Supporting Higher-Order Controllers for Magnetic Bearings in a High-Speed, Real-Time Platform Using General-Purpose Computers*

Kevin Skadron[1], Marty Humphrey[1], Bin Huang[1], Edgar Hilton[2], Jihao Luo[3], and Paul Allaire[1]

[1]University of Virginia, Charlottesville, VA 22904
[2]FSM Labs, 3466 Hyde Park Way, Tallahassee, FL 32309
[3]AFS Trinity Power, Inc., Charlottesville, VA 22901

Abstract

One approach for implementing a control system for a magnetic bearing suspension system for high-speed rotating machinery is to use embedded DSP boards.Yet control systems based on DSP boards often require specialized programming and development tools, lack interoperability with standardized architectures and tools, and lack flexibility when computational requirements change. For reasons of cost and upgrade capabilities, it is instead desirable to implement these control systems with general-purpose, commodity PCs. Achieving adequate computational throughput is a major challenge however, even with the most advanced computer systems available today. This paper describes several improvements we make on a previous, uniprocessor version of our real-time controls platform in order to support more computationally-intensive, higher-order magnetic bearing controllers. First, the controls platform is made multiprocessor-capable and gain-scheduled controllers are computed asynchronously on the second processor in the dual-CPU system. Second, new floating-point computation instructions supported by the Pentium III and Pentium 4 are used to speed up the matrix calculation. Finally, other performance-tuning techniques are used in combination with increases in commodity processor computing speeds to optimize the controller. The results show a tremendous improvement in the overall throughput of this real time control platform, without sacrificing predictability. This improvement makes it feasible to implement these high-order magnetic bearing controllers which in turn push the performance of rotating machinery to a higher level.

1. Introduction

One important application of magnetic bearing suspension is in the area of high-speed rotating machinery. Unfortunately, magnetic bearing suspension systems are open loop unstable, making feedback controllers necessary to achieve stability. Improvements in control design are improving the system performance in the sense of disturbance rejection, damping, and robustness for use in high-speed rotating machinery. Yet these modern control techniques frequently result in high-order state space controllers, which, to make the problem worse, often require a high sampling rate to implement them. Furthermore, for reasons of cost and upgrade capabilities, it is desirable to implement these control systems with general-purpose, commodity PCs.

The first phase of this project successfully used RT-Linux and a commodity PC to implement a controller for a high-speed magnetic bearing [1][2][3]. A 700 MHz Intel Pentium III with a single A/D card with 5 channels and a single D/A card with 5 channels was used to implement a Mu-Synthesis controller and an anti-imbalance controller. A general graphical user interface (GUI) was implemented to allow controls engineers to both test new controls algorithms and view graphical output from running systems. Most importantly, RT-Linux provided the predictabilty necessary to implement a robust controller, at a significantly reduced hardware cost.

The second phase of the project is significantly more challenging. The plant will be controlled via a 44th order LPV (Linear Parameter Varying) controller running at 40KHz, which easily overwhelms the hardware capacity of the computer used in the first phase and stresses the limits of even the most advanced computer systems available today. Three approaches are used to meet the computational requirements of the LPV controller:

1. Parallelization of the LPV controller (and use of a multiprocessor version of RT-Linux). The core controller computations (output and state update)

* This work supported in part with a grant from AFS Trinity Power.

saturate the processor bandwidth. In addition, the controller matrices must be updated periodically to adjust to the rotor speed. Fortunately, this interpolation task, although computationally expensive, is not a real-time task and can be offloaded to a second processor. Dual-processor computer systems are now only incrementally more expensive than conventional single-processor systems. A key question, however, is whether the matrix can be communicated between processors with sufficient speed and without causing undue contention for the system bus.

2. Use of new processor instructions. The "Streaming SIMD" (SSE) instructions in the Pentium III and Pentium 4 processors support "mini-vector" operations in which the 128-bit floating-point hardware is used to perform 4 single-precision (32-bit) operations in parallel. Since single-precision provides sufficient accuracy for our controller, this permits up to a four-fold speedup in controller computation. This peak computation rate is moderated, however, by the overhead of setting up the packed data format in SSE registers and subsequently extracting results back into conventional registers.

3. Use of higher-speed commodity processors and appropriate I/O hardware. The Pentium 4 processor provides almost twice the clock rate and hence twice the peak execution rate of the Pentium III. However, speed alone is not sufficient without suitable hardware for A/D and D/A conversions.

