Improving a CellFS Implementation for the x86 Architecture

Pthread Coroutine Support

P. Jorrit Tijben
p.j.tijben@student.utwente.nl

ABSTRACT

With multi-core processors becoming mainstream, programmers are trying to take full advantage of their parallel possibilities. One of the relatively new and powerful multi-core CPUs is the Cell Broadband Engine Architecture, which excels at floating point calculations. To ease development on this platform suitable for supercomputing, frameworks have been created offering a range of programming models. One of these frameworks is CellFS. There has been progress on making CellFS available for the x86 architecture, providing a way to test and run code without the need for actual Cell hardware. CellFS on x86 supports coroutines using a straightforward implementation based on setcontext/getcontext. A different and perhaps more promising way to realize coroutines is using pthreads; the possibility and feasibility of such a solution is investigated.

Keywords

Coroutines, CellFS, Cell Broadband Engine Architecture, x86, pthreads, concurrency

1. INTRODUCTION

When real hardware is not available, or already in use, building applications for the Cell processor can be done by means of a virtualized environment, for example with IBM's Full-system simulator [3] as part of its Software Development Kit.

Either on the real hardware or within the virtualized machine, CellFS [15] can make development easier. It is a framework, borrowing concepts and programming models from Plan 9 [23] to simplify I/O between individual processing units and the main memory of the Cell CPU. Mols [21] has implemented the CellFS part dealing with these memory transfers for *x86*; this implementation makes it possible to develop and test CellFS based algorithms on more common hardware. A finished algorithm can then be compiled and tested for the Cell architecture in its unmodified form.

Part of CellFS is its coroutine model. Coroutines provide a convenient way to implement cooperative tasks and will be explained in the background section of this paper. The current implementation of this model for CellFS x86 is based on setcontext/getcontext: C library functions used for context control. Providing a coroutine model based on pthreads may be more portable however. Pthreads are POSIX threads [2], a standardized API for creating and using threads.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission.
11th Twente Student Conference on IT, Enschede 29th June, 2009
Copyright 2009, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science

For Windows there seems to be one user-level implementation of the setcontext/getcontext functions, based on Win32 threads [25]. Because this implementation is thread-based, creating a pthread coroutine solution should be very attainable. While the concept of pthread-based coroutines need not necessarily be contained within CellFS, it will get first attention.

After dealing with background information about the Cell BE architecture, CellFS and coroutines in section 2, further relevance and implications will be discussed in section 3. Section 4 describes the thesis statement and research goals, followed by the approach/methodology in part 5. Results can be found in section 6. Finally, conclusions and a discussion constitute part 7.

2. BACKGROUND

An article by Gschwind and others [12] provides a good overview of the Cell BE architecture, and also the open source environments surrounding it. The Cell Broadband Engine Architecture consists, in its standard configuration, of a Power Processor Element (PPE) and several Synergistic Processing Elements (SPE). All the SPEs and the PPE are connected by the Element Interconnect Bus (EIB). The PPE is a 'normal' Power architecture processor and is the main processor controlling the SPEs. SPEs are optimized for floating point operations. They contain a Synergistic Processing Unit (SPU) which uses SIMD operations, and a Memory Flow Controller (MFC). See figure 1 for an overview.

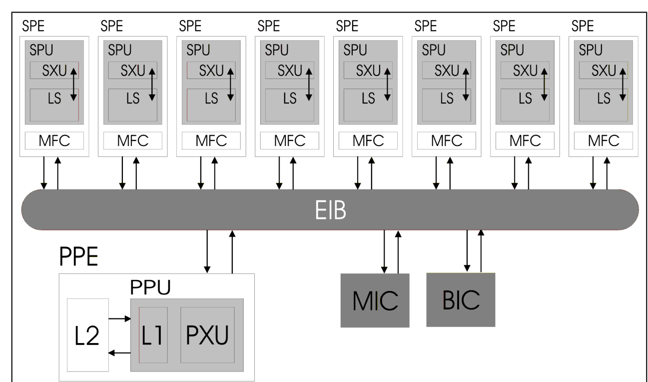

Figure 1: Cell BE Architecture

The SPU cannot directly communicate with the system memory, but memory transfers are handled by the MFC which uses Direct Memory Access to access the system's memory. This is where CellFS comes in. Ionkov et al. [15] describe the design and implementation of CellFS: a library that abstracts memory transfers to and from an individual SPE which would otherwise be done

manually.

