Jetson TX1 platform).

As mentioned above, the focus of this research was done in the user space utilizing Linux as the primary operating system. Real time applications require deterministic responses, and since Linux is naturally a time-sharing operating system (splitting individual processes to execute in a time slice), this required some research and development to create a Linux environment suitable for real-time applications. Two promising improvements found are overriding the natural time-sharing scheduling policy by changing the scheduler to schedule processes in FIFO order (i.e. First in First out), and secondly locking a process to a specific core.

First, changing the scheduler to run SCHED_FIFO allows for processes running at higher priority than other processes to pre-empt the lower priority processes and fully utilize the system's hardware resources. The Linux scheduler can be modified in the user space with root privileges and by utilizing its posix thread library (i.e. pthread library). Allowing a process that is real time to run at a higher priority than another process not only allows for it to fully utilize the hardware resources available, but also significantly reduces the context switching time, which can be very significant with virtual memory page swapping depending on the size of the real time application. This option proved to be more successful in achieving real-time results from applications in comparison to the natural round robin scheduling policy the Linux scheduler employs.

Second, locking a process to a specific core also significantly increased predictability of responses. This was done by utilizing Linux's "set_affinity" functionality, which is a system function call from the user space to the kernel to selectively place a process from its process id (i.e. pid) to a specific processor core. This method proved to be more effective on x86 and ARM targets. Overall, utilizing both methodologies will lead to an increase in predictable responses for real-time applications.

Since much of the focus and improvements found in the user space were overriding the Linux scheduler's policy and placement of processes on the system, it made sense to dive deeper into this area in the kernel level. The scheduler can be found in the Linux source tree "Linux/kernel/sched/." Looking through the source code, the context switching can be found in core.c, and then processor specific interfaces to switch threads are located in "<processor_name>/include/asm/switch_to.h". Both ARM and x86 have similar functionality of switching threads, but with some slight differences. They both utilize structs to hold information about “previous” and “next” thread addresses and states. Which appears to be a quick interchange, but doesn’t quite acknowledge how large each threads context might be. This can be a major issue in real time systems and will significantly impact execution times. An improvement here can be to associate a process with the last known stack size of which the process was previously executing in (i.e. knowing the processes page size before context switching). If the scheduler is executing with SCHED_FIFO policy, it can check not only the priorities of processes in its ring buffer, but also the page size associated with each process. It may make more sense to execute a process with a smaller a smaller page size in comparison to a different process that has a slightly higher priority with a larger page size. A higher priority process doesn’t need to execute fast, it only need to execute on time.

Lastly, when looking at the hardware experimental setup, the x86 architecture proved to be more effective than ARM (which is very surprising because many real time systems, especially in the aerospace industry utilize ARM processors). The x86 architecture across all user space improvements gave more predictable results than the ARM processor.

Overall, the goal of this research was to extend on previous research done in this area, but at the same time be unique by utilizing different techniques. The first research paper relevant in this area was done by NASA’s Jet Propulsion lab. They performed research on SMP architectures and laid out the pros and cons of migrating legacy flight software from single processor systems to multicore processors in space systems. They performed this research by comparing single processors, specifically BAE rad hardened RAD750 processor, to multicore processors; additionally, they look into the advantages and disadvantages of an asymmetric multiprocessing (i.e. AMP) core with a hypervisor. They go onto to analyse inter-process communication, interrupt handling, fault tolerance, and other software applications at a higher level of which these processes may benefit in a multicore environment in comparison to a single core system [1]. The University of Leuven also had similar research in this area, but their focus was on the evaluation of SMP versus AMP for embedded architectures for telecommunications applications, but no real time focus. Their research focuses more on the advantages and disadvantages of sharing system resources, and why AMP would be more suitable for applications that require resources that are more locked in comparison to processes that rely on the sharing of resources [3]. The final paper was done by Dayananda Sagar Institutions which focused their research more on the distribution of tasks between processing cores in regards to multicore applications in real time systems [2]. The research performed in this paper is to focus on improving the deficiencies found in the SMP architecture, implement those improvements, and to analyse the improvements on various hardware targets (e.g. embedded SoCs to modern desktops).

II. Related Work

The three research papers, briefly talked in section two, discuss how SMP architectures impact real time and telecommunication applications. The first research paper, performed by NASA, focuses on how legacy real time embedded flight software may benefit or not in an SMP environment [1]. The second and third papers were done by international universities (i.e. University of Leuven in Belgium, and Dayananda Sagar institutions in India). University of Leuven performed research comparison SMP and AMP architectures in regards to telecommunication applications (most of the time telecommunication applications require a high QoS—quality of service and is loosely tied with providing predictable results to enable a high QoS) [3]. Lastly, Dayananda Sagar institutions performed research focusing on multicore applications performances in AMP verses SMP with the utilization of real time operating systems (i.e. RTOS). [2].

