The Use of Mini-Vector Instructions for Implementing High-Speed Feedback Controllers on General-Purpose Computers

Kevin Skadron[1], Marty Humphrey[1], Bin Huang[2], Edgar Hilton[3], Jihao Luo[4], and Paul Allaire[5]
[1]Dept. of Computer Science, University of Virginia, Charlottesville, VA 22904
[2]Dept. of Electical and Computer Engineering, University of Virginia, Charlottesville, VA 22904
[3]FSM Labs, 3466 Hyde Park Way, Tallahassee, FL 32309
[4]AFS Trinity Power, Inc., Charlottesville, VA 22901
[5]Dept. of Mechanical and Aerospace Engineering, University of Virginia, Charlottesville, VA 22904

ABSTRACT
This paper describes the use of Intel Pentium-III SSE instructions for high-order control computations in a high-spin-rate flywheel. This application is representative of many control environments that require both high performance and real-time guarantees. We find that the SSE instructions are able to provide a dramatic increase in performance but that they are difficult to use unless the programmer is provided with a simple API. Unfortunately, the resulting abstraction means that some of the potential gains are lost to function-call overhead, and use of the API still requires substantial interaction between the controls engineer and the experienced SSE programmer. The use of SSE entails a tension between performance and ease of use that is not easily resolved.

General Terms
Measurement, Performance, Design, Experimentation.

Keywords
Active magnetic bearing, flywheel, vector instructions, API, multi-threaded execution.

1. INTRODUCTION
Recent advances in the cost and speed of general-purpose microprocessors suggest that it is time to reconsider whether digital signal processors (DSPs) can be replaced by general-purpose (multi-)processors (GPPs) for many streaming-media and otherwise high-capacity numerical computations. GPPs offer many advantages, including native support for floating-point execution, greater flexibility, easier programmability, greater portability from generation to generation, a huge menu of available commodity software products, and general ease of use for non computer scientists. Unfortunately, there are also many unresolved issues, centering on whether GPPs have the raw computational capacity, can meet the stringent timing constraints, and can meet the predictability requirements of applications for which DSPs were previously considered to be the only solution

This paper is not a comparison of DSP and GPP approaches, but rather describes experiences from designing and implementing a testbed based on GPPs for developing high-speed, computational feedback controllers and the value of so-called mini-vector instructions for this purpose. Our application for feedback control is real-time control of active magnetic bearings (AMBs) in a high-speed, energy-storage flywheel, and the purpose of the testbed is to provide a platform in which new and hopefully more accurate controllers can be evaluated. This testbed is designed for use by controls engineers with limited depth of knowledge in computer science and computer architecture and hence the emphasis is not only on performance and predictability, but also on ease of use. The system is prototypical of the control and system-design requirements in many types of rotating machinery.

The first phase of this project focused on developing an RT-Linux [1] based, real-time controls-testing environment called *RTiC-Lab* (see Section 2.2) for this testbed, and successfully satisfied real-time, predictability, and capacity concerns [2,3,4]. Since the emphasis was on the development of RTiC-Lab, only controllers with modest computational demands (small controllers with 4 KHz update rates) were employed.

This paper reports on the second phase of the project, which seeks to improve the overall performance of the testbed to study control for AMBs in high-speed environments like the flywheel, where greater speeds provide greater energy-storage capabilities. Both the computational requirements of the controller (*i.e.,* what is done during each iteration) and the frequency of the control iterations (40,000 updates/sec., *i.e.,* 40 KHz) are significantly increased, easily overwhelming the hardware capacity of the computer used in the first phase and surpassing the limits of even the most advanced commodity PC available today. Measurements on a 700 MHz Intel Pentium-III (Coppermine) with a 133 MHz bus using gcc 3.1 -O3 showed that on average, each iteration of the controller took 73.30 μs, much longer than the 25 μs needed to attain an update rate of 40 KHz. One solution to this speed problem is to simply ride the speed curve, as processors double in speed every 1.5 years. Yet controls engineers desire improved throughput today, and techniques that can provide speedups now, while still riding the speed curve, should certainly be explored. In addition, our goal of 40 KHz is only an intermediate goal, as the controls engineers will continue

In *Proceedings of the 3rd Workshop on Media and Stream Processors*, Dec. 2001, held in conjunction with the 34th Annual ACM/IEEE International Symposium on Microarchitecture, Austin, TX.

to design more sophisticated and demanding algorithms that will require continuing improvements in computational speed.