As stated in the introduction, CellFS also makes use of coroutines. Conway [6] first coined the term in 1963, naming an idea by him and Joel Erdwinn. Coroutines are autonomous parts of program code that have no fixed order of execution. They can suspend and resume at different points in the code. Knuth [17] sees them as a generalized form of subroutines. This generalization is expressed by thinking of coroutines as cooperating equals, while subroutines have a clear 'caller' and 'callee' relationship. The most crucial difference then, is that a subroutine always kicks off at the first address of its implementation code, while coroutines instead *continue* at the point where the code last yielded. Yielding implies stopping the current routine prematurely, allowing another coroutine to run instead. Its local environment will be preserved; upon returning to the first coroutine the program will continue at this preserved point.

A simple example:

```
/* This is the coroutine. */
void count()
{
  int i = 0;

  printf("%d\n", i);
  i++;
  yield();
  printf("%d\n", i);
  i++;
  yield();
  printf("%d\n", i);
  i++;
}

int main(int argc, char* argv[])
{
  count();
  count();
  count();
}
```

Would produce:

```
0
1
2
```

Assuming that `count()` is a coroutine implementation for C, yielding would transfer control back to `main()` while at the same time saving `count()`'s stack. When `count()` is called again, it will restore the context from the last yield and continue from that point.

The example above has a somewhat limited use. Because of the inherent concurrent nature of coroutines, a better and perhaps a more useful example is the well-known producer-consumer problem:

```
int queue[10] = {0};

void producer()
{
  for (;;)
  {
    /* While queue not full. */
    while(queue[10] != 0)
    {
      produce(queue);
    }
    yield();
  }
}

void consumer()
{
  for (;;)
  {
    /* While queue not empty. */
    while(queue[0] != 0)
    {
      consume(queue);
    }
    yield();
  }
}
```

Although the earlier explanation of coroutines states that they have no fixed order of execution and therefore run concurrently, a specific implementation *could* allow to transfer back control to a calling function as in the first listing. Another explanation of coroutines along with good examples is given by De Moura and Ierusalimschy [8].

3. RELEVANCE AND IMPLICATIONS

Programming for the Cell processor, or 'parallel programming' in general, has a reputation of being hard [22, 1]. This is the main reason several frameworks exist that may provide more convenient interfaces to the programmer. The IBM redbook[1] about programming the Cell processor lists several frameworks [14], CellFS being one of them. Frameworks often involve a trade-off between performance and the level of abstraction. Because a higher level of abstraction mostly implies more ease of use, performance and simplicity can be seen as conflicting goals.

Apart from simplifying programming on just the Cell BE platform, higher abstractions can also increase the potential to run programs using these abstractions on other architectures. Because the programs do not make use of architecture-dependent code, individual implementations of frameworks can be written for the different architectures.

Frameworks like CellFS thus provide programming models *and* the possibility to compile, run, and test code on different machines. Programming concepts which are straightforward to use are relevant to the programmer to save time. Coroutines are part of the CellFS model but are currently implemented in two different ways: using custom stack management for the Cell processor, or by means of the already mentioned setcontext/getcontext as used by Mols in his x86 'port'. As stated however, the major drawback of using setcontext/getcontext is that it is not available in many x86 environments. Implementations and usage of pthreads are more widespread, so it seems logical to map a coroutine to a separate thread. These are the main reasons for a coroutine implementation bases on pthreads.

Using pthreads to implement coroutines may seem a bit redundant. After all, threads and coroutines both allow running two or more concurrent tasks. The major difference between coroutines and threads however, is that threads are preemptive while

[1]IBM Redbooks are books of a technical nature, developed and published by IBM's International Technical Support Organization (ITSO).

coroutines explicitly suspend themselves.

One could imagine the concurrent nature of coroutines allows for parallel execution on different cores. While this is true, CellFS coroutines run mutually and concurrently, but *not* in parallel on a *separate* SPE. This means that coroutines on one SPE are independent from the coroutines on another SPE; they cannot 'see' each other. The different groups of coroutines of all SPEs will, however, run in parallel.

