When Do Real Time Systems Need Multiple CPUs?

Paul E. McKenney
IBM Linux Technology Center
15350 SW Koll Parkway, Beaverton OR 97006 USA
paulmck@linux.vnet.ibm.com

Abstract

Until recently, real-time systems were always single-CPU systems. The prospect of multiprocessing has arrived with the advent of low-cost and readily available multi-core systems. Now many RTOSes, perhaps most notably Linux™, provide real-time response on multiprocessor systems.

However, this begs the question as to whether your real-time application should avail itself of parallelism. Furthermore, if the answer is "yes," the next question is what form of parallelism your application should avail itself of: shared memory parallelism with locking and threads, process pipelines, multiple cooperating processes, or one of a number of other approaches.

This paper will examine these questions, providing rules of thumb to help you choose whether your real-time application should be parallel, and, if so, what sort of parallelism is best for you.

1 Introduction

Moore's-Law-induced increases in single-threaded performance ended in the year 2003, as can be seen in Figure 1 [8]. It is perhaps unsurprising that the first multicore version of that quintessential embedded CPU, ARM, was announced just one year later. This figure plots clock frequency for recent CPUs capable of retiring at least one instruction per cycle, and Dhrystone MIPS for older CPUs. This sudden cessation of exponential single-threaded hardware performance growth has motivated increased investigation, development, and deployment of systems exploting parallel processing, including in the real-time arena. Furthermore, parallel hardware is now low cost and readily available, which means that parallelism has become an attractive way to increase performance and decrease response times. Finally, a parallel real-time Linux kernel is available in the form of the -rt patchset [9].

Unfortunately, most software is still single-threaded, and, worse yet, numerous workloads are reputed to lack efficient and scalable parallel implementations. Even for workloads that do have efficient and scalable parallel implementations, these parallel implementations are often larger, more complex, and much more difficult to validate than are their single-threaded counterparts. In short, although parallel hardware is low cost and readily available, not so for new parallel software.

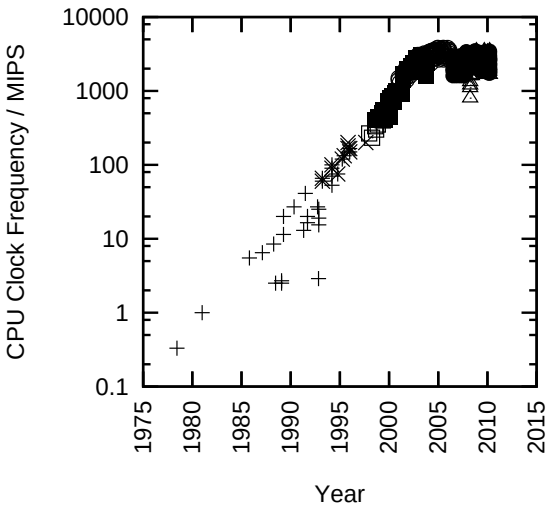

FIGURE 1: *MIPS/Clock-Frequency Trend for Intel CPUs*

This situation motivates some rules of thumb to help determine when parallel hardware and software is the right tool for the job. Section 2 discusses parallelizing the control loop itself, Section 3 covers some ways of dealing with difficult-to-parallelize workloads, Section 4 discusses alternatives to parallelizing control loops, Section 5 enumerates uses for leftover CPUs, and finally Section 6 presents concluding remarks and rules of thumb.

2 Parallelizing the Control Loop

This section looks at use of parallelism for the real-time control loop itself. Rather than specialize this discussion to any specific workload, we will consider the following randomly chosen synthetic workload:

```
1 void mung(int *x, int n)
2 {
3   int i;
4
5   for (i = 0; i < n; i++)
6     x[i] = 10 + x[i] / 10;
7 }
```

This function carries out a simple transform on each element of an array of integers. We can then examine the effectiveness of parallelism as a function of array size and number of passes over the array. The experiments in this paper will vary the array size and number of passes in tandem: ten passes over a ten-element array, one hundred passes over a one-hundred-element array, and so forth.

The two major modes of parallelism are pipelining and data parallelism. Pipelining partitions the workload in time, so that a given unit of work flows through the processors in the pipeline, as shown in Figure 2.

