

Evaluation of Cell BE SPU Scheduling for Multi-programmed Systems

Julio M. Merino-Vidal, Isaac Gelado and Nacho Navarro

Departament d'Arquitectura de Computadors. Universitat Politècnica de Catalunya

C/Jordi Girona 1-3, Campus Nord. 08038 Barcelona, Spain

E-Mail: {jmerino,igelado,nacho}@ac.upc.edu

Abstract—The Cell Broadband Engine (CBE or Cell for short) is an heterogeneous many-core processor capable of achieving a theoretical maximum performance peak of 204.8 GFLOPS. In order to get such peak performance, programmers have to partition applications in a set of tasks that are executed by the different processing elements found in the Cell. Application partitioning requires fine-tuning the computational requirements of each task to accomplish correct load-balancing and, therefore, to use all the Cell computational elements. Despite the programmer's efforts, if more than one application runs concurrently on the Cell, the ultimate responsible for accomplishing load-balancing is the operating system scheduler.

This paper presents an experimental evaluation of the Cell Synergistic Processing Unit (SPU) scheduler that is part of the Linux kernel. We analyze the strengths and limitations of the SPU scheduling algorithm currently implemented in Linux. We show that under certain conditions, which occur often, the current scheduler leads to the under-utilization of the Cell hardware resources. Finally, we sketch how the Linux SPU scheduler should be modified to accomplish a more efficient utilization of the aforementioned resources.

I. INTRODUCTION

The Cell Broadband Engine (CBE or Cell for short) [1] is an heterogeneous architecture developed by Sony, Toshiba and IBM. The Cell includes a general purpose core (the PowerPC Processing Unit (PPU)), multiple programmable accelerators (the Synergistic Processing Units (SPUs)), and a high-speed interconnection bus (the Element Interconnection Bus (EIB)). The current version is a single chip that contains one PPU and eight SPUs. The PPU is an in-order, 2-way Simultaneous MultiThreading (SMT), PowerPC 970-compatible general purpose core running at 3.2GHz. Each SPU is a vectorial accelerator targeted at the execution of Single-Instruction Multiple-Data (SIMD) code and is composed of an execution core, 256KB of Local Store (LS) memory and a Memory Flow Controller (MFC). Examples of systems including the Cell are the IBM Cell Blades and the Sony PlayStation 3.

The Cell has been officially supported by Linux since version 2.6.16. Linux defines two classes of threads: the *PPU threads*, which run on the PPU, and the *SPU tasks*, which run on the SPUs. The overall scheduling approach for SPU tasks

is to create a *control PPU thread* for each SPU task. Coupling each SPU task to a PPU thread simplifies the system support for the Cell but might lead to contention problems whenever there are more SPU tasks than physical SPUs.

The first SPU scheduler implemented in Linux used a cooperative algorithm: a task assigned to a given SPU ran until it voluntarily yielded execution. Task scheduling was completely re-implemented in Linux 2.6.23 [2]: PPU threads are managed by the Completely Fair Scheduler (CFS) [3] and the SPU scheduler supports time-sharing in multiprogrammed workloads.

Current applications for the Cell typically create as many SPU tasks as physical SPUs are in the system. At run-time, the old cooperative SPU scheduler assigned each SPU task to a physical SPU, and these tasks were not preempted until their execution voluntarily yielded. If any other application spawned a SPU task, it blocked until an SPU was released. Under these circumstances, the performance of the first application was not affected by the second one, but the execution time of the second application became unacceptable.

The new Linux SPU scheduler allows the preemption of SPU tasks¹. When there is more than one Cell application running on the system, it is highly probable that there will be more active SPU tasks than physical SPUs. In this scenario, the contention in the access to the SPUs degrades the performance of the applications. This situation is currently avoided in Cell-based systems by allowing only one active application to run at the same time.