4. THESIS STATEMENT

The project has several objectives in view. The first and primary goal is to make a coroutine implementation that is based on pthreads. This coroutine mechanism will be part of the CellFS library and should run on both the x86 and Cell BE architecture. A comparison of both design and performance is made with the use of POSIX setcontext/getcontext for coroutines. Other coroutine implementations could also be evaluated.

In the end, adding a portable coroutine implementation to CellFS serves a greater goal. This greater goal consists of compiling and running code transparently on both the Cell BE and x86 architecture. Optimizing CellFS in other ways should also contribute to making programs run on x86 as well as Cell BE.

A research question could be formulated as follows:

> Is it possible to substitute the setcontext/getcontext coroutine support of CellFS with a solution that makes use of pthreads, and if so, does this result in a better alternative?

5. APPROACH/METHODOLOGY

5.1 Initial concepts

Using pthreads to realize coroutines requires at least the following things:

- Finding a way to transfer control to other coroutines, i.e. 'bypassing' the thread scheduler. This can probably be done by means of a general lock for all coroutines on a single SPE.
- A lock for each shared critical section/data structure when the coroutines run concurrently.
- Taking blocking and fairness into account. What happens if one coroutine blocks on I/O?
- Considering CellFS' short API and adapting the implementation to it.
- Take pthread's own stack size into account.

Comparing a pthreads implementation with the original solution should be made based on a few criteria: Portability, difficulty to program, ease of use, correctness, and performance (speed). Performance is evaluated in section 6 by running benchmarks; but only for a sample program outside CellFS.

Initial coroutine functionality was designed apart from CellFS. A paper design and a functional, but separate implementation were the first to concentrate on the problem and ignore details required by CellFS. This initial test program is much like the ascending/descending example used in section 6. With a separate lock for every coroutine, execution can be limited to one coroutine at a time. Shared code should also be protected by locks but for now this is left to the user.

A solution is also required for the fact that CellFS' API expects a custom stack when creating a coroutine.

5.2 Eventual Solution

By using the pthreads library for a coroutine implementation, there is, conveniently, already an private stack and some other context per thread. Trying to make something other than a one-to-one mapping between a single coroutine and a single pthread would therefore seem illogical.

The first problem that arises is the fact that every pthread has to be suspended whenever a `yield()` function call is done by the user. A way has to be found to either adapt or bypass the standard pthread scheduler.

The only 'native' possibilities to make changes to the scheduling order is by making use of `pthread_attr_setschedpolicy()` and the associated function `pthread_attr_setschedparam()`. The different alternatives are SCHED_OTHER, SCHED_RR and SCHED_FIFO for priority scheduling, a round-robin scheduling and a first in, first out scheduling, respectively. However there is no option to schedule to another thread at any given time, and as such to make the library non-preemptive/semi-preemptive.

It becomes clear that it is both an advantage as well as a disadvantage to use pthreads. The advantage is having a cheap private context, but at the cost of sacrificing the possibility to yield a thread at chosen but arbitrary moment. The only two options that remain are:

1. Adapting the pthread scheduler itself. This can be relatively tricky because some implementations make use of system calls and/or interact with the kernel in other ways. Some userspace implementations, for example the GNU Portable Threads [10], do provide non-preemptive functions (like `pth_yield()`). But using these would not be portable thus defeating the whole purpose of using pthreads in the first place.
2. Suspending the threads by using all means of pthreads itself, like mutexes and condition variables.

Because making changes to the pthreads scheduler seems overly complex when a solution with mutexes and condition variables suffices, and is not even portable, only the second option will be implemented.

The solution is to block a thread with condition variables in order to guarantee only one thread at a time will be active and executing code. When coroutines are used by the main program, first a mutex and an accompanying condition variable will be created. Making a coroutine comes down to calling `pthread_create()`, after which it is immediately locked on the condition variable in its entry routine.

```
1 void* coroutine(void* unused)
  {
3   pthread_mutex_lock(&cmutex);
    while (control != cornr)
5   {
      pthread_cond_wait(&sequence, &cmutex);
```

```
    }
    pthread_mutex_unlock(&cmutex);

    ...
```

The global variable control is used to tell which coroutine is going to be scheduled; cornr represents the number/id of the current, 'own', coroutine. When it is not this coroutine's turn it blocks on the condition variable.

Thus, scheduling a new coroutine takes place by assigning a new value (a new thread number) to the condition control and by making use of pthread_cond_broadcast() to stop the target thread from blocking and allowing it to continue.