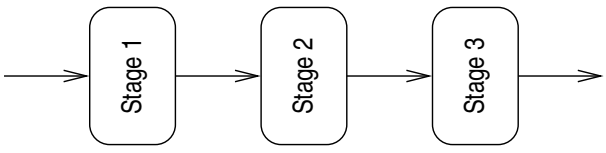

FIGURE 2: *Pipelined Parallelism*

In contrast, data parallelism partitions the workload in space, so that different sets of data are handled by different CPUs, as shown in Figure 3.

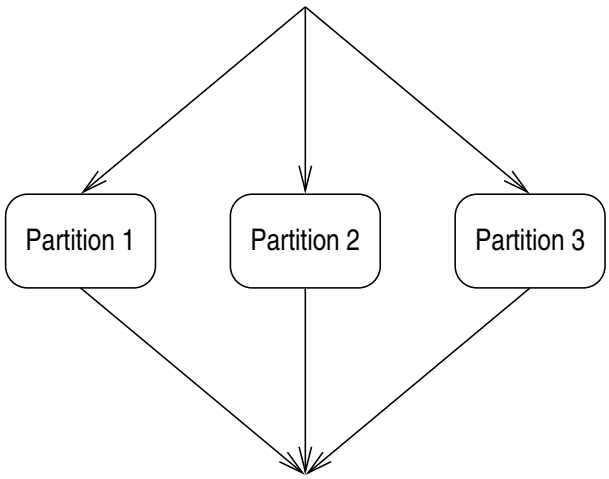

FIGURE 3: *Data Parallelism*

Suppose that a given workload had a large array that required two mathematical operations to be carried out on each element. In pipelining, one CPU would carry out the first operation on all elements, and another CPU would then carry out the second operation, again on all elements. In contrast, in data parallelism, one CPU would carry out both operations on each element in the first half of the array, and another CPU would carry out both operations on the second half of the array.

The next two sections look at each of these approaches in more detail.

2.1 Pipelining

We implement pipelining by processing half of the passes through all of the elements, then using `pthread_create()` [4] to handle the remainder of the passes. The results on an Intel® dual-core 2.53GHz dual-core laptop are shown in Figure 4. The dashed line is the response time of pipelining, and the solid line is the response time of single-threaded execution. From the results for small computations at the left, we can see that the overhead of `pthread_create()` and `pthread_join()` approaches 100 microseconds, which is far in excess of the single-threaded loop overhead of roughly 2 microseconds. Worse yet, the pipelined response time is never better than that of single-threaded execution, even for 10,000 passes through a 10,000-element array, which likely far exceeds the overhead of anything likely to appear in a self-respecting real-time control loop.