A single-application environment might be acceptable for super-computing or gaming consoles. However, we expect the Cell and/or other heterogeneous many-core chips to reach the desktop market. Desktop systems present very different workloads: many applications might want to make use of SPU tasks concurrently. For instance, consider a PlayStation 3 acting as a Linux set-top-box and playing a high-definition H.264 movie. The video decoder requires the six SPUs available in the PlayStation 3 to render the video stream. If the user launches the web browser to get information about the movie, the JPEG decompression process might also require one or more SPUs, therefore degrading video playback performance.

We argue that a programming model that assumes an unlimited number of SPUs presents several advantages when

This work has been supported by the European Commission in the context of the SARC Integrated Project (EU contract 27648-FP6), the HiPEAC European Network of Excellence, the Ministry of Science and Technology of Spain and the European Union (FEDER) under contract TIN2007-60625. All products or company names mentioned herein are the trademarks of their respective owners.

¹The old SPU scheduler did support preemption, but only for real-time contexts. We do not consider this case.

implementing commercial software. First, it does not force programmers to fine-tune applications for a specific hardware configuration, which reduces development time. Second, this programming model produces scalable applications. An application that spawns more SPU tasks than physical SPUs fully utilizes the current hardware resources, as current applications do, but when a new chip with a higher number of SPUs is released, that same application transparently exploits all the hardware resources without being rebuilt. However, this programming model relies on the operating system scheduler to accomplish efficient load-balancing in the SPUs.

This paper presents an experimental evaluation of SPU task scheduling on multiprogrammed systems. Experimental results show that the performance of an application can decrease up to a 50% due to inefficient management of hardware resources. We identify the major issues in the current SPU scheduler that prevent it from working efficiently with multiprogrammed workloads and sketch the modifications that should be introduced in the Linux scheduling code to accomplish better load-balancing in the SPUs.

The main contributions of this paper are:

- An introduction to the *Cetra* tool-set. *Cetra* is a set of software tools to allow fine-grained tracing of the Linux SPU scheduler and user-level applications. The traces produced by *Cetra* are useful to analyze the execution of the applications, to study the kernel scheduling decisions and to identify performance bottlenecks.
- An experimental evaluation of the Linux SPU scheduler. This evaluation shows that the current scheduling algorithm does not perform efficiently whenever there are more running SPU tasks than physical SPUs.
- A description of the required changes in the Linux task schedulers to allow efficient utilization of the hardware resources present in the Cell processor.

This paper is organized as follows. Section II describes the current Linux support for the Cell; Section III describes our experimental methodology; Section IV explains and analyzes the experimental results; Section V sketches the necessary modifications to the Linux task schedulers required to accomplish adequate SPU load-balancing; and, finally, Section VI concludes.

II. CELL SUPPORT IN LINUX

The Linux kernel code that manages the PPU is the very same code used to manage PowerPC 64-bit processors, with some minor additions to deal with the specific details of the architecture. The SPUs are managed by Cell-specific code. The kernel exports SPUs to user-level applications using the SPU File System (*spufs*) [4], a virtual file-system that abstracts SPUs and implements the SPU scheduler.

This section describes the key concepts and parts of the Linux code that are needed to understand the rest of the paper.

A. The CFS scheduler

The CFS scheduler, added to Linux 2.6.23, attempts to model an ideal, precise multi-tasking CPU on real hardware. This concept refers to a CPU that is able to run N threads at

precise equal speed, in parallel, each one at $1/N$ of the total CPU speed. CPUs can only run a thread at once², so while one thread has the CPU, those that are waiting for it get an unfair amount of CPU time. This unfairness is tracked per thread at the nanoseconds level.

Each thread has a virtual run-time clock known as *vruntime*, which, roughly speaking, keeps track of the amount of CPU time the thread has consumed with respect to others. CFS orders threads according to their *vruntime*, and the scheduling policy always picks the one with the lowest value to allow it to gain more run-time and match those that have already got more CPU time in a scheduling period.