NASA’s Jet Propulsion Lab’s research in multicore SMP considerations for real time application looks at the pros and cons of multicore systems running legacy embedded flight software verses the traditional single core environment. They acknowledge that multiple processors add an increase to the overall system performance by showing how Amdahl’s law will double the performance from a single core to an eight core system. Additionally, they look at how flight software can be migrated from single core to multicore by potentially utilizing a hypervisor to manage system resources and having real time applications run in an independent AMP core [1]. The research performed by NASA motivated this research paper to dive into this area, but did not address how pitfalls of SMP architecture in real time systems could be addressed or improved, which is the focus of this research.

The next two research papers are similar by comparing AMP and SMP architectures on multicore systems for telecommunication and real time applications. The University of Leuven sought to evaluate the performance of an SMP architecture in comparison to AMP architecture in regards to telecomm applications. Their approach is similar to the approach used in this research by setting up an environment to measure SMP verses AMP by utilizing performance monitoring tools, and then by setting up experiments using telecommunication benchmarks to compare dual-core SMP multicore embedded systems to dual-core AMP embedded systems. Their overall results found that AMP systems hit a bottleneck in performance to deliver packets from processor A to processor B, whereas SMP naturally excelled in distributing tasks because its memory hierarchy is not divided like AMP [3]. Their process of performing research motivated this research’s process by setting up a simulation environment to gather CPU metrics from the benchmark under test, as well as actually developing the benchmarks to be tested. Overall, this research does not evaluate the differences in AMP verses SMP, but instead focuses on the real time improvements to SMP which the University of Leuven did not perform.

Lastly, Dayananda Sagar Institutions performed research on the performance of multicore applications in real time systems. Instead of proposing improvements to SMP they evaluated the performances of an RTOS. They looked into how RTOS’s differ from AMP and SMP architectures. Their findings were that AMPs are advantageous in heterogeneous systems, but fail to perform well with multiple cores, which bottlenecks fast in terms of overall throughput. SMP architectures picked up the slack in this area by sharing systems resources across multiple cores and caches. In conclusion, they believe an RTOS can offset a lot of the programmer’s worries when it comes to writing real time applications on a single-core processor since this has been proven out on AMP architectures, but is still underdeveloped on multicore SMP architectures [2].

III. Methodology

The first part of this research was researching where improvements could be made to the SMP architecture utilizing Linux as the operating system to test on. While performing this research, it made sense to try to improve the scheduler in SMP and attempt to reduce context switching times, or even make this time predictable. Since SMP architectures have the ability to execute multiple processes simultaneous on multiple cores, this research seeks out how a user can control the scheduler to create a real time environment in the user space on a multicore SMP system.

Second part was to improve the real time SMP deficiencies by implementation. In the user space the user can create a soft real time environment for SMP multicore systems by doing one or both of the following: set the real time application process to run at the highest priority by changing the scheduler's scheduling policy to FIFO, and secondly by locking the real time application's process to a specific core (similar to AMP, but still running in an SMP environment). The user can change the scheduling policy by invoking the "set_sched_policy" to SCHED_FIFO when setting up pthreads in the user space. Additionally, the user can lock a process to a specific core by calling the system function "set_affinity" by giving it an argument of a valid core id. The following UML diagram displays the software architecture used to construct these SMP real time improvements, which can be easily integrated in a benchmark:

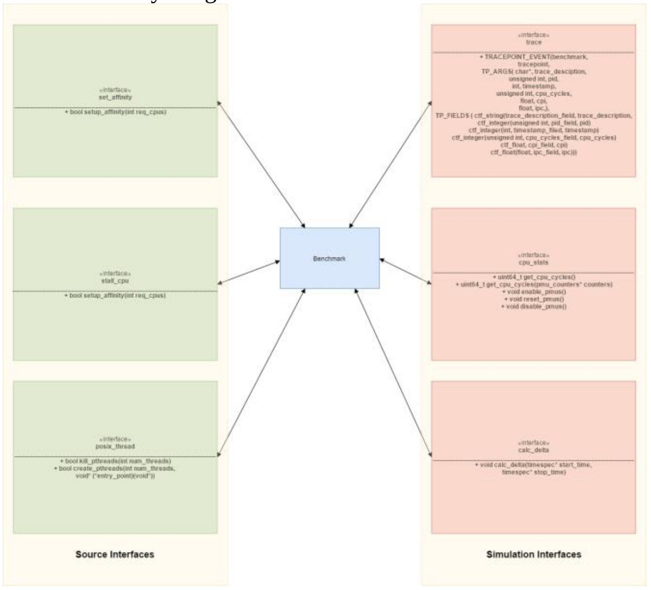

«interface»
set_affinity
+ bool setup_affinity(int req_cpus)
«interface»
stall_cpu
+ bool setup_affinity(int req_cpus)
«interface»
posix_thread
+ bool kill_pthreads(int num_threads)
+ bool create_pthreads(int num_threads, void* (*entry_point)(void*))
Source Interfaces
Benchmark
«interface»
trace
+ TRACEPOINT_EVENT(benchmark, tracepoint, TP_ARGS(char*, trace_desciption, unsigned int, pid, int, timestamp, unsigned int, cpu_cycles, float, cpi, float, ipc,), TP_FIELDS (ctf_string(trace_description_field, trace_description, ctf_integer(unsigned int, pid_field, pid) ctf_integer(int, timestamp_filed, timestamp) ctf_integer(unsigned int, cpu_cycles_field, cpu_cycles) ctf_float, cpi_field, cpi) ctf_float(float, ipc_field, ipc)))
«interface»
cpu_stats
+ uint64_t get_cpu_cycles()
+ uint64_t get_cpu_cycles(pmu_counters* counters)
+ void enable_pmus()
+ void reset_pmus()
+ void disable_pmus()
«interface»
calc_delta
+ void calc_delta(timespec* start_time, timespec* stop_time)
Simulation Interfaces

Fig 3.1 UML Diagram of software architecture used in research

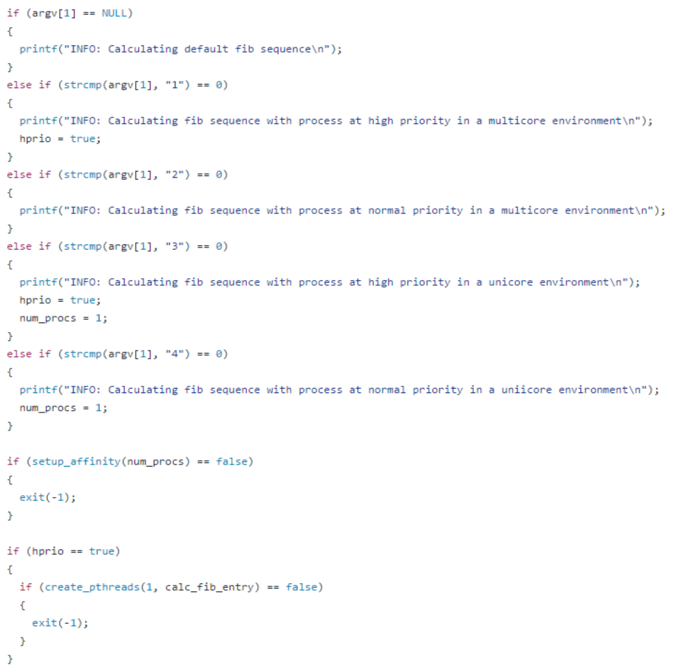
The software architecture includes three main interfaces to enable the benchmark executing in a soft real time environment: posix_thread, stall_cpu, and set_affinity. The posix_thread interface gives the user creating benchmarks an easy function call to create posix threads running at the highest priority available according to the SCHED_FIFO policy, it also will enable this scheduling policy when invoked. The set_affinity interface, when invoked by the benchmark, will take its pid and lock it to a validated and properly configured processor. Overall both of these techniques led to overall improvements in predictability of execution times.

The third part of the research process was to create a suitable simulation environment to measure the benchmark's execution time, cycles per instruction (i.e. CPI), instructions per cycle (i.e. IPC), and CPU cycles. Since the benchmark will be executing at a high priority and has a modified scheduler, changed from round robin to FIFO, it's counter-intuitive to run another process simultaneously to measure the benchmark. The reasoning for this is primarily because the benchmark will pre-empt whatever process is it's trying to measure. Additionally, if one wanted to change the simulation tools execution priority, then the benchmark would potentially have contention with processor and memory resources with the simulation tool. Thus, the research was performed using a hybrid approach from utilizing lttng-ust in x86 (not ARM) to help trace the process's metrics, but also manually configuring and reading processor monitoring units (i.e. PMUs) in ARM and the control register(s) in x86 to measure help measure CPU metrics. Lastly, static analysis was used to count the instructions from the benchmark by compiling it with the "-s" flag on both ARM and x86. This allowed for the hybrid simulation environment to extrapolate CPI and IPC by utilizing the following formula(s):

(1) CPI = CPU Time / (Instruction count * Cycle Time)
(2) IPC = 1 / CPI

The interfaces that helped enable this simulation environment are displayed in figure 3.1 on the right (i.e. trace, calc_delta, and cpu_stats). Trace helped with logging the metrics outlined above for the benchmark utilizing lttng-ust's library. The second interface, "calc_delta", essentially calculates the execution time between start and stop timespec struct which is defined in the benchmark (helped with measuring response times). Lastly, the "cpu_stats" interface helps calculate the CPU cycles from the PMUs in ARM and control registers (i.e. eax) in x86. Thus gathering all of these metrics from the benchmark were measured manually and within the process context which allowed for the truest results without hardware resource contention or unnecessary pre-emption. In addition to these interfaces, there are two main python scripts (namely sim_lttng_rt.py for x86 and sim_arm_rt.py for ARM targets) that quantify and correlate all of the benchmark metrics produced on both ARM and x86 architectures to produce csv files to be used to create graphs based on the results.