Because the controllers are based primarily on matrix operations and require only single-precision floating-point capabilities, the mini-vector instructions now prevalent in many instruction-set architectures are a natural vehicle for obtaining improved throughput. These instructions partition existing high-precision hardware to perform multiple lower-precision operations in parallel—similar to vector operations provided by some supercomputer. For example, with minimal extra overhead, quad-precision floating-point hardware can be used to perform four single-precision operations.

In particular, we use the Streaming-SIMD (SSE) instructions [5] for floating-point operations provided by the Intel Pentium III and Pentium 4 processors to achieve nearly 2x speedups on the key section of the controller. As we describe in this paper, SSE technology is an important building block upon which we were able to meet the computational challenges, but the benefits of SSE were reduced by a number of issues with its suitability for use in the testbed as well as with its suitability for use by non-expert programmers.

An example use of the "mulps" SSE operation is shown in Figure 1. Before the processor executes the "mulps", the 4 32-bit floats must be loaded into the two registers, *xmm0* and *xmm1*. Then the four 32-bit numbers in *xmm0* are multiplied in parallel with the corresponding 32-bit numbers in *xmm1*, with the resulting four 32-bit numbers being placed back into *xmm0*. It is important to note that if the resulting numbers are to be used individually (*i.e.*, not in subsequent SSE instruction), each 32-bit value must be "unpacked" from the special-purpose *xmm0* register, which requires one or more explicit instructions depending on the values' next use.

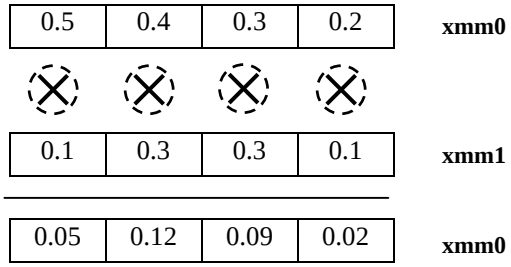

Figure 1: Example of the "mulps xmm0, xmm1" instruction

The next section describes the energy-storage flywheel and controls testbed and its computational requirements. Section 3 describes the use of SSE to ultimately solve the computational challenges, and Section 4 discusses some of the issues that arise when trying to use SSE instructions in an environment like the testbed described here.

2. CONTROL ENVIRONMENT

Before the use of SSE on the controller can be described, it is important to briefly describe the computational environment in which the controller executes. Section 2.1 briefly describes the energy-storage flywheel and the testbed system. Section 2.2 introduces the RT-Linux-based software environment we have constructed around it, called RTiC-Lab. Section 2.3 describes the characterization of the controller application in order to more concretely describe the challenges, and Section 2.4 briefly describes how we parallelized the controller in order to reduce, but not solve, the computational challenges.

2.1. Testbed Overview

Energy-storage flywheels offer compelling advantages as backup energy sources and as replacements for batteries. They are particularly attractive for environments where low weight and long operating life are important, like satellites. Indeed, in satellites, flywheels have the additional advantage that they can also replace some of the gyroscopes used for navigation and orientation.

To avoid the parasitic effects on energy and the maintenance problems that mechanical ball bearings incur, the energy-storage flywheel under study at the University of Virginia Rotating Machinery and Controls (ROMAC) Laboratory uses active magnetic bearings (AMBs) [6], which are simply electromagnets situated around the flywheel housing. The bearing currents produce a magnetic field that controls the position and flexion of the flywheel shaft.