To incorporate this method within CellFS, work was based off the setcontext/getcontext implementation: functions were reduced to stubs, after which they were implemented with pthread-specific lines. Some code, like the simple round-robin scheduler, could be reused. All required functionality is put in a single file (cor.c) in which the following functions are important, and call each other in the following way (see figure 2 for an overview):

main() is nothing more than the beginning of the program but should not be seen as a separate coroutine. It will instantly create a main coroutine by calling mkcor(). This is essentially a wrapper around pthread_create(). Subsequently the helper function mkcor_aux() is called which locks on the condition variable. When this condition gets a signal, control is transferred to the earlier supplied coroutine method,

Whenever yield() is called by one of the coroutines, sched() is executed to schedule a next coroutine. sched() chooses the next one on a round-robin basis and is the function that signals/broadcasts the coroutines blocked on a condition variable.

There is an advantage of pthreads versus setcontext/getcontext that showed up while implementing. Setcontext/getcontext can only switch to a function that has integer parameters. Using pthreads, pointers or in fact every type is allowed, allowing more flexible code.

Internally, CellFS expects it is given an initial stacksize for a coroutine. Because every pthread has a default stacksize, this value can be ignored. But, every pthread library does have a function to set the stacksize too, so there is no reason *not* to use the stacksize information. Even more, the default stacksize is unspecified and implementation-specific: relying on a default stacksize would be *less* portable.

6. RESULTS

6.1 Speed in Theory

Before any benchmarks are done, one might reason about the speed of the setcontext/getcontext and pthreads implementations. For Linux, the GNU libc [11] library is very common and it provides implementations of setcontext, getcontext ..., as well as a pthreads library: Native POSIX Thread Library. With version 2.1, makecontext, setcontext, getcontext and swapcontext are implemented in assembly for the x86 architecture and can be found in /sysdeps/unix/sysv/linux/i386/. The NPTL library can be found in /ntpl. While speed tests should be done to be sure, one can imagine that utilizing the setcontext/getcontext functions is a more efficient than using pthreads for context switching. A pthread simply has more overhead and is capable of doing more than only switching context. However, creating eight threads (the current CellFS maximum but this could be changed) on a modern Linux system should be negligible, and would not really be a disadvantage to setcontext/getcontext. For Windows testing, the POSIX Threads for Win32 [16] library will be used but the pthread support in Microsoft Windows Services for UNIX [20] could be a good alternative.

6.2 Simple Benchmark

To measure the speed differences between the solutions, two programs which do exactly the same are tested. They consist of the functions ascending and descending, both running a counter 50,000,000 times. These functions alternate between each other by means of setcontext/getcontext for the first program:

```
#include <stdio.h>
#include <stdlib.h>
#include <ucontext.h>

ucontext_t ascContext, descContext, mainContext;
void ascending(void);
void descending(void);

int main(int argc, char* argv[])
{
  int isSwapped;
  char ascStack[SIGSTKSZ],
  char descStack[SIGSTKSZ];

  getcontext(&ascContext);
  ascContext.uc_link = &mainContext;
  ascContext.uc_stack.ss_sp = ascStack;
  ascContext.uc_stack.ss_size =
    sizeof(ascStack);
  makecontext(&ascContext,
              (void (*)(void)) ascending, 0);

  getcontext(&descContext);
  descContext.uc_link = &mainContext;
  descContext.uc_stack.ss_sp = descStack;
  descContext.uc_stack.ss_size =
    sizeof(descStack);
  makecontext(&descContext,
              (void (*)(void)) descending, 0);

  isSwapped = 1;
  getcontext(&mainContext);
  isSwapped = 0;

  printf("Switching from main to ascending\n");
  if (!isSwapped)
  {
    swapcontext(&mainContext, &ascContext);
  }
  printf("Exiting program\n");
  exit(EXIT_SUCCESS);
}

void ascending(void)
{
  int i;
  for (i = 0; i < 50000000; i++)
  {
    printf("Ascending: counter is '%d'\n", i);
    printf("Switching from ascending"
           "to descending\n");
    swapcontext(&ascContext, &descContext);
  }
}

void descending(void)
{
```