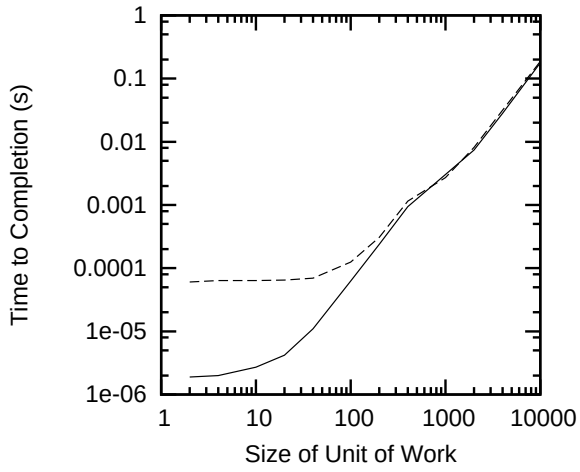

FIGURE 4: *Pipelining Performance*

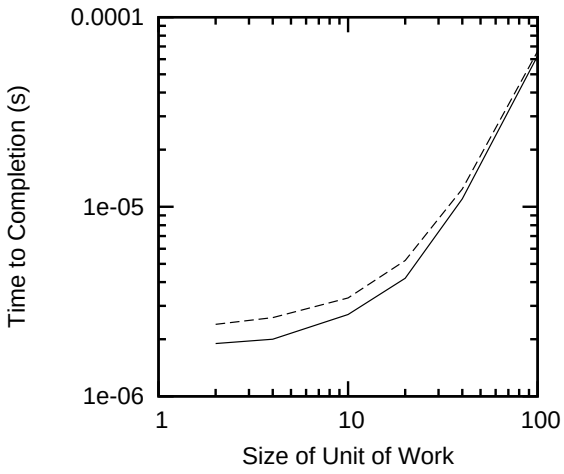

FIGURE 5: *Pipelining Performance, Pre-Existing Threads*

We clearly need more efficient synchronization mechanisms that `pthread_create()` and `pthread_join()`. A reasonable alternative is to create the needed threads at initialization, and then reuse these pre-existing threads for each new unit of work. This approach is more difficult to design than simple use of `pthread_create()` and `pthread_join()`, but, as can be seen in Figure 5, the synchronization overhead of this approach is roughly two orders of magnitude smaller, so that the response time of the pipelined approach nears that of the single-threaded approach with "only" 100 passes through a 100-element array.

But the goal of parallelism is not to merely approach the performance of single-threaded implementation, but rather to greatly exceed it. It is easy to see that pipelining is simply not up to this task, at least when work appears in isolation—the response time will always be greater than that of the single-threaded implementation, with the addition of synchronization overhead.

However, if units of work can arrive in quick succession, then pipelining can provide improvment in response time, as shown by Figure 6. The improvement occurs when a second unit of work arrives shortly after the first unit begins executing. Given single-threaded execution, the second unit will have to wait for the first unit to finish, as shown by the top half of the figure.

However, given pipelining, the second unit can begin execution as soon as the first unit finishes using CPU 0, so that the execution of the first half of the second unit overlaps with the second half of the first unit, as shown by the bottom half of the figure. The rhombuses denote synchronization overhead, showing further that reasonable improvement is possible even when the synchronization overhead is a substantial fraction of the duration of the units of work.

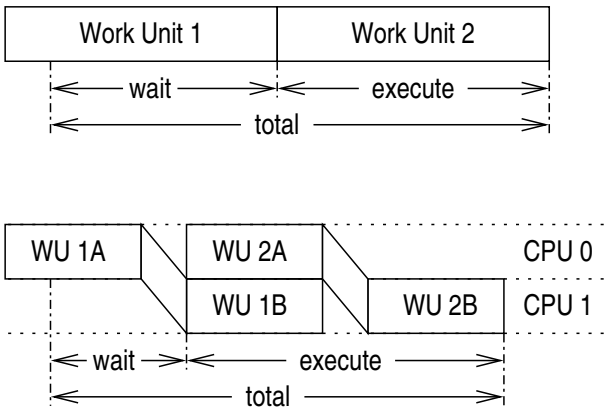

FIGURE 6: *Improving Response With Pipelining*

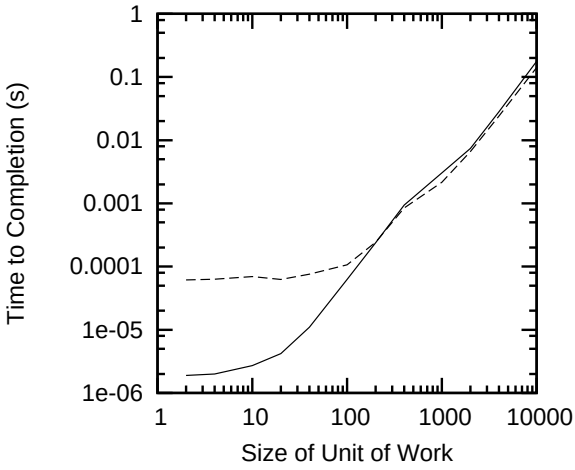

FIGURE 7: *Data-Parallel Performance*

In short, pipelining cannot reduce the response time of a single isolated unit of work, but can provide reductions in response time by overlapping the execution of successive units of work. Pipelining implementation is generally straightforward, but the overhead of each stage of the pipeline should be chosen so as to significantly exceed synchronization overhead, including both communication overhead and cache misses.