Last part was to create a benchmark to pull all of this together. The benchmark chosen was a simple program that executes the Fibonacci sequence for up to 5ms. Keeping the benchmark simple allowed for the research to pin-point where the improvement gains were, as well as where future research can be conducted. Additionally, the benchmark used in this research can be configured dynamically by user input either from the simulation script input or by executing the raw binary (e.g. the configuration setup can be: high priority process on a multicore environment, normal priority process in a multicore environment—default settings, high priority process in a single-core environment, and normal priority process in a single-core environment… this can be seen in figure 3.2). Below are code snippets from the benchmark used:

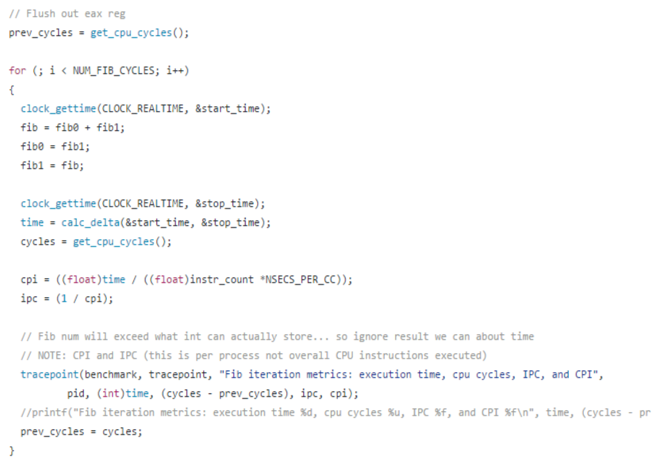


```
if (argv[1] == NULL)
{
  printf("INFO: Calculating default fib sequence\n");
}
else if (strcmp(argv[1], "1") == 0)
{
  printf("INFO: Calculating fib sequence with process at high priority in a multicore environment\n");
  hprio = true;
}
else if (strcmp(argv[1], "2") == 0)
{
  printf("INFO: Calculating fib sequence with process at normal priority in a multicore environment\n");
}
else if (strcmp(argv[1], "3") == 0)
{
  printf("INFO: Calculating fib sequence with process at high priority in a unicore environment\n");
  hprio = true;
  num_procs = 1;
}
else if (strcmp(argv[1], "4") == 0)
{
  printf("INFO: Calculating fib sequence with process at normal priority in a uniicore environment\n");
  num_procs = 1;
}

if (setup_affinity(num_procs) == false)
{
  exit(-1);
}

if (hprio == true)
{
  if (create_pthreads(1, calc_fib_entry) == false)
  {
    exit(-1);
  }
}
```

Fig 3.2 Dynamic configuration for real time SMP benchmark(s)



```
// Flush out eax reg
prev_cycles = get_cpu_cycles();

for (; i < NUM_FIB_CYCLES; i++)
{
  clock_gettime(CLOCK_REALTIME, &start_time);
  fib = fib0 + fib1;
  fib0 = fib1;
  fib1 = fib;

  clock_gettime(CLOCK_REALTIME, &stop_time);
  time = calc_delta(&start_time, &stop_time);
  cycles = get_cpu_cycles();

  cpi = ((float)time / ((float)instr_count *NSECS_PER_CC));
  ipc = (1 / cpi);

  // Fib num will exceed what int can actually store... so ignore result we can about time
  // NOTE: CPI and IPC (this is per process not overall CPU instructions executed)
  tracepoint(benchmark, tracepoint, "Fib iteration metrics: execution time, cpu cycles, IPC, and CPI",
        pid, (int)time, (cycles - prev_cycles), ipc, cpi);
  //printf("Fib iteration metrics: execution time %d, cpu cycles %u, IPC %f, and CPI %f\n", time, (cycles - pr
  prev_cycles = cycles;
}
```

Fig 3.3 Fibonacci benchmark algorithm code snippet

As seen in figure 3.3, the Fibonacci sequence benchmark calculates the CPU metrics—CPI, IPC, execution time, and CPU cycles, and logs these events by using the tracepoint lttng-ust function call setup by the trace interface. Using system logs or traces helps with gathering execution times in a more real time environment rather than using a printf calls which will interrupt the current process to log the event to the terminal (this adds extra unpredictability this research seeks to prevent).