This paper describes the implementation of these three approaches and their effectiveness. The next section describes the energy-storage flywheel that forms the basis for our test system. Section 3 characterizes the computational demands of the control application. Using this data, Section 4 then shows the benefits of spreading the tasks over a multi-processor and of using mini-vector instructions. Section 5 discusses I/O considerations and Section 6 concludes the paper.

2. Control Environment

2.1. Control Requirements

Control of complex modern systems usually involves the design and implementation of several independent tasks that must process data with varying degrees of logical and temporal importance. For an active magnetic bearing (AMB) system, we have identified the following tasks:

- *Five degree of freedom suspension controller*, runing at a fixed periodic rate, usually in the order of 25-200μs, providing the appropriate signals necessary to suspend the rotor. This task is critical—missing one period can have catastrophic consequences.
- *Spin rate measuring task*, triggered once per revolution, used to calculate the spin rate of the rotor. This task also divides the period of one rotation into 256 scheduling points and schedules the imbalance controller to be executed at each of these scheduling points. The accuracy of the spin rate calculated by this task makes this task very intolerant to temporal error.
- *Imbalance controller*, executed 256 times per revolution, used to produce a synchronous force used to counteract the effect of rotor imbalance. Slight temporal incorrectness is allowed.
- *Dta transfer and data plotting tasks*, sending data to disk, screen, or other devices. These tasks allow a relatively large temporal error.
- *Network transfer tasks*, which transfer data and commands to and from other computers. These tasks allow a larger temporal error and data loss. For safety reasons, especially, these networking tasks are critical for the safety of the operator so that the AMB can be operated and monitored from a remote location.
- *Miscellaneous tasks*, such as screen refresh, shell programs, computational engines (Matlab, Scilab, MuPAD, Mathematica, MathCAD, etc.). There is no temporal limitations on these tasks.

The success of the AMB is heavily dependent on the proper design of the suspension controller (task 1) and on the predictable execution of the controller. This is, of course, due to the inherent open loop instability of all AMBs. The purpose of the controller is to stabilize the closed loop system, meaning that the rotor will be suspended within the AMB with the necessary stiffness and damping. All the other tasks listed above act to provide a higher level support or control of the AMB, and are also very important in their own right. These too must also be executed in a predictable fashion, although, depending on the task, some tasks will allow for varying degrees of temporal predictability.

2.2. RTiC-Lab

The Open Source movement and Linux have lead to the birth of a hard real time operating system which is entirely based on Linux. This hard real time Linux, or RTLinux, is developed by FSMLabs, Inc.

and uses many of the strengths of Linux without interfering with the general Linux development.

RTLinux works by introducing a virtual machine in between the general purpose operating system (GPOS) of Linux and the underlying hardware, as shown in Figure 1. This virtual machine intercepts all interrupts being generated by the underlying hardware and passes these as soft interrupts to the GPOS only when real-time scheduling permits. Within RTLinux, a priority based scheduler identifies a group of hard real time tasks for scheduling. One of these real time tasks is a special task which is the combination of the full Linux GPOS and its underlying user tasks. Thus, the Linux GPOS cannot interact with any of the higher priority tasks unless the hard real time developer explicitly asks for this interaction. The end effect is that for a PC, the worst case interrupt latency and jitter are 15 and 30 µs, respectively. The resulting latencies are near those of the underlying hardware.

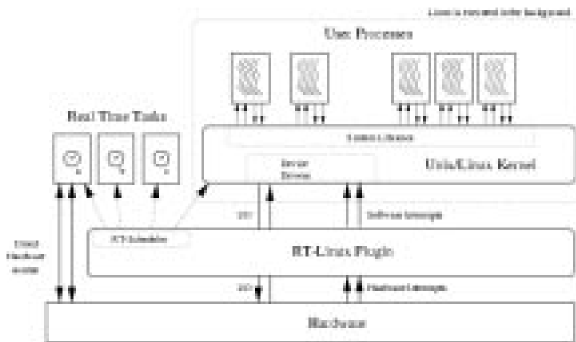

Figure 1: RTLinux Architecture

Control of AMBs requires an exhaustive tuning and characterization process during the early design stages. The Real Time Controls Laboratory, or RTiC-Lab, is software that uses the strengths of Linux and RTLinux, and is used not only during these early stages of controller design and plant characterization, but also during subsequent monitoring and control. Designed and tested at the University of Virginia's Rotating Machinery and Controls Laboratory, it provides an environment in which to implement controller algorithms while providing real time access to controller states, plant outputs, controller actions, controller parameters, and other controller information. All this information can be plotted and filtered—via user defined filters—in soft real time. The user can further filter the necessary data either in soft real time or post mortem. Most importantly, controller parameters can be updated in real time through a user-defined graphical user interface.