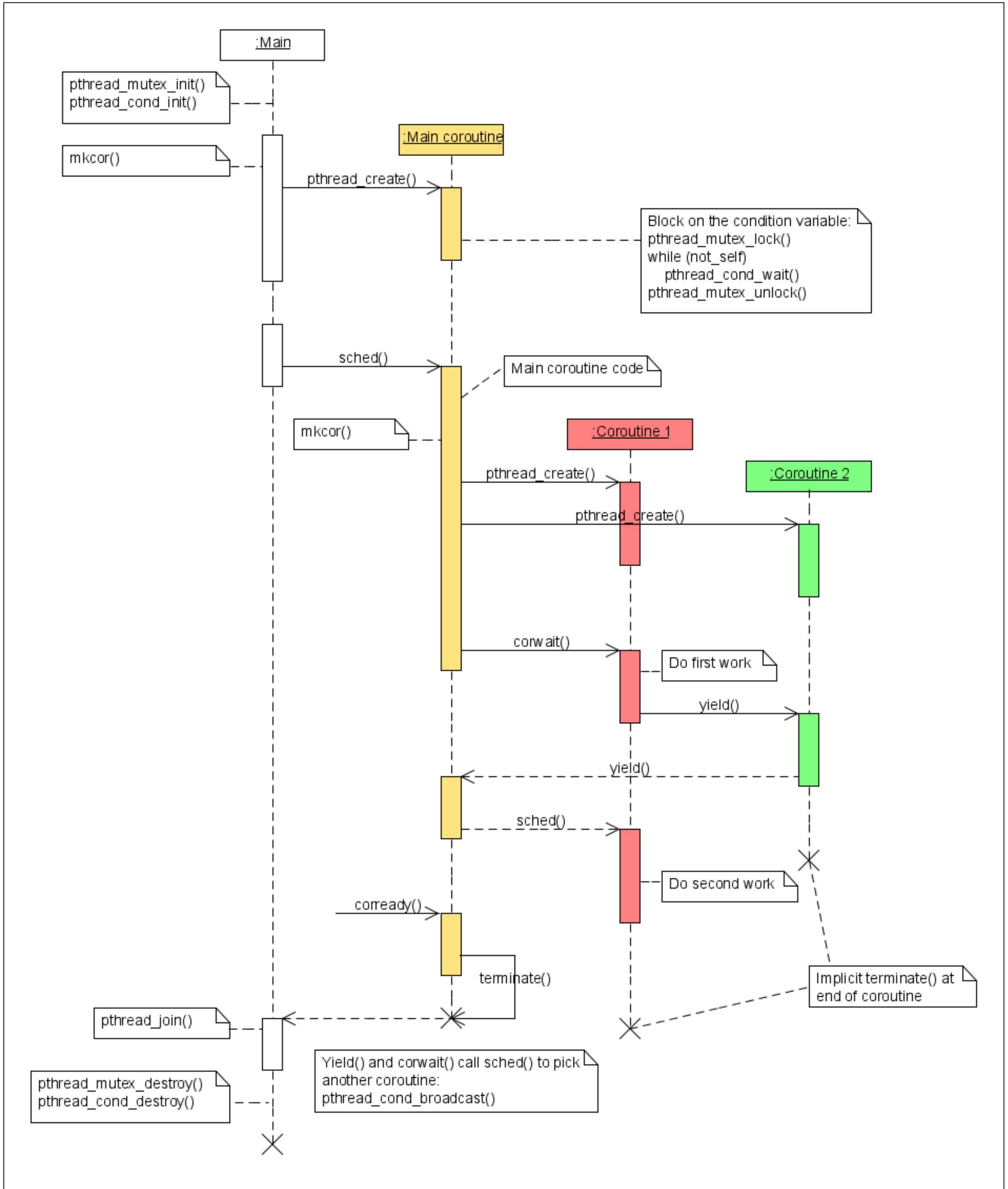

Figure 2: Creation and usage of two worker threads

```
    int i;
    for (i = 50000000; i > 0; i--)
    {
      printf("Descending:␣counter␣is␣'%d'\n", i);
      printf("Switching␣from␣descending"
             "to␣ascending\n");
      swapcontext(&descContext, &ascContext);
    }
}
```

`time pthread > /dev/null` and
`time setcontext > /dev/null`

Output of the `printf`s is redirected to /dev/null because it isn't particularly interesting. Results of the the elapsed real time between invocation and termination are displayed in table 1.