Control of the AMB currents is achieved with a five-degree-of-freedom state-space controller running at a fixed periodic rate that corresponds to multiple current updates per revolution of the flywheel [7], providing the appropriate signals necessary to suspend the rotor and counteract the effects of rotor imbalance and any disturbances. The use of a state-space controller entails the manipulation of large matrices that embody the current state of the system and corresponding control elements. The state-space controller also requires a very simple spin-rate-measuring task to calculate the current rotational speed of the rotor. This rate is used to update the controller matrices (see Section 2.4). Both tasks (spin rate and controller computations) are presumed to be critical—missing one period can allow sufficient deviation in the flywheel's position for collision with the housing—hence the use of RT-Linux. Even in the presence of mechanical backup bearings, at high rotational speeds the flywheel stores so much energy that such collisions can be catastrophic, at best leading to large energy losses and at worst leading to explosive destruction of the system (*e.g.*, the satellite).

Accurate control of the flywheel is important not only to avoid catastrophic failure, but also to minimize the need for adjustments in its position and consequent loss of stored energy. The development of feedback controllers—especially state-space controllers for open-loop unstable systems like the bearing currents—requires an exhaustive tuning and characterization process during the early design stages. This imposes an additional set of tasks into the control system, namely data transfer so that the control engineer can observe and tune the performance of the current control design. More detail about the

overall control system can be found in [8], but the exact nature of the control algorithms remains confidential.

To deal with these difficulties, ROMAC is developing a high-performance testbed for controller design. The goal is to provide an environment in which a controls engineer with little programming experience can safely test new controllers at full speed and evaluate their accuracy. An overview of the flywheel in this test system appears in Figure 2. This figure shows the flywheel and bearings in a typical configuration. The critical task is control of the radial bearings, especially since the rotors under study are not rigid at high speeds. Axial motion, on the other hand, is minimal and requires minimal computational effort. Note that the computational infrastructure and techniques developed for this testbed are easily ported to other systems of rotating machinery, especially those using AMBs.

2.2. RTiC-Lab

The open source movement and Linux have led to the birth of a hard real time operating system which is entirely based on Linux but which does not interfere with its development. RT-Linux [1] works by introducing a virtual machine between Linux and the underlying hardware, intercepting all interrupts generated by the underlying hardware and passing these as soft interrupts to Linux only when real-time scheduling permits. This is shown in Figure 3. Within RT-Linux, 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 Real Time Controls Laboratory, or RTiC-Lab [2,3,4], is software that builds on RT-Linux 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 ROMAC, 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 in soft real time. The user can further filter data in a post-mortem fashion. Most importantly, controller parameters can be updated in real time through a user-defined graphical user interface. The desire to provide this “plug-and-play”-type capability in which controls engineers (not computer scientists) insert their new controllers has a direct consequence to the use of SSE which we discuss in Section 4. The organization of a testbed using RTiC-Lab is shown in Figure 2, with RTiC-Lab running on the host computer. When using RTiC-Lab, depending on computational-throughput and safety requirements, the controllers may also run on the host computer or on devoted control computers (DCCs). Given that radial-bearing control saturates current microprocessors and that flywheel failure presents severe physical danger, the flywheel is isolated in a separate chamber and controlled by one or more DCCs connected to the host computer by Ethernet and transmitting relevant data to the display/host computer.

RTiC-Lab has two 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 RT-Linux 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 Mbit 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..

Figure 2 shows an RTiC-Lab testbed configured for use with the energy storage flywheel. 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 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, mass storage, mouse, video card, or monitor. They only have the necessary I/O cards to interface to the plant hardware and the necessary Ethernet card to communicate with the DHC.

In accordance with the RT-Linux paradigm (Figure 3), 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) displaying data to screen, file, or printer.

2.3. Characterizing the Computational Requirements of the Controller

To determine the most appropriate approach for speeding up the computation, it was first necessary to determine the source: sheer complexity of the algorithm, caches misses, TLB misses, branch mispredictions, etc. To do so, we needed to be able to accurately count events like cache misses for a known number of iterations. We chose to use the SimpleScalar simulator [9], which can model 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. An alternative approach would have been the use of the Pentium-III’s performance counters, but SimpleScalar gives us the ability to predict performance on a variety of processor configurations.

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 (Table 1).