RTiC-Lab has two very important features not found in any other real time controls implementation platforms. First. RTiC-Lab is and will be—as with its underlying Linux and RTLinux platforms—Open Source Software, released and protected under the Free Software Foundation's General Public License. That is, users of RTiC-Lab can download the source code, use it, enhance it, and share it with their colleagues. Second, control using RTiC-Lab can be distributed over a common network of personal computers. That is, RTiC-Lab can be used over a common 10/100 Mb Ethernet network. Note, however, that if the controlled plant is both computationally simple and safe enough to be handled exclusively in a single computer, then RTiC-Lab can collapse into one single computer to control the entire plant.

An AMB example of RTiC-Lab is shown in Figure 2. A devoted display or host computer (DHC) is networked via 10 or 100 Mb/s TCP/IP network to a set of devoted controls computers (DCCs). The controls engineer sits at the DHC (which may or may not be at the same room or even building as the DCCs) and coordinates, codes, and synchronizes all DCCs from the DHC. Run-time parameters, such as sampling rate, startup delay, and networking parameters, can be set for each of the DCCs from the DHC. Each of the DCCs is a minimal computer system having no keyboard, harddrive, mouse, video card, or monitor. They only have the necessary I/O cards which are used to interface to the plant hardware and the necessary ethernet card to communicate with the DHC.

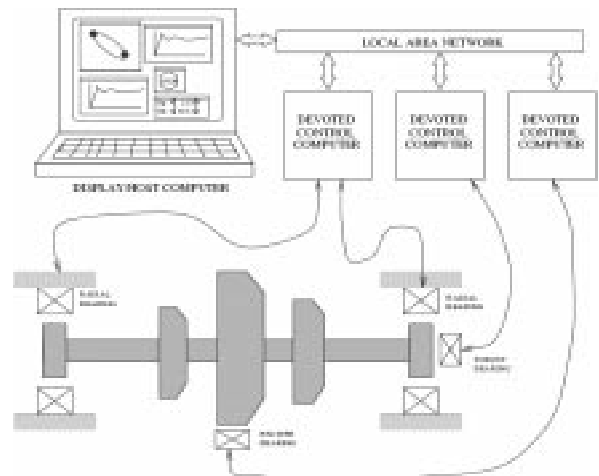

Figure 2: Example Configuration of RTiC-Lab

In accordance with the RT-Linux paradigm (Figure 1), RTiC-Lab separates the AMB controller into the hard real time or "embedded" part and the

soft real time or "reactive" part. The embedded part of the controller (resident exclusively in the DCCs) includes all tasks having hard timing constraints: 1) the AMB suspension controller(s) (both periodic and event driven), 2) a software watchdog, and 3) a set of interrupt service routines that are used for communication with the reactive task. The reactive task (resident in both DHC and DCCs) is a multi-threaded, user-space application which runs within the Linux kernel, performing the following functions: 1) communication with the embedded tasks via RT-FIFOs, 2) display of a graphical user interface for the user, 3) error checking of the user's controller code, 4) sending parameter updates to the embedded tasks as requested by user, and 5) sending data to either screen, file, or printer.

3. Characterizing the Computational Requirements of the LPV Controller

On a 733 MHz Intel Pentium-III with a 133 MHz bus, the measured time for one iteration of the 44th order LPV control computation was 100.21 µs, well short of the 25 µs needed to attain an update rate of 40 KHz. Various computing bottlenecks might be the source of the slow execution: sheer complexity of the algorithm, caches misses (which stall computation while the data is fetched from memory), TLB misses, branch mispredictions, etc.

To characterize the source of the delay, we needed to be able to accurately count events like cache misses for a known number of iterations. While the Pentium-III does provide built-in performance counters**,** they can be difficult to use for obtaining precise event counts. Instead, we found that the easiest way to obtain our measurements was to use the SimpleScalar simulator [4] that is widely used in microprocessor-design research. SimpleScalar includes a simulator that models the clock-cycle by clock-cycle flow of instructions through the processor pipelines. We configured the simulator to approximately imitate a Pentium-III and to only produce statistics for a single iteration of the control algorithm.