Table 1: Program timings

setcontext time (s)	pthread time (s)
58.377	469.841
55.791	414.312
56.387	484.965
55.351	508.893
76.688	485.916
61.588	504.488
63.668	463.366
63.685	453.490
56.539	489.556
55.763	482.995

Table 2: Statistical values

	setcontext	pthread
Avg. (s)	60.3837	475.7822
Var. (s^2)	43.50219446	757.12170529
Std. dev. (s)	6.59561934	27.51584462
Std. dev. (%)	10.92284729	5.78328584

The following example does exactly the same as the setcontext/getcontext program but uses pthreads, mutex locks and condition variables to achieve the same alternation between ascending and descending. Both programs don't have error checking etc. to save some space. The tests ran on system with a dualcore processor to potentially make use of parallelism, with Linux x86_64 SMP 2.6.24.5.

```

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int main(int argc, char* argv[]);
void* ascending(void*);
void* descending(void*);

const int ascControl = 0;
const int descControl = 1;
int control;

pthread_t pAsc, pDesc;
pthread_mutex_t cmutex;
pthread_cond_t sequence;

int main(int argc, char* argv[])
{
  int status;
  control = -1;

  pthread_mutex_init(&cmutex, NULL);
  pthread_cond_init(&sequence, NULL);

  pthread_create(&pAsc, NULL, ascending,
                 NULL);
  pthread_create(&pDesc, NULL, descending,
                 NULL);

  printf("Main␣method\n");

  pthread_mutex_lock(&cmutex);
  control = ascControl;
  pthread_cond_signal(&sequence);
  pthread_mutex_unlock(&cmutex);

  pthread_join(pAsc, NULL);
  pthread_join(pDesc, NULL);

  pthread_mutex_destroy(&cmutex);
  pthread_cond_destroy(&sequence);
  pthread_exit(NULL);
  return 0;
}

void* ascending(void* unused)
{
  int i;
  for (i = 0; i < 50000000; i++)
  {
    pthread_mutex_lock(&cmutex);
    while (control != ascControl)
    {
      pthread_cond_wait(&sequence, &cmutex);
    }
    pthread_mutex_unlock(&cmutex);

    printf("Ascending:␣counter␣is␣'%d'\n", i);
    printf("Switching␣from␣ascending"
           "to␣descending\n");

    pthread_mutex_lock(&cmutex);
    control = descControl;
    pthread_cond_signal(&sequence);
    pthread_mutex_unlock(&cmutex);
  }
}

void* descending(void* unused)
{
  int i;
  for (i = 50000000; i > 0; i--)
```

```
74    {
        pthread_mutex_lock(&cmutex);
76      while (control != descControl)
        {
78        pthread_cond_wait(&sequence, &cmutex);
        }
80      pthread_mutex_unlock(&cmutex);

82      printf("Descending: counter is '%d'\n", i);
        printf("Switching from descending"
84             "to ascending\n");

86      pthread_mutex_lock(&cmutex);
        control = ascControl;
88      pthread_cond_signal(&sequence);
        pthread_mutex_unlock(&cmutex);
90    }
  }
```

Ideally this test program should be run and timed on several machines, environments and workloads. The goal of these benchmarks however, is to get a general impression of the relative speed of pthreads versus setcontext/getcontext. and to test whether significant differences arise. Having very limited and controlled environments would be more correct but overkill in this case. Doing 10 runs on each machine gave a standard deviation in percentages of 11% (setcontext/getcontext) and 6% (pthreads). Because the speeds of running the examples varies some magnitudes this is sufficient for a general conclusion.

7. CONCLUSION

To answer the question whether a pthread-based coroutine solution within CellFS-x86 is a better alternative to the current implementation, the results need to be verified.

First of all, a pthread implementation is indeed possible, as a working example is realised. Portability of this solution also appeared to be greater than the setcontext/getcontext solution. There exist more, and more full-fledged, pthread libraries than is the case for setcontext/getcontext. Additional difficulties do not appear because the CellFS API remains unchanged.

As mentioned in the results, pthreads can run routines with arbitrary arguments and return values, while for setcontext/getcontext only integers can be used which might be a great drawback. The fact that pthreads are about eight times slower makes a final answer somewhat more difficult. Because CellFS for x86 would mainly be used to test algorithms and sample programs, this slowness is not critical. The final program will eventually be run on the Cell processor anyway.