To avoid over-scheduling situations, which could arise due to the fine granularity of *vruntime* (also measured in nanoseconds), some thresholds are dynamically defined. Their values are based on the number of active runnable threads and user-tunable parameters. These user-tunable parameters are described in Section V.

B. SPU contexts

The core abstraction for the SPUs is the SPU context. Each SPU context virtualizes a physical SPU and decouples the SPU task from its real execution. The relation between SPU contexts and SPUs is similar to threads and PPUs, respectively. Linux assumes that user-space applications explicitly create and manage SPU contexts to prepare and send tasks to SPUs. Then, the SPU scheduler is in charge of assigning each SPU context to a physical SPU, and time-multiplexing the physical SPUs among all the runnable SPU contexts.

A SPU context is a data structure stored in main memory and contains the execution state of the SPU. This state is composed by the values of the SPU registers and the contents of the LS. Context switching is implemented through two operations: *bind* restores a SPU context on an SPU and *unbind* saves the execution status of an SPU into an SPU context.

User-space code requests the execution of a SPU context with the *run* operation (*spu_run* system call). This operation is synchronous: it does not return until the given SPU task finishes its execution. Internally, the SPU context is added to the run queue of the SPU scheduler and the run loop is entered (see Section II-C).

While the user-space control thread is blocked waiting for the *run* call to return, its SPU task can be bound and unbound by the SPU scheduler at will. These actions trigger the control thread to wake up and run its kernel-level activation routine (details to follow). The user thread is never notified of the context switches.

C. The SPU scheduler

SPU scheduling in Linux 2.6.23 implements two different policies: *NORMAL* and *FIFO*. The *NORMAL* policy implements time-sharing capabilities and the *FIFO* policy implements the run-to-completion behavior. One of these two policies is assigned to each SPU context.

²SMT processors can run more than one thread at once but, from the point of view of the kernel, these threads are seen as independent scheduling entities.

NORMAL is the default policy for new contexts. If we set up *all* the SPU contexts in the machine to use *FIFO* as their scheduling policy, we effectively emulate the behavior of the cooperative SPU scheduler in pre-2.6.23 versions.

```

Input: ctx
1 repeat
2   (found, spu) ← find a free SPU ;
3   if found then
4     Bind ctx to spu ;
5   else
6     Sleep until awoken by scheduler ;
7   end
8 until ctx is bound to a SPU ;

```

Figure 1. Overview of SPU context activation routine.

Each SPU context has an *activation routine* (see Figure 1) attached to it that is part of its control thread, and is executed whenever the context becomes runnable. The activation routine takes a free SPU from the SPUs list (line 2, Figure 1) and binds its managed context to that SPU (line 4, Figure 1). If there are no free SPUs at the time of the call, the activation routine waits until a SPU is released and then repeats the SPU selection process. When the routine completes, it returns to the caller: the *run loop*. The run loop is in charge of managing callbacks and other events, and is initiated by the *run* operation.

The entity in charge of exiting the wait period (line 6, Figure 1) is the SPU scheduler. This situation happens whenever the scheduler frees a SPU by preempting a currently-running SPU task or when a SPU task voluntarily yields.

```

1 foreach spu do
2   if spu is running a context then
3     ctx ← context in spu ;
4     Decrease timeslice(ctx) by one ;
5   if timeslice(ctx) = 0 then
6     (found, new_ctx) ← find runnable context ;
7     if found then
8       Remove new_ctx from SPU run queue ;
9       Unbind ctx from spu ;
10      Append ctx to SPU run queue ;
11      Mark spu as free ;
12      Wakeup activation of new_ctx ;
13      Wakeup activation of ctx ;
14     end
15   end
16 end
17 end

```

Figure 2. Overview of SPU scheduler tick routine.