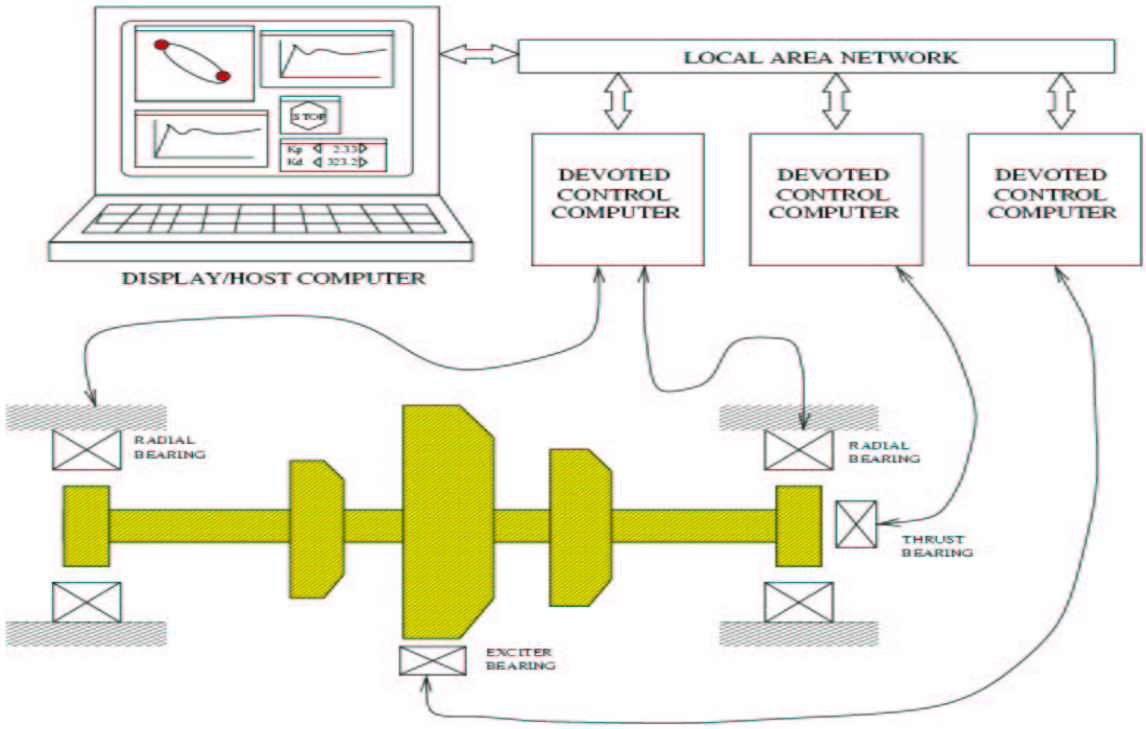

Figure 2. Flywheel-control testbed, including flywheel, AMBs, host computer, and optional devoted control computers.