Overall to the environment setup is designed to be fairly simple for any user to compile and run simulations against the binaries on x86 or ARM architectures. When running on x86, the user can simply execute the sim_lttng_rt.py script by giving it the benchmark binary, the type of trace (can be kernel or user space), the binary configuration options, and an output name for the output csv file (e.g. execute "./sim_lttng_rt.py –i fib –t userspace –opt 1 –o fib-output"). Similarly, on ARM architectures the user can compile and run the benchmark normally and then simply run the sim_arm_rt.py script to correlate all of the results written to syslog's and output them into a csv file (the script takes arguments similarly to the sim_lttng_rt script).

IV. EXPERIMENTAL RESULTS

The experimental setup contains software architecture for setting up the SMP real time improvements, the simulation environment scripts created in python, and the source files for ARM and x86 benchmark architectures, as well as the following test hardware:

	NVIDIA Jetson TX1	Lenovo Workstation
CPU	Quad Core ARM A57 Cortex A8	Octa Core Intel i7-3840QM
CPU Clock Speed	1.8 Ghz	2.8 Ghz
Memory	4Gb LPDDR4	16Gb DDR3

Table 4.1 Hardware test devices used to perform research

The benchmarks can be compiled for ARM and x86 architectures, and the simulations are also split up to be ran on both environments as well. Thus this makes testing benchmarks more independent of one another depending on the target hardware. Both hardware test devices are running Linux (with minute differences in their kernel version due to NVIDIA issuing a proprietary JetPack which has a "Linux4Tegra" kernel version, but is still based off of the current Linux kernel source).

The experimental results can be analysed from a holistic view of the benchmark, as well as analysing each loop iteration. The first presentation of the results will reflect the a holistic view by taking the average of five simulations of each of the four user space real time SMP improvements configurations executing the Fibonacci sequence benchmark, which is shown in figure 3.2. The following graphs show both the simulations for ARM and x86 which show the measurements for total execution time per execution count (i.e. each individual run).

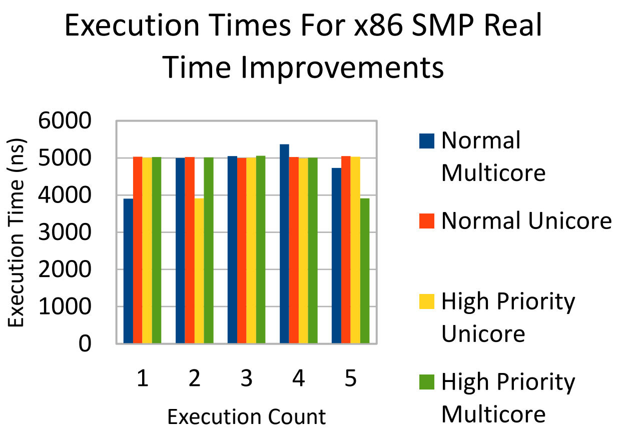


Fig 4.1 Total execution times calculated on the Fibonacci sequence benchmark for SMP x86 architecture

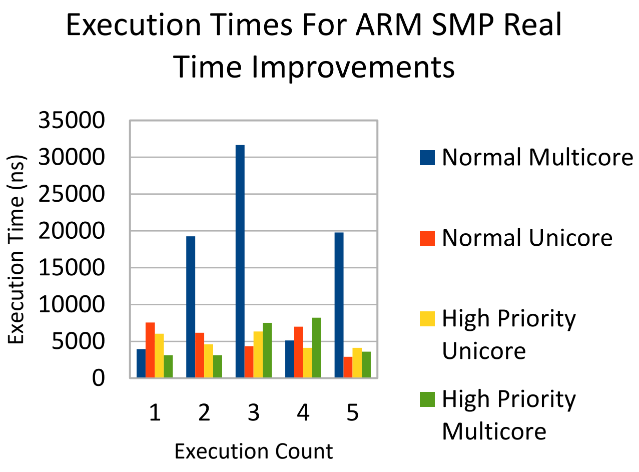


Fig 4.2 Total execution times calculated on the Fibonacci sequence benchmark for SMP ARM architecture