The SPU scheduler thread (see Figure 2) is awoken at constant periodic intervals. The routine iterates over the list of all physical SPUs and decreases the time-slice counter (line 4, Figure 2) of the contexts run by the SPUs. If any context time-slice counter reaches zero (line 5, Figure 2), the

scheduler selects a new candidate to replace that context (line 6, Figure 2). If a candidate is found in the SPU run queue, the scheduler unbinds the old context from the SPU, appends it to the run queue, and marks the released SPU as free (lines 8–11, Figure 2). Then, it awakes the activation routine of the selected context, gathers the freed SPU and binds its controlled SPU context.

The activation routine *also* awakes the activation routine of the preempted SPU context (line 13, Figure 2). This is an artifact to force the run loop to call the activation routine again. This fact is critical to understand our analysis in Section IV.

III. METHODOLOGY

We have developed the *Cetra* [5] tool-set to collect information of applications running in the Cell and used version 0.1 in this paper. These tools are mainly targeted to trace the interaction between Cell-based applications and the Linux kernel. The tools included in *Cetra* are classified in two categories:

- The tracing tools, which in the current version of the tool-set only include the `cetra-trace` utility. This tool runs a given application and generates an execution trace for it.
- The analysis tools, which process the raw traces captured by the tracing tools and extract information out of them. In this paper we focus only on `cetra-paraver`, a tool that builds a time-based plot of the states of the threads.

The traces are obtained using the *SystemTap* utility, a free software infrastructure that allows gathering information about a running Linux system [6]; it is very similar to Sun `dtrace` [7]. We have instrumented the kernel code to add markers at multiple key points in the `spufs` module and at other architecture-independent points.

The `cetra-trace` tool generates a *SystemTap* script that hooks a tiny logging routine into the kernel markers we have defined. Each logging routine appends, when called, a single-line stanza to the trace. Each stanza contains: a time stamp, which is the value of the PowerPC time counter register; the CPU on which the marker was triggered; the Process Identifier (PID) and Thread Identifier (TID) of the thread context under which the marker was executed; the marker name; and the variable list of parameters associated to that marker point. These parameters include information that is specific to the marker, such as the SPU context being created or the SPU affected by the operation.

The main events captured by `cetra-trace` are:

- Context switches in the PPU. We hook into the `scheduler.cpu_on` and `scheduler.cpu_off` probe points provided by the standard *SystemTap* tapsets.
- Context switches in the SPUs. We hook into the `bind` and `unbind` operations of the SPU scheduler (`spufs/sched.c` file) to monitor SPU context switching events and their cost.
- Activation of SPUs and stop notifications. These go in the `spufs/run.c` file.

SystemTap adds overhead to the system and thus could distort our scheduling-related traces. This overhead is depreciable

because the tracing process is lightweight and because all the work happens on the PPU. Our workloads barely need any PPU processing power.

The analyses presented in this paper are based on the time-based plots generated by the `cetra-paraver` analysis tool, which are later visualized with Paraver [8]. We use such graphs to study the scheduling decisions performed by both the CFS and the SPU schedulers and how they affect the global execution of our workloads.

To simplify our studies, we use the `loop-bench` microbenchmark, which accurately models the behavior of most SPU-bound Cell applications. This benchmark creates N SPU threads, each of which repeats a computation K times before exiting. This is very close to real-world Cell programs: their main control thread spawns parallelism on the SPUs and then sits idle waiting for the termination of all the SPU tasks. The N parameter is configurable and allows us to run the application with different degrees of parallelism, *including multiprogrammed workloads*.

`loop-bench` does not consider any data preparation or recollection on the PPU because we are only interested in studying the SPU scheduler behavior. PPU bottlenecks would be the basis of another different study. `loop-bench` has no data dependencies between the iterations of the loop performed by each SPU task, and there are no dependencies between the SPU tasks themselves. This models a heavily-loaded multiprogrammed environment where the SPUs do not need to perform any synchronization operation and, therefore, utilize the hardware to its full capacity.