Unless you are testing some time-critical applications on an x86-machine, the pthread solution is indeed a better alternative.

Recommendations for further research consist of the following points:

- Benchmarks of the implementation with sample programs for CellFS.
- Evaluations against other, standalone coroutine solutions. SystemC [4] has one based on pthreads too, but is C++ and SystemC specific. Several C-based open source coroutine libraries using other options like setcontext/getcontext or setjmp: libtask [7], libpcl [19], coro [24], libcoroutine [9] and libcoro [18].
- Model checking using tools as SPIN [13] or Uppaal [5] could help to verify the solution.

REFERENCES

[1] Sega saturn. *Next Generation Magazine*, page 43, February 1995.

[2] Portable operating system interface (posix). *ISO/IEC 9945-1: 2003 (IEEE Std. 1003.1: 2001)*, Part 1: Base Definitions, 2001.

[3] IBM full-system simulator for the cell broadband engine processor: A full-system simulation infrastructure and tools for the cell broadband engine processor. `http://www.alphaworks.ibm.com/tech/cellsystemsim` [Last checked: March 25, 2009], November 2005.

[4] IEEE Standard SystemC Language Reference Manual. *IEEE Std 1666-2005*, pages 1–423, 2006.

[5] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal — a tool suite for automatic verification of real–time systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, October 1995.

[6] Melvin E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408, 1963.

[7] Russ Cox. Libtask: a coroutine library for c and unix. `http://swtch.com/libtask/` [Last checked: March 25, 2009].

[8] Ana L. de Moura and Roberto Ierusalimschy. Revisiting coroutines. Technical Report 15/04, PUC-Rio, Rio de Janeiro, RJ, June 2004.

[9] Steve Dekorte. Libcoroutine: a portable coroutine implementation. `http://www.dekorte.com/projects/opensource/libcoroutine/` [Last checked: March 25, 2009].

[10] Ralf S. Engelschall. Gnu pth - the gnu portable threads. `http://www.gnu.org/software/pth/` [Last checked: June 15, 2009].

[11] Free Software Foundation. Gnu c library - gnu project - free software foundation. `http://www.gnu.org/software/libc/` [Last checked: June 15, 2009].

[12] M. Gschwind, D. Erb, S. Manning, and M. Nutter. An open source environment for cell broadband engine system software. *Computer*, 40(6):37–47, June 2007.

[13] G.J. Holzmann. The model checker spin. *Software Engineering, IEEE Transactions on*, 23(5):279–295, May 1997.

[14] IBM Redbooks. *Programming the Cell Broadband Engine Architecture: Examples and Best Practices*, chapter 3.1.4, pages 41–27. Vervante, 2008.

[15] L. Ionkov, A. Nyrhinen, and A. Mirtchovski. Cellfs: Taking the "dma" out of cell programming, apr 2007.

[16] Ross Johnson. Posix threads (pthreads) for win32. http://sourceware.org/pthreads-win32/ [Last checked: June 15, 2009].

[17] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*, section 1.4.2. Addison-Wesley, Reading, Massachusetts, third edition, 1997.

[18] Marc Lehmann. Libcoro. http://software.schmorp.de/pkg/libcoro.html [Last cheched: March 25, 2009].

[19] Davide Libenzi. Portable coroutine library. http://www.xmailserver.org/libpcl.html [Last checked: March 25, 2009].

[20] Microsoft TechNet. Pthread support in microsoft windows services for unix version 3.5. http://technet.microsoft.com/en-us/library/bb463209.aspx [Last checked: June 15, 2009].

[21] A.H. Mols. A cellfs implementation for the x86 architecture. http://referaat.cs.utwente.nl/new/paper.php?paperID=444 [Last checked: March 2, 2009], jun 2008.

[22] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2 sub edition, August 1997.

[23] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *j-COMP-SYS*, 8(3):221–254, Summer 1995.

[24] E. Toernig. Coro. http://www.goron.de/%7Efroese/coro/ [Last checked: March 25, 2009].

[25] Xdoukas. Unix ucontext_t operations on windows platforms. http://www.codeproject.com/KB/threads/ucontext.aspx [Last checked: March 25, 2009], May 2007.