By visually inspecting figures 4.1 and 4.2 it appears that the x86 architecture produced much better results overall in producing deterministic responses from the benchmark targeting 5ms execution times (especially just for employing the regular SMP architecture by default). Additionally, both improvements (i.e. modifying scheduling priority and locking processes) to the SMP architecture gave more predictable response times in comparison to the "normal multicore" environment the Linux scheduler naturally utilizes. Surprisingly the "normal unicore" configuration produced similar if not the best results from these simulations in comparison to the expected best configuration which is "high priority unicore". The normal unicore configuration is just simply locking a process to a specific core but allowing the scheduler to execute that process at normal priority). A couple reasons for this could be that since Linux is a natural time-sharing operating system that it executes processes the best this way (allowing the scheduler to run priorities in a round robin fashion utilizing its shared memory which may lead to less memory contention issues), another reason may be that when the user runs a process at the highest priority, the system potentially will lock up and may starve necessary kernel processes from executing (the user actually has to run as root in order to set a process in user space to run at the highest priority utilizing SCHED_FIFO because the user space is protected). Though utilizing both “high priority and single-core” environments led to significant improvements to the SMP architecture to become much more real time than what it naturally is (especially in ARM architecture).

Next looking at the rest of the metrics collected at a holistic view, one can see how CPU cycles, IPC, and CPI might be intertwined with the effects of the execution times shown in figures 4.1 and 4.2.

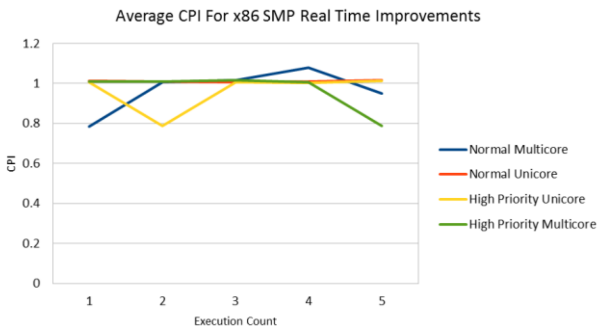


Fig 4.3 Average CPI per iteration for x86 SMP architecture executing the Fibonacci benchmark

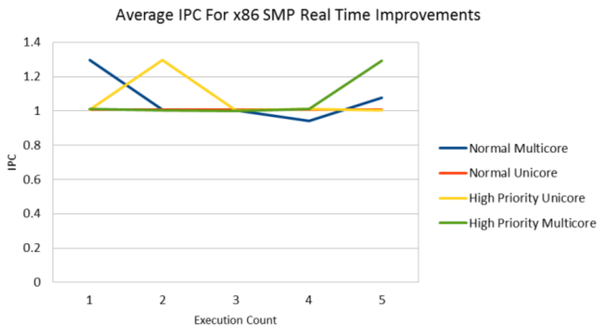


Fig 4.4 Average IPC per iteration for x86 SMP architecture executing the Fibonacci benchmark

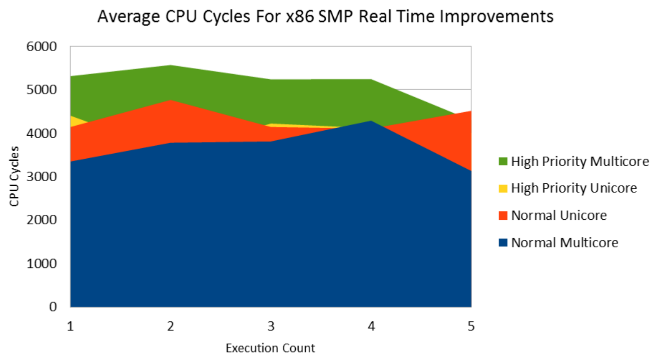


Fig 4.5 Average CPU cycles per iteration for x86 SMP architecture executing the Fibonacci benchmark

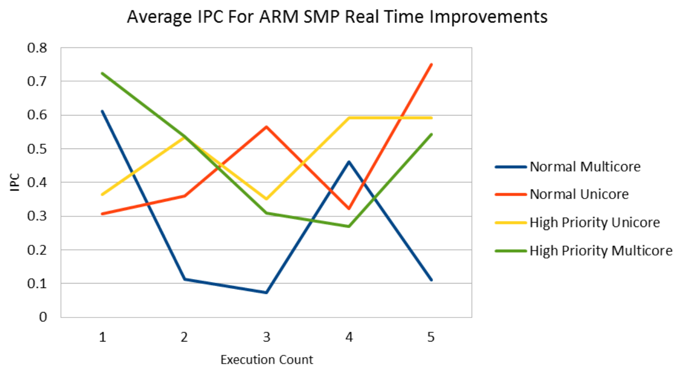


Fig 4.6 Average IPC per iteration for ARM SMP architecture executing the Fibonacci benchmark