Even though all the study has been simplified using the `loop-bench` microbenchmark for illustrative purposes, we have also made some sample tests with real Cell applications to confirm that the observed results are not biased by the simplicity of the microbenchmark. For example, we analyzed the behavior of the Julia Set application included in the Cell SDK after removing the limitation that prevents it from using more SPU contexts than physical SPUs.

We ran all of our experiments on a PlayStation 3 machine running Fedora 8 with Linux 2.6.24.4 and the most current development version of SystemTap. Linux and SystemTap were modified with the Cetra patchset. The PlayStation 3 only has 6 usable SPUs when running under Linux and has 256 MB of RAM. The results of our experiments are easy to extend to the IBM Cell Blades—which have 2 Cell processors with 8 SPUs each, and higher RAM capacity—due to the low memory requirements of `loop-bench` and its linear scalability. We have done simple measurements to confirm this statement.

IV. ANALYSIS

This section presents the results of our experiments and is organized in the same order we performed them; i.e. from our first goal, which was to quantify the cost of SPU context switching, to how we discovered problems in the scheduling code and diagnosed them.

A. Context switch cost

The first experiment here presented quantifies the cost of SPU task context switching implemented in the SPU scheduler. Switching SPU contexts is a costly operation [9], a fact that could seriously penalize performance in a multiprogrammed environment.

We first show the details of executing the `loop-bench` benchmark with 6 and 12 SPU tasks and with the *FIFO* and *NORMAL* scheduling policies. All the executions in this experiment were made with $K = 500$ million iterations per task. The results are presented in Table I.

Policy	#SPU tasks	Wall time	#CSs	CS time
<i>FIFO</i>	6	15.57s	6	0.001%
	12	31.14s	12	0.0006%
<i>NORMAL</i>	6	15.57s	6	0.001%
	12	42.86s	2263	0.101%

Table I
RESULTS OF TWO DIFFERENT EXECUTIONS WITH THE OLD AND NEW SCHEDULING POLICIES. (CS STANDS FOR CONTEXT SWITCH.)

The context switch cost in all cases is negligible: it is much below 1% of the total execution time. Note, though, that the fourth execution shows two orders of magnitude more context switches than all the others.

As expected, doubling the number of tasks exactly doubles the total execution time for the *FIFO* policy: each task runs to completion and is not disturbed in the process, so the twelve run at maximum efficiency.

However, the results for the *NORMAL* policy are different: the execution with 12 SPU tasks is 2.75 times slower than the one with 6 tasks. Comparing the two scheduling policies, we notice *an extra 37.63% time overhead*. Given that the context switch cost is negligible, all that extra time is spent doing other activities; details follow in Section IV-C.

B. SPU scheduler scalability

In this experiment we study how the SPU scheduling policies scale as we increase the number of concurrent SPU tasks. Figure 3 plots the execution time (Y axis) of `loop-bench` using $K = 1$ billion iterations and the number of N SPU tasks varying from 1 to 36 (X axis).

The stair-case effect shown by the *FIFO* policy is because all tasks in each execution are doing the same amount of work and have no data dependencies among them. Therefore, the scheduler always schedules the N contexts in groups of 6. The last remaining $N \bmod 6$ contexts of a given execution ($6 = \#SPUs$), are also scheduled as a full group, and thus they do not use all physical SPUs at once.

The *NORMAL* policy shows an almost-linear scalability. Ideally, if context switching had no negative performance impact, all points on the *NORMAL* line could be under the *FIFO* one and the scalability could be strictly linear. However, context switching costs add extra run time, and therefore it is reasonable for some points on the *NORMAL* line to be above their *FIFO* counterparts. This is effectively the case.