2.2 Data Parallelism

We implement data parallelism by processing half of the array elements by the parent task and the other half by a child task. As can be seen in Figure 7, this provides faster response time than does the single-threaded implementation, but only for work-unit sizes that are unrealistically large for most applications. Again, the solid line is the response time for single-threaded execution, and the dashed line is the response time for data-parallel execution, and again the base overhead of the data-parallel execution is dominated by the overhead of `pthread_create()` and `pthread_join()`.

And again, an alternative approach creates the needed threads at initialization time and passes work based on shared-memory operations, resulting in the much-improved response times shown in Figure 8. Significant improvements in response time appear with work-unit sizes as small as 100 (keep in mind that this plot uses logarithmic scales).

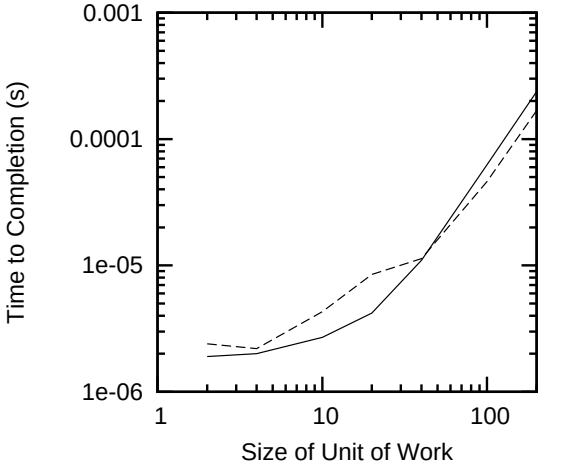

FIGURE 8: *Data-Parallel Performance, Pre-Existing Threads*

2.3 Control Loop Analysis

Of course, the best way to evaluate a given design is to implement it and measure the results. That said, a small amount of analysis can sometimes save considerable quantities of implementation and measurement.

Therefore, suppose that the overhead of a unit of work is T, that the synchronization overhead (including cache misses, atomic instructions, locking, etc.) is C, and that there are N CPUs. The duration of data-parallel execution is then given by:

$$\frac{T}{N} + C \tag{1}$$

The speedup is the ratio of the single-threaded execution time to that of the data-parallel implementation:

$$S = \frac{T}{\frac{T}{N} + C} \tag{2}$$

This can be normalized by multiplying the numerator and denominator by N/C:

$$S = \frac{N\frac{T}{C}}{\frac{T}{C} + N} \tag{3}$$

This in turn permits us to estimate an upper bound for the speedup given only number of CPUs and the ratio of the single-threaded execution time to the synchronization overhead, as shown in Figure 9 for N equal to 2, 3, and 4 CPUs. As can be seen from this figure, the ratio T/C must be reasonably large to attain reasonable speedups.

Although achieving linear speedup is a common parallel-programming goal, a more appropriate goal for real-time programming is meeting the specified deadlines. Figure 9 can be used for this purpose as well by taking the required speedup and scanning across the chart to find feasible values of T/C and N. Mathematically inclined readers might instead wish to solve Equation 3 for either T/C or N, and also to note the similarities to Amdahl's Law [1].

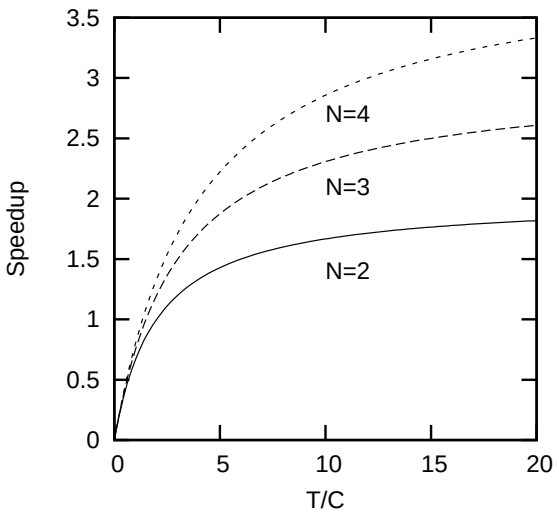

FIGURE 9: *Limits to Speedup*

2.4 Control Loop Discussion

Pipelining and data parallelism are often simple enough to fit into the tight deadlines that are often associated with real-time and embedded projects. These approaches to parallelism can significantly reduce response times, but only if the overhead of the control loop is at lead several tens of microseconds.