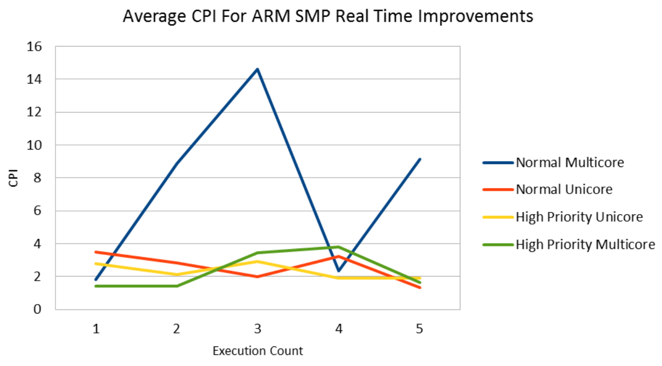


Fig 4.7 Average CPI per iteration for ARM SMP architecture executing the Fibonacci benchmark

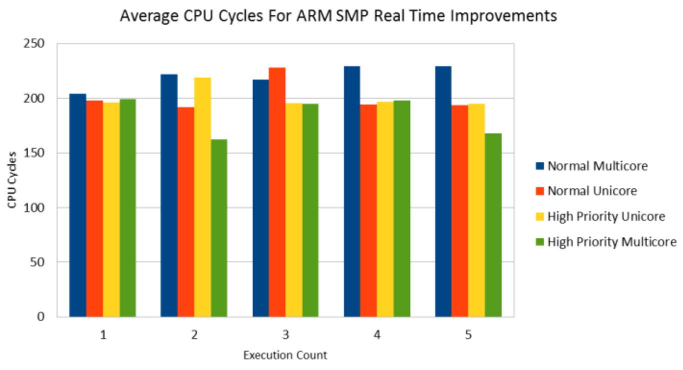


Fig 4.8 Average CPU cycles per iteration for ARM SMP architecture executing the Fibonacci benchmark

Now by visually inspecting figures 4.3 through 4.8 the x86 i7 processor definitely performs better overall than the ARM A57 processor. Since the x86 processor used is much more powerful than the ARM processor it may have helped mitigate potential hardware resource contention in the processor and memory, which ultimately produce overall better results for real time capabilities. The average CPU cycles issued by the i7 processor was about fifteen times more than the ARM A57 processor. Additionally, the average IPC and CPI hovered around one for the i7 process under test, which is much better than the results received on the ARM processor in which its IPC would range dramatically (e.g. in a normal multicore environment it would range from a high of 0.6 to a low of under 0.1 which is not optimal for real time in the slightest). On average the IPC of the ARM processor for the improvements could be labelled at around 0.5, and the CPI around 2-3. Whereas, the average IPC and CPI for all architectures in SMP on the x86 processor was around 1-1.2 which is much more optimal for real time systems. These results are very surprising considering the market for real time systems is ARM based at the moment. One reason the x86 processor outperformed ARM in this case might be due to the resources it had—16Gb DDR3 RAM, 2.8Hz clock which had a much higher CPU cycle count than the ARM processor, and also had eight cores.

Since the x86 processor produced much more ideal results from the improvements made to the SMP architecture to become more real time capable, it's worth taking a deep dive into the ARM processor as to why it produced less-than optimal results for real time capabilities. The following graphs display the worst configuration to best configuration's CPU metrics captured on the NVIDIA Jetson TX1 SoC.

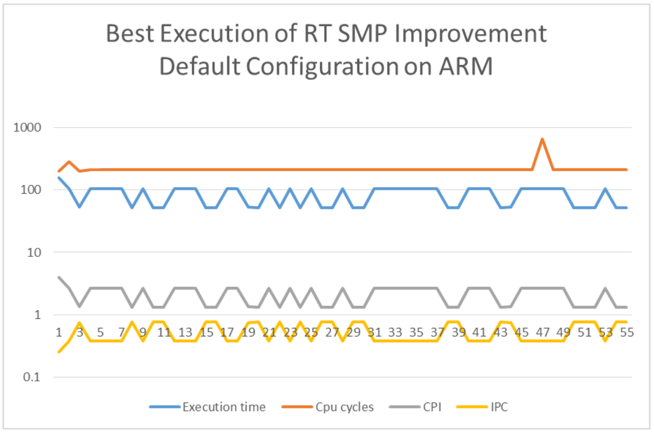


Fig 4.9 Execution time, CPU cycles count, CPI, and IPC of the best results on ARM for real time SMP improvements on a log base 10 scale