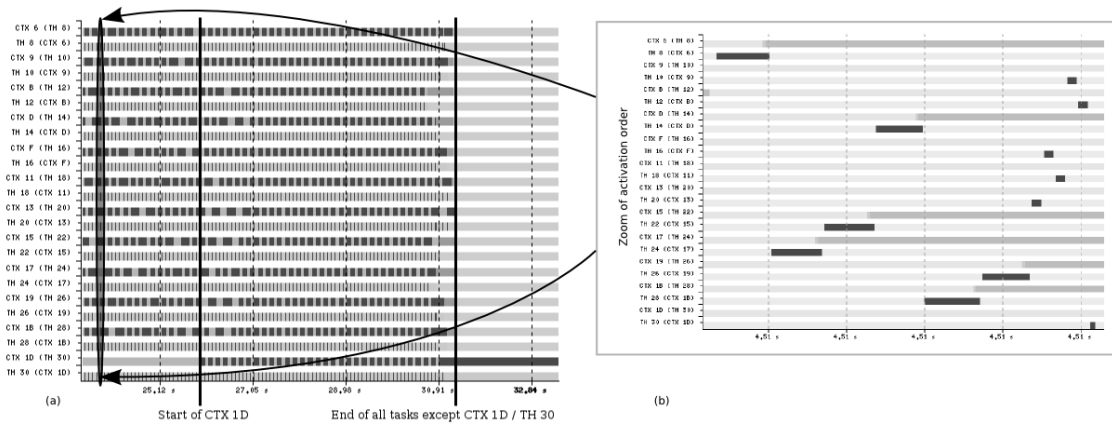


Figure 4. Time-based plot of the state-changes suffered by the SPU tasks and the control threads of `loop-bench`, showing an starvation problem (a) and the activation order pattern (b) that causes it.

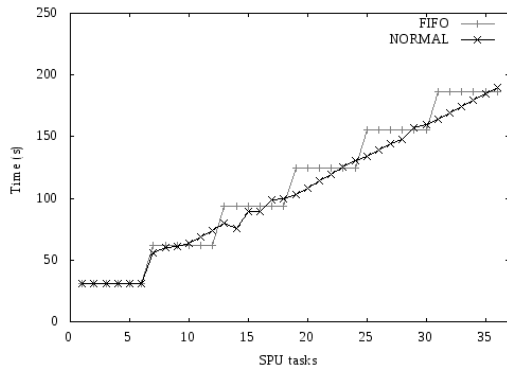


Figure 3. Wall time of the `loop-bench` application with different amounts of SPU contexts.

Note also that the *NORMAL* executions with 11 and 12 SPU tasks have significantly more overhead with respect to the *FIFO* policy when compared to the executions with, for example, 29 and 30 SPU tasks.

C. SPU task starvation

We show a time-based state-change plot of the tasks created by `loop-bench` with $N = 12$ SPU tasks and the *NORMAL* policy using Paraver. The goal of this experiment is to identify the source of the slowdown mentioned in section IV-A.

Figure 4a depicts a different horizontal bar for each SPU task and control thread and was obtained using $K = 100$ million and $N = 8$. The labels on the left specify whether the bar corresponds to a SPU task (CTX) or a control thread (TH); the text in between the parenthesis indicates the CTX-TH coupling. The X axis shows time in seconds.

The shades of gray of each bar specify a specific state for the thread or SPU task. The darkest gray indicates that the thread is running and thus consuming processor time. The thin gray regions in between dark gray regions indicate, for TH bars, that the thread is blocked and, for CTX bars, that the task has been unbound from the SPU and is thus waiting to be scheduled again. The light gray at the very beginning of CTX

1D indicates that the task is waiting for a physical SPU to be freed (stalled in the activation routine). Finally, the lightest gray at the end of each bar indicates that the thread or task has finalized its execution.

The delayed start of CTX 1D explains the 35% wall-time overhead of this experiment: this SPU task does not get any SPU time until a very late point in time, which causes it to run alone for many seconds. Ideally, this delayed task should have executed concurrently with the other tasks, but it is not considered for execution until after the 26th second. We consider this to be a *starvation problem* that *penalizes the last spawned SPU task*.