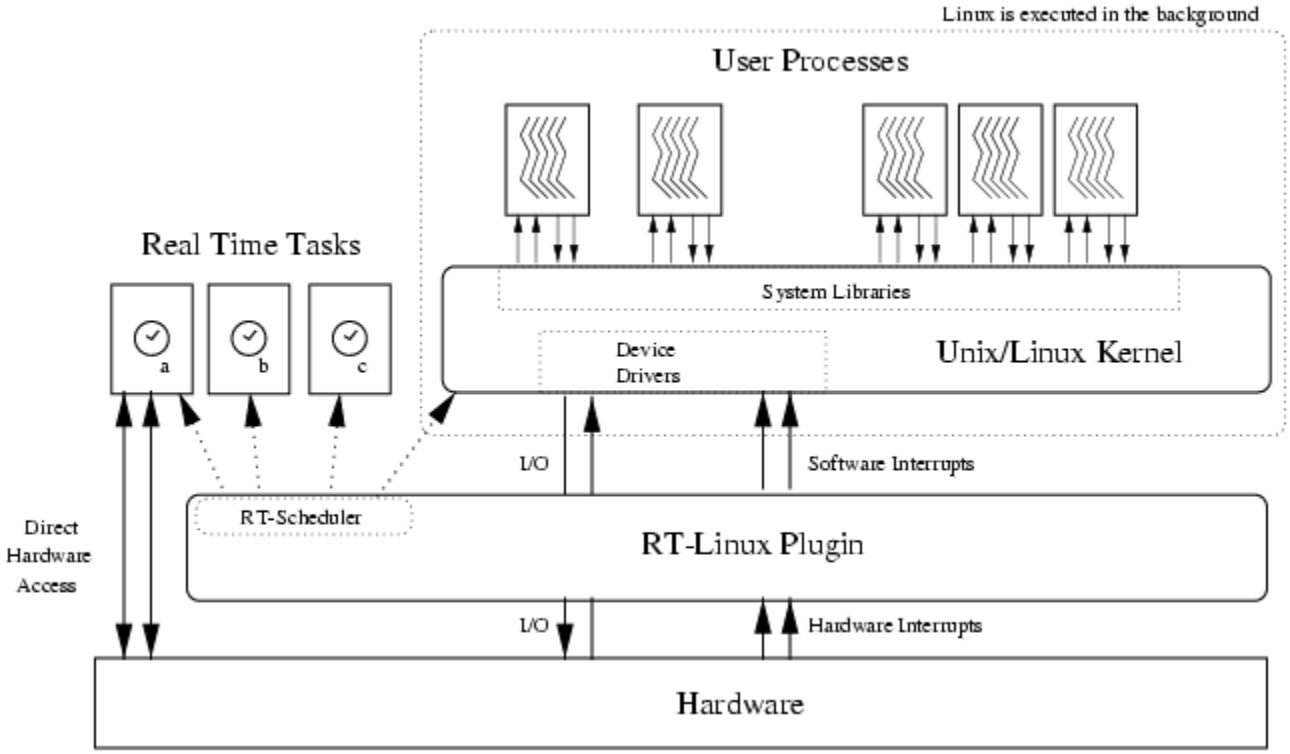

Figure 3. RT-Linux architecture.

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. Note that although 1.7GHz processors are available, the overhead of the operating system is not factored in, implying that increased commodity speed alone will not solve our computational capacity problems.

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

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

An additional performance issue that is not characterized above is the impact of I/O. Newly available PCI boards permit the necessary 5 channels of A/D or D/A to be performed within 6 μs; A/D (input) is synchronous and cannot be overlapped with the controller computation, but D/A (output) can be overlapped with the subsequent iteration. Out of the 25 μs budget per iteration to achieve 40 KHz update rates, A/D therefore leaves us with 19 μs per iteration that can be dedicated to computation.

2.4. Parallelizing the Controller Computation

Upon analyzing the controller algorithm, a computationally expensive, but non-real-time, section was identified that could be executed in parallel with the main controller iteration. This code dynamically adjusts the control parameters to the flywheel's current rotational speed by performing a linear interpolation between two 48x48 matrices. This interpolation uses the speed of the flywheel's rotation 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. In the original version of the control code, this interpolation is performed every iteration. The 73.30 μs time reported in Section 2.3 is based on this version. However, because this interpolation does not have to occur every iteration, it can be moved to a second processor and still achieve the real-time correctness guarantees. This second processor was previously unused by the real-time controls system.

Upon restructuring the algorithm to use POSIX threads and offload the interpolation task to a second processor, the duration of one iteration in the main thread was reduced to an average of 34.17 μs, still shy of the 19 μs target. Measurements were made using the RDTSC Pentium instruction, hence no simulation is involved here. A double-buffering approach is used to ensure that that main controller thread always has a recent version of the interpolated matrices. Newly interpolated versions of the matrices are not time-critical and hence are transmitted back to the first processor gradually.

The rest of the paper focuses on using SSE instructions to improve the computational throughput of the remaining, critical computational tasks.

3. APPLYING SSE INSTRUCTIONS TO THE MAIN CONTROLLER THREAD

Two approaches were used to apply SSE instructions to the main controller thread. In the first approach, we used two SSE-enabled compilers, gcc 3.1 [10] and the Intel C++ Compiler 5.0.1 for Linux [11,12]. In the second approach, we manually selected and inserted SSE instructions in the code. Again, measurements were made using the RDTSC Pentium instruction, hence no simulation is involved here. For the experiments in this section, interpolation was removed altogether to isolate the effects of SSE. This change in the code accounts for the difference in "no-interpolation" iteration times between Sections 2.5 and 3.