We found that for the 16 KB data caches of the Pentium-III, the control algorithm experiences a negligible number of cache misses, TLB misses, or branch mispredictions—see Table 1. Instead, the bottleneck is sheer computational throughput: one iteration of the controller constitutes approximately 128,000 assembly-language instructions. Even if the controller can operate at the Pentium-III's peak sustained bandwidth of three instructions-per-cycle (IPC), one iteration would take 42,667 clock cycles, which can only be accomplished in 25µs if the processor has a clock rate of 1.7 GHz. Based on our actual measurement of 100.21µs, the controller is only able to attain an average IPC rate of 1.4.

Event	Count
Instructions	128,423
First-level instruction-cache misses	340
First-level data-cache misses	1750
Second-level (unified) cache misses	2
TLB misses	0
Branch mispredictions	3

Table 1: Event counts in the simulated Pentium-III for a single iteration of the control algorithm.

The recently introduced Pentium-4 is now available at such clock rates, although sustained IPCs of 3 instructions are rare even in regular, loop-oriented code like the controller. Upgrading to a Pentium-4 with a higher clock speed clearly helps, but other methods are needed to reduce the computational complexity. The remainder of this paper shows how we use a second CPU and the Pentium family's multimedia or SSE instructions to reduce the instruction bandwidth. These techniques allow us to attain the desired 25 µs time per iteration, and the general techniques we describe are useful not only for our specific control application but for any real-time computing workload that requires high-speed computation.

4. Meeting the Computational Requirements of the LPV Controller

4.1. Multi-Processing Techniques

In theory, a commodity multiprocessor executing RTLinux provides the necessary computing capability by which to achieve the target execution frequency of higher-order controllers. However, it can be challenging to determine which parts of a sequential code can be executed in parallel such that the computation and communication occur prior to the hard real-time deadlines.

The nature of the LPV controller, from the perspective of its computational requirements, is that a linear interpolation of the various matrices in the controller parameters must occur periodically, based on the speed of the plant. That is, the speed of the plant rotation is used to determine the correct controller parameters, based on a linear combination of the parameters at the low end of the operating environment and on the high end of the operating environment. The measurement of 100.21µs

discussed in Section 3 is based on computing the interpolation *every* iteration of the control algorithm. However, because the system does not experience large changes of speed from one iteration to the next, this interpolation does not have to occur every iteration, and can be moved to a second processor and still achieve real-time correctness guarantees.

Posix real-time threads were used in RTLinux to move the interpolation computation to a second processor in the dual-processor PC. This second processor was previously unused by the real-time controls system. By extracting the interpolation from the main controller thread, we were able to reduce the duration of the main controller thread to 37.91µs. A double-buffering approach is used to ensure that that main controller thread always has a recent version of the interpolated matrices.

While this reduction is a significant step toward our goal of 25µs, there are still open issues with regard to the parallelization of the LPV controller. Even while repeatedly computing the interpolation, there is spare capacity on the second processor. Intuitively, as control algorithms become more complex and higher-order, it will be necessary to exploit some of this excess capacity. The fundamental challenge will be to precisely determine exactly how much capacity is available, and how to ensure the predictable communication between the processors given such tight deadlines. This is an area of future research for both us and the RTLinux community.

4.2. Mini-Vector Instructions

The Intel Pentium III processor family, introduced in February 1999, contains a new set of instructions: Streaming SIMD (Single Instruction, Multiple Data) Extensions (SSE). SSE allows a single microprocessor to perform multiple arithmetic operations in parallel, as if it were a miniature vector machine. SSE is similar to the previously introduced MMX instructions in that they share the concept of SIMD, but they differ in the data types they handle. MMX instructions provide SIMD for integers, while SSE instructions provide SIMD for single-precision floating-point numbers. MMX instructions operate on two 32-bit integers, while SSE instructions operate on four 32-bit floats simultaneously.

An example use of the “mulps” SSE operation is shown in Figure 4. Before the processor executes the “mulps”, the 4 32-bit floats must be loaded into the two registers, *xmm0* and *xmm1*. Conceptually, in a single time step, the 4 32-bit numbers in *xmm0* are multiplied in parallel with the corresponding 32-bit numbers in *xmm1,* with the resulting 4 32-bit numbers being placed back into *xmm0*. It is important to note that if the resulting numbers are to be used invidually (*i.e.*, not in subsequent SSE instructions), each 32-bit value must be “unpacked” from the special-purpose *xmm0* register, which requires a small amount of time.