However, note that TH 30, which is in control of CTX 1D, is scheduled many times before CTX 1D is finally bound. This is because the SPU scheduler correctly wakes it up to run the activation routine. Unfortunately, the activation routine fails to locate a free SPU because other activation routines have already stolen all free SPU. This is partly caused by line 13 of Figure 2: the preempted context can be immediately rebound again to a SPU if its control thread is scheduled *before* the control thread of the new context is awoken in line 12.

Figure 4b shows a closeup view of a very narrow time window of the trace in Figure 4b. The running periods of the control threads are highlighted. Those control threads with long running periods are the ones that find a free SPU and spend time issuing a *bind* operation; those with short running periods are the ones that fail to locate a free SPU. We notice that CFS schedules the activation routine (TH 30) of the last spawned SPU task (CTX 1D) *after* the activation routines of all others.

This execution pattern repeats many times until about the 26th second of the execution (see Figure 4b). At that point, the scheduling decision performed by CFS changes and the penalized thread (TH 30) is executed before some others (not all). Due to this, TH 30 is finally able to bind its context to a free SPU.

We have retried this experiment with several different values of N , for $N > 6$, and always see that the last spawned SPU task suffers from this start-up delay. Additionally, we have also performed this same experiment with the Julia Set application

and observed a huge performance penalty: with 6 SPUs it just takes 10 seconds to run and with 12 SPUs it takes 480 seconds.

D. Accumulated run time

Section II-C described the scheduler tick routine and how it awakes the activation routines to let themselves bind their own contexts to free SPUs. The problem of this approach is that CFS is the final responsible for selecting which of the awoken control threads gets the PPU first. If the awoken threads are executed in an incorrect order, the SPUs end up executing tasks in a different order than the one planned by the SPU scheduler. This incorrect wake up ordering is what exposes the starvation problem described above.

This problem is caused by the extremely short run-time of the controlling threads every time they are scheduled to run the activation routine. Their *vruntime* is so small that, when CFS makes the decision of which thread to run, its policy does not make the right choice because all threads have very small and similar *vruntime* values.

The behavior of the control threads, as seen by CFS, is the same as interactive tasks: short run times and extremely long wait times. Control threads, if considered altogether with their SPU tasks, should really be treated as computationally-intensive threads.

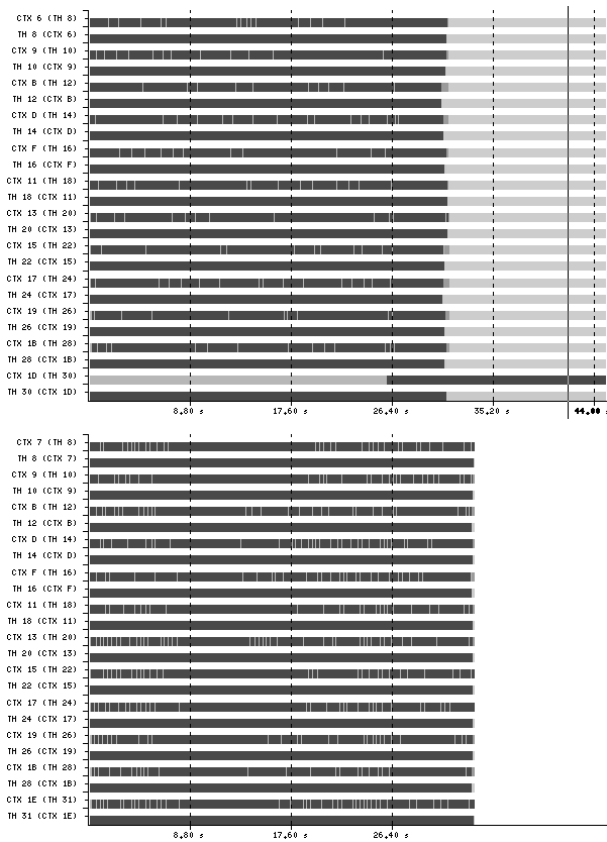


Figure 5. Time-based plot of the state-changes suffered by the SPU tasks and the control threads of loop-bench, comparing the starvation problem case with the ideal one.