3.1. Using SSE-Enabled Compilers

Gcc 3.1 supports both the SSE and SSE-2 instruction set for Pentium-III and Pentium-4 processors, as does the Intel C++ Compiler for Linux. The result of applying these SSE-enabled compilers to this specific controller code is shown in Table 3.

Table 2. Performance of SSE-enabled compilers.

Compiler	Time per Iteration
gcc 3.1 (no SSE)	28.75 μs
gcc 3.1	28.05 μs
Intel C++ Compiler (no SSE)	32.11 μs
Intel C++ Compiler	32.00 μs

The Intel C++ Compiler is very useful at describing its attempts to vectorize the compiled code (via the *–vec_report3* directive). There were 23 loops that conceivably could have been vectorized. Of these, ten were not vectorized because they were not inner loops, two were not vectorized because they contained unsupported loop structures, nine were not vectorized because they contained an unvectorizable statement (generally another loop statement), one was not vectorized because of having mixed data types, and one *was* vectorized (a loop initializing all entries of a 5-element array to 0). While only 1 out of 23 is arguably of little value, our conclusion is not that the Intel Compiler is particularly bad in vectorizing our code (we believe the gcc compiler is similar, although its diagnostics do not allow us to easily confirm this); rather, the use of SSE in compiler technology is very complex and will be limited to hand-coding—possibly with the use of libraries—for the foreseeable future.

3.2. Manually Inserting SSE Instructions

Given that there was not a significant performance gained by compiler-enabled use of the SSE instructions, we were forced to manually insert SSE into the controller code. The overall result of our application of SSE is shown in Table 3.

Table 3. Performance speedup when manually inserting SSE instructions

Compiler	Time per Iteration
gcc 3.1 (no SSE)	28.75 μs
gcc 3.1 (manual use of SSE)	17.96 μs

Before attempting to use SSE, we decided that it was not fruitful to blindly apply SSE instructions to the *entire* controller iteration. To determine the best sections of code on which to apply SSE, we instrumented the code using the RDTSC instruction and identified a particularly time-consuming section that perform a matrix multiply of 48x48 by 48x1. We developed a straightforward matrix-multiply routine in assembly code based on the SSE operations *movaps*, *mulps*, *addps*, and *xorps*. By using the SSE instructions, we were able to reduce the average duration of this section from 19.60 μs to 9.52 μs, approximately a 2x speedup. In theory, the use of SSE instructions should result in a 4x speedup, but packing and unpacking the data into and out of the XMM registers significantly reduces the actual speedup. Nevertheless, while our ultimate goal is to reduce the duration as much as possible, an overall iteration time of 17.96 μs is almost fast enough for us to conclude that we have met our target frequency with a sufficient margin of safety. Faster processors will certainly allow us to meet that frequency. Further, smaller speedups can be achieved by vectorizing other important but smaller sections of code. Note that we chose not to use existing libraries like Intel's Performance Libraries [13]. Although the results reported here use a straightforward implementation that may have inferior performance to the tuned libraries from Intel, hand-coding gives us the flexibility to explore the impact of various alternatives that are customized for our control application.

4. DISCUSSION

While SSE directly facilitated satisfying our throughout requirements for *this* particular controller and *this* particular target frequency, there are open issues and concerns, especially in regard to the application of such techniques by users of RTiC-Lab.

The goal of RTiC-Lab is to provide a computational environment in which people who are *not* experts in computer programming (and certainly not experts in the use of SSE) can insert their domain-specific controllers into a hard real-time environment. Clearly, fully-featured SSE-enabled compilers will not be available for some time. Explaining the concept behind SSE instructions is easy enough, but learning to program with them is often difficult for non-computer-scientists, especially if they have no prior experience with assembly-language programming. This makes effectively programming a control algorithm using SSE almost prohibitively difficult for most controls engineers. Reusability of SSE assembly code is equally difficult for the same reasons.