Although such overheads are not unheard of, this does beg the question of what is to be done in cases where the control-loop overhead is to small to efficiently parallelize, or where the workload is not well-suited to parallelization. Of course, if the control loop meets performance criteria given a single-threaded implementation, you should simply declare victory and retain the single-threaded implementation. The following sections discuss what you might do otherwise.

3 Dealing With Difficult-to-Parallelize Workloads

The prototypical difficult-to-parallelize workload would be encryption and decryption, where dependencies are often inserted into the workload for the express purpose of making it difficult to operate on the stream in parallel. If you have control of both

ends of the conversation and if there are large quantities of data being encrypted, one approach is interleaving. The idea is to split the unencrypted stream into blocks, so that the n^{th} CPU encrypts every n^{th} block. The receiver can then use multiple CPUs to decrypt and then splice the stream back together. The same approach may be used for compression, albeit most likely at some compression-ratio penalty.

Audio and video encoding and decoding can be more challenging, although one possible approach for video is to partition the display, and treat each partition as a separate video stream.

4 Alternatives to Parallelized Control Loops

Although parallel programming is not the impossible task that some believe it to be [8], there can be no doubt that it is more difficult than sequential programming. It therefore makes a lot of sense to consider alternatives to parallelizing the control loop, including the following:

1. Hand-optimizing the (single-threaded) control loop.

2. Use approximations in the control loop, with periodic calibration from a thread running on some other CPU.

3. Use hardware accelerators to speed up the computations. Such accelerators are especially attractive for audio and video encoding and decoding, for which a number of accelerators are available, though normally integrated into a given system-on-a-chip (SoC).

4. Use FPGAs to speed up the computations. This won't necessarily be easier than parallelism, but there are some workloads better suited to FPGAs than parallel software, and vice versa.

As always, use the right tool for the job!

And when the system is fast enough for your purposes, stop messing with it! If you can't stop yourself from messing with it, at least record the working fast-enough version in your favorite source-code control system.

5 Leftover CPUs

In sharp contrast with decades past, there is a very real possibility that your real-time system will be able to operate using only some of the available CPUs. The question then becomes "what to do with the leftover CPUs?" Here are some possibilities:

1. Power them off.

2. Switch to hardware with fewer CPUs. (The glass is neither half full or half empty, but rather twice as big as it needs to be!)

3. Use them to run any needed UI or reporting system.

4. In the case of enterprise real-time [6], run some of the enterprise portion of the application on the leftover CPUs.

These last two items require communicating from and to the real-time control loop. This is often done using special real-time messaging software, which can work well, especially between systems. However, given that the communication is occuring within a single system faster shared-memory techniques may be used. These techniques are often somewhat more difficult to use, but can offer much better performance and latencies. The optimal techique depends on the situation:

1. Messaging can use real-time implementations of linked lists [2] or user-mode equivalents of `kfifo` [3].

2. Lookups using linked structures such as hash tables, lists, or search trees on read-mostly data can use RCU [2, 5, 7].

3. Other communications mechanisms can use locking, provided that priority boosting is implemented [4].

However, thread placement is important, as can be seen in the following table.

Operation	Cost (ns)	Ratio
Clock Period	0.4	1.0
"Best-case" CAS	12.2	33.8
Best-case lock	25.6	71.2
Single cache miss	12.9	35.8
CAS cache miss	7.0	19.4
Single cache miss (off-core)	31.2	86.6
CAS cache miss (off-core)	31.2	86.5
Single cache miss (off-socket)	92.4	256.7
CAS cache miss (off-socket)	95.9	266.4

The first column of this table gives the type of operation and the locality, the second column gives the cost in nanoseconds, and the third column gives the ratio of the cost to the clock period, in other words, the number of cycles. This data was collected on a 16-CPU 2.8GHz Intel X5550 (Nehalem) system.