We confirmed this problem by adding a short-lived tight loop in the activation routines. The loop has the effect of

adding a noticeable *vruntime* to the caller and tricks CFS into correctly ordering them. This change fixes the starvation problem as shown in Figure 5. Note the significant wall-time difference between the two cases.

V. PROPOSED SOLUTIONS

There are many workarounds for the starvation problem that involve tuning the CFS debugging parameters. All these parameters have the same underneath effect, which is the modification of the CFS next-thread selection algorithm either by changing the policy itself or by modifying the *vruntime* of the threads.

For example, it is possible to obtain good results by lowering the value of the `kernel.sched_latency_ns` parameter by a factor of 40 at the expense of more PPU scheduling load. Another possibility is to enable and/or disable specific features by tuning the `kernel.sched_features` parameter. We found that enabling the `APPROX_AVG` feature or disabling `NEW_FAIR_SLEEPERS` or `START_DEBIT` have similar effects.

However, we must consider the fact that *only the last* spawned SPU task is the one that suffers the starvation problem. This makes us believe that there is an *off-by-one bug in CFS* that causes it to make bad scheduling decisions for interactive tasks.

A solution to this problem could be to rewrite the SPU scheduler to make it independent from CFS decisions. This way we could avoid this specific problem and any other potential misbehaviors caused by incorrect interactions between the two completely-independent schedulers.

A different approach to resolve the problem involves accounting the execution time of SPU tasks to their control threads. This could make CFS see them as computationally-intensive processes and then could automatically make a correct decision at all times. Doing this change could modify the semantics of the abstraction for SPU and therefore should be carefully considered.

VI. CONCLUSIONS

Thorough this paper, we have presented the current Linux execution model for parallel Cell applications and how the two Linux task schedulers —CFS and the SPU scheduler— do not interact well with each other. This lack of synchronization degrades the performance of multiprogrammed SPU environments. We have provided a diagnosis for the problem and have proposed ways in which it can be worked-around or fixed.

We plan on implementing new Cell scheduling policies ourselves to improve SPU scheduling, first by fixing the starvation issue and second by lowering the context switching overhead even more.

Additionally, we have also presented Cetra, the tool-set we are developing to trace and analyze the interactions of Cell applications with the Linux kernel. These tools allow us to obtain detailed execution information that help us in identifying performance bottlenecks and misbehaviors in all the parallel components of a Cell application.

REFERENCES

- [1] C. R. Johns and D. A. Brokenshire. Introduction to the Cell Broadband Engine Architecture. *IBM Journal of Research and Development*, 51(5), 2007.
- [2] Arnd Bergmann. Linux on Cell Broadband Engine status update. In *Proceedings of the Linux Symposium*, 2007.
- [3] Ingo Molnar. *Scheduler design CFS*, 2007. Linux 2.6.24.4 kernel documentation.
- [4] Arnd Bergmann. *Spufs: The Cell Synergistic Processing Unit as a virtual file system*, 2005. IBM DeveloperWorks.
- [5] Cetra website. <https://gso.ac.upc.edu/projects/cetra/>.
- [6] Frank Ch. Eigler et al. Architecture of SystemTap: a Linux trace/probe tool. 2005.
- [7] Sun Microsystems. *Solaris Dynamic Tracing Guide*, 2005. Part No. 817-6223-11.
- [8] CEPBA. *Paraver – Parallel Program Visualization and Analysis tool – REFERENCE MANUAL*, 2001.
- [9] IBM. *Cell Broadband Engine – Programming Handbook*, chapter SPE Context Switching. IBM, 2006.