Even if the controls engineer does not perform the SSE programming, another necessity for explaining the SSE code to the controls engineer is to describe the programmed functionality in the event that either this functionality is not correct or the timing issues are not satisfied. That is, if SSE is used, there appears to be a fragile, repeated discourse necessary between the controls engineer and the computer scientist until the timing issues and functionality are satisfied. For the most part, we have found this process difficult, time-consuming, and error-prone.

Instead of dealing directly with assembly code, perhaps a better approach is to provide library routines and an API that the controls engineer *must* invoke in order to achieve the benefits of SSE (*e.g.*, routines like *create_matrix()* and *multiply_matices()*). Using libraries hides the complexity of SSE implementations, and requiring their use has the advantage that SSE-based implementations can transparently use alternative representations to minimize packing and unpacking overhead.

For simple matrix-matrix and vector-matrix operations, this has worked well. The routines are flexible enough to be useful in the design of new controllers and the interface is reasonably straightforward for controls engineers to use in their own programming. Unfortunately, the use of an API does not eliminate the need for discourse between the controls engineer and the computer scientist, since the best way to implement a control algorithm is not always obvious, since the API cannot be used blindly for just any vector or matrix operation, and since the performance results are not always obvious to interpret.

We used a simple library with that we developed with just a limited set of hand-coded vector/matrix operations. We next plan to port our code to instead use the Intel Performance Libraries [13], which are available for use with Linux. The Intel libraries have a greater range of functionality and are likely to have more finely tuned SSE implementations. They also provide a cleaner upgrade path to new versions of the Intel architecture.

We remain concerned that any chosen API will not provide the exact mix of functionality needed as control algorithms become more complex. We also remain concerned that the generality of the library functions incurs unnecessary overhead. Given that the nature of controls applications is to continually increase controller complexity, which usually directly correlates with computational and programming requirements, the effort to effectively use SSE remains considerable.

5. SUMMARY

We have described a testbed for the design of controllers for rotating machinery and its use to develop state-space controllers for an energy-storage flywheel. Applications like this require high-performance floating-point throughput, and we have found that mini-vector instructions, like the SSE instructions provided by the Intel Pentium III and Pentium 4, can provide at least a factor of two speedup for the vector/matrix operations in our

control code. Unfortunately, we have found SSE difficult to use, especially given the multi-disciplinary environment in which the SSE code is developed and the need for non-expert programmers to work with SSE. Use of a library of common functions has helped, but some knowledge of the underlying SSE behavior remains necessary.

The current status of the project is that we believe SSE is an important technology for computationally-intensive, real-time applications, but that the use of SSE entails a tension between performance and ease of use that is not easily resolved.

ACKNOWLEDGMENTS

This work was supported in part by a grant from AFS Trinity Power Corp. We would also like to thank the anonymous reviewers for their helpful comments.

REFERENCES

[1] M. Barabanov and V. Yodaiken. "Introducing Real-Time Linux." *Linux Journal*, Feb. 1997.

[2] 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.

[3] 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.

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

[5] S. Thakkar and T. Huff. "Internet Streaming SIMD Extensions," *IEEE Computer*, 32:26-34, 1999.

[6] P. E. Allaire *et al.* "Magnetic Bearings." In *CRC Handbook of Lubrication and Tribology*, vol. 3, E. R. Boozer, editor, pages 577-600, 1994.

[7] J. Collado, R. Lozano, and A. Ailon. "Semi-global stabilization of discrete-time systems with bounded inputs using a periodic controller." *Systems & Control Letters*, vol. 36, pages 267-75, 1999.

[8] K. Skadron *et al.* "Supporting Higher-Order Controllers for Magnetic Bearings in a High-Speed, Real-Time Platform Using General-Purpose Computers." In *Proceedings of the 2001 International Symposium on Magnetic Suspension Technology*, Oct. 2001.

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

[10] Gnu C Compiler. http://gcc.gnu.org/

[11] Intel C Compiler for Linux. http://developer.intel.com/software/products/compilers/c50/

[12] A. Bik, M. Girkar, P. Grey, and X. Tian. "Efficient Exploitation of Parallelism on Pentium III and Pentium 4 Processor-Based Systems", *Intel Technology Journal*, Q1, 2001.

[13] Intel Performance Libraries. http://www.intel.com/software/products/perflib/