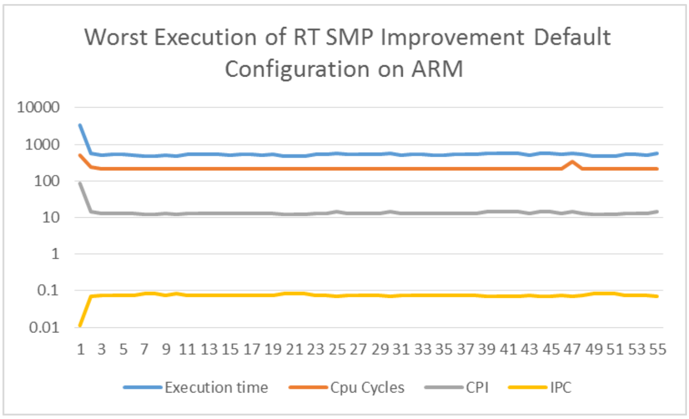


Fig 4.10 Execution time, CPU cycles count, CPI, and IPC of the worst results on ARM for real time SMP improvements on a log base 10 scale

Both of these graphs give good insight as to why there is such a drastic difference between best and worst execution times on ARM. In the first iteration there is a compulsory cache miss which produced terrible CPI, IPC, and execution time results. Secondly, the worst case execution time (i.e. WCET) appears to be consistently around 500 ns, which ultimately leads to a total execution time of around 30,000 ns which is shown in figure 3.2 (can also be calculated from 55 iterations multiplied by 500 ns will equate to 30,000 ns). Each iteration ideally should be executing at around 90 ns, which on figure 4.9- the best configuration, shows each iterations WCET hovering around 80-100ns which is ideal. Additionally, one can see how CPU cycles, IPC, and CPI are intertwined to the overall execution time by looking at the best configuration displayed on figure 4.9. Besides the spike around iteration 47, the CPI and IPC are closer to 1, and the CPU cycles are just over 100 which ultimately leads to predictable results from the Fibonacci benchmark.

V. Conclusions and Future Work

The research conducted on Linux to improve SMP architecture to become more real time capable for real time applications overall was successful. The two proposed improvements to the SMP architecture by overriding the scheduler's scheduling policy to become FIFO and run processes at a high priority led to more predictable results in comparison to the default round robin scheduling policy. Additionally, locking a process to a specific core (similar to AMP) also led to more predictable responses from processes in SMP.

Overall these improvements are advised to be soft real time improvements to the SMP architecture. There are no configurations that support 100% success in achieve exactly 5ms response times in the simulations ran in this research, but overall the configurations implemented do improve the deficiencies in SMP to be more real time capable.

There still is a lot of research to be done in this area. As consumers begin utilizing more telecommunication applications or perhaps Internet of Things (i.e. IoT) applications, real time capabilities will be necessary to provide a high QoS but still maintain a high throughput. Thus, research focusing on real time application improvements in SMP on multicore systems with a focus on memory intensive or IO applications would be very interesting. In addition to the telecomm industry, the space industry is also transitioning over from single core processors to multicore processors to allow satellites and space vehicles to process more data in space rather than downlinking down to earth which takes much longer to do. Thus, expanding on this research and focusing on how to maintain a high throughput but still maintain a SMP real time environment would be very interesting.

Additionally, continuing this research to improve SMP further by other techniques in the user space layer and to test out these improvements on more hardware targets (e.g. A53, Intel Atom, AMD processor, etc.) would help gather knowledge in this area of research. It was very interesting to see the x86 processor to perform so well in comparison to the ARM processor; so research into as why this was and to compare other types of architectures with different memory hierarchies would also be very conducive to this research.

In conclusion, the user space improvements to SMP architecture to become more real time proved out to be modifications made to the scheduler—modifying the scheduler to run FIFO with processes at high priority, in addition to locking a process to a specific core. When analysing the kernel, the context switching interfaces for specific hardware devices can be updated and modified to check the page size before switching to a new process (larger stack equivalently means larger page size to load from virtual memory). The goal of this modification would be to keep these sizes as small as possible, and to switch to processes with smaller page sizes more frequently.

References

[1] Kenneth Vines, and Len Day, "Multicore Considerations for Legacy Flight Software Migration" JPL NASA [Online] Available:https://trs.jpl.nasa.gov/bitstream/handle/2014/43215/12-5292_A1b.pdf?sequence=1 Accessed: Feb. 7, 2017

[2] Vaidehi M, T.R. Gopalakrishman Nair, "Multicore Applications in Real Time Systems" [Online] Dayananda Sagar Institutions

Available: https://arxiv.org/pdf/1001.3539.pdf, Accessed: Feb 8, 2017

[3] Nico De Witte, Robbie Vincke, Sille Van Landschoot, Eric Steegmans, and Jeroen Boydens, "Evaluation of a Dual-Core SMP and AMP Architecture based on an Embedded Case Study" July 2013 KU Leuven https://www.researchgate.net/publication/309829008_Evaluation_of_a_Dual-Core_SMP_and_AMP_Architecture_based_on_an_Embedded_Case_Study, Accessed: Feb 8, 2017

[4] Group 3's Source Control Repository: https://github.com/dakre21/ECE-562