The first four rows following the clock period are operations within a given core, so that the "cache misses" merely move the data from one thread to another, while the "best case" rows have a single thread operating on the data. Interestingly enough, on this hardware, it is faster to move data from one thread to another within a core than it is to have a single thread operate atomically on that same data.

The next two rows are operations on different cores within the same socket. Moving data from one core to another is thus quite expensive compared to operating on that same data within a given core.

The final two rows are operations on different sockets, which is almost an order of magnitude more expensive than atomic operations within a given core and more than two orders of magnitude more expensive than simple non-atomic instructions.

It can therefore be beneficial to use facilities such as `sched_setaffinity()` to place processes so as to minimize communications latencies. This is especially important if communications operations appear in your real-time control loop, because any cache misses will add directly to your loop's response time. Read-mostly data is an important special case. Where RCU can be used, the processor caches will retain copies of the data where needed, so that placement becomes less critical.

6 Conclusions

As always, use the right tool for the job. In particular, if simple sequential execution permits you to meet your response-time goals, you probably should not bother with parallelism. However, if sequential execution is insufficient, both pipelining and data parallelism can improve response times, particularly when each pass through your control loop takes more than an order of magnitude longer than the execution time for the synchronization mechanism you have chosen.

Interleaving can be used in some cases to batch streamed workloads in order to make them easier to parallelize. Alternatively, sequential-code optimizations and hardware acceleration can be an attractive option for some types of workloads.

Difficult though it may be for old-timers such as the author to grasp, it is not unusual to find that your workload fails to use all of the CPUs available to you. These can be powered off, or they can run non-real-time portions of your extended workload, for example, the user interface. However, on larger systems, careful process and thread placement can optimize communications latencies and overheads.

Although parallelism is not always the right tool for the job, the low cost and easy availability of multicore systems and parallel software make parallelism much more attractive than in years past.

Acknowledgements

No article mentioning the -rt patchset would be complete without a note of thanks to Ingo Molnar, Thomas Gleixner, Sven Dietrich, K.R. Foley, Gene Heskett, Bill Huey, Esben Neilsen, Nick Piggin, Steven Rostedt, Michal Schmidt, Daniel Walker, Karsten Wiese, John Stultz, John Kacur, Carsten Emde, Clark Williams, GeunSik Lim, Uwe Kleine-Knig, Darren Hart, Dinakar Guniguntala, Luis Claudio R. Goncalves, and many others besides. I also owe thanks to Kathy Bennett for her support of this effort.

Legal Statement

References

[1] Gene Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, pages 483–485, Washington, DC, USA, 1967. IEEE Computer Society.

[2] Mathieu Desnoyers. [RFC git tree] userspace RCU (urcu) for Linux. Available: `http://lkml.org/lkml/2009/2/5/572` `http://lttng.org/urcu` [Viewed February 20, 2009], February 2009.

[3] Jake Edge. A new api for kfifo? Available: http://lwn.net/Articles/347619/ [Viewed September 20, 2010], August 2009.

[4] The Open Group. Single UNIX specification. http://www.opengroup.org/onlinepubs/007908799/index.html, July 2001.

[5] D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. *IBM Systems Journal*, 47(2):221–236, May 2008. Available: http://www.research.ibm.com/journal/sj/472/guniguntala.pdf [Viewed April 24, 2008].

[6] Paul E. McKenney. SMP and embedded real time. *Linux Journal*, (153):52–57, January 2007. Available: http://www.linuxjournal.com/article/9361 [Viewed May 31, 2007].

[7] Paul E. McKenney. Deterministic synchronization in multicore systems: the role of RCU. In *Eleventh Real Time Linux Workshop*, Dresden, Germany, September 2009. Available: http://www.rdrop.com/users/paulmck/realtime/paper/DetSyncRCU.2009.08.18a.pdf [Viewed January 14, 2009].

[8] Paul E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* kernel.org, Corvallis, OR, USA, 2010. Available: http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html [Viewed March 28, 2010].

[9] Ingo Molnar. Index of /pub/linux/kernel/projects/rt. Available: http://www.kernel.org/pub/linux/kernel/projects/rt/ [Viewed February 15, 2005], February 2005.