Figure 3: Example of the "mulps xmm0, xmm1" instruction

However, there are limitations to the use of the SSE instructions. In theory, the use of the SSE instructions result in a 4x speedup. However, packing and unpacking the data into and out of the XMM registers significantly reduces the actual speedup. Also, at this time, compilers do not transparently insert the appropriate SSE instructions, but rather still rely on their non-SSE counterparts. Exploiting small-scale SIMD parallelism is an active area of development in the compiler community.

Therefore, to use the SSE instructions in the LPV code, we had to manually insert the assembler routines into our C source code. This use of assembly-language instructions currently restricts the upgrade path to the family of Pentium-III compatible processors, but permits us to demonstrate the value of SIMD instructions for high-speed, real-time control. As commercial compilers begin capitalizing on SIMD behavior, the use of assembly will no longer be necessary.

To use the SSE instructions required us to determine the best candidates for instruction-level parallelization, and to manage the contents of the SSE registers by hand. After careful instrumentation of the LPV controller code, we focused on a small number of the most appropriate statements from the C code, and replaced approximately 5 C statements with 300 hand-coded assembly routines and validated the resulting operation. The modified controller showed that the dual-processor version of the controller with SSE routines executed in 27.09 µs. We believe that this version of the multiprocessor, SSE-enabled version of the LPV controller will enable us to meet our timing requirement of 25 µs with only modest improvements of raw hardware capacity (certainly the

Pentium 4, available in clock speeds of 1.4-2.0 GHz, will more than suffice) and the choice of proper hardware for A/D and D/A conversions, the subject of the next section.

5. I/O Requirements

While improved computational throughput is necessary to meet our goals, it is not the computational limitations that cause the greatest limitation, but rather I/O. For example, using ISA A/D and D/A cards, a system executing a five degree of freedom control algorithm will spend 55 µs on I/O. If output conversion (D/A) of an already-computed result is overlapped with the next iteration of the controller computation (this pipelining of conversion and computation is supported by RTiC-Lab), the system still must spend 30 µs on I/O. For our 8kHz (125 µs), phase-one controller, this was accepetable, but for a controller running at 25 µs, even the time spend on mere A/D conversion is greater than our entire desired time per iteration.

The chief problem is that ISA cards take approximately 5 µs to perform each conversion. Faster ISA cards can be purchased that will reduce the conversion time to under 1 µs. Unfortunately, ISA have the further problem that they hold the system bus for extended periods of time and so even the faster ISA cards are not suitable for high-speed, real-time control.

PCI cards for A/D and D/A are now widely available and are much better suited for such applications. They provide throughput, in bytes, that is equivalent to the bus frequency times two divided by the number of cards in the bus. For a 100MHz bus, this corresponds to a maximum theoretical sustained transfer rate of 200 MB/s for one card, 100 MB/s for two cards, etc. Furthermore, PCI cards hold the bus for only fractions of a microsecond each time that they are accessed.

The final part of our solution, then is simply the choice of PCI cards for A/D and D/A that have the sufficiently fast conversion latency. If A/D is now reduced to 6 µs and RTiC-Lab pipelines D/A to be overlapped with the subsequent iteration, we are left with 19 µs per iteration that can be dedicated to computation—easily sufficient in conjunction with the use of the fast Pentium4 processors.

6. Conclusion

This paper shows that high-speed active magnetic bearings can be supported by commodity PCs, but that the use of multiple processors, the use of "mini-vector" SIMD instructions now supported by most microprocessors, and intelligent choice of I/O hardware are essential components for extracting the best performance from the real-time controls platform and maximizing the sophistication of magnetic bearing control that these systems can support.

References

[1] E. Hilton, M. Humphrey, J.A. Stankovic, and P. Allaire. "Design of an Open Source, Hard Real-Time Controls Implementation Platform for Active Magnetic Bearings." In *Proceedings of the Seventh International Symposium on Magnetic Suspension Technology,* Zurich, Switzerland, August 2000.

[2] E. Hilton, V. Yodaiken, M. Humphrey, and P. Allaire. "The Real Time Controls Laboratory and Open Source, Hard Real Time Controls Implementation Platform." In *Proceedings of the Second Real-Time Linux Workshop*, Orlando, Florida, November 2000.

[3] M. Humphrey, E. Hilton, and P. Allaire."Experiences using RT-Linux to Implement a Controlller for a High Speed Magnetic Bearing System." in *Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium"*, Vancouver, Canada, June 1999.

[4] D.C. Burger and T.M. Austin. "The SimpleScalar Tool Set, Version 2.0." *Computer Architecture News*, 25(3):13-25, June 1997.