Thread-Based Virtual Duplex Systems in Embedded Environments

Virtual duplex systems provide the cost benefit of requiring only one rather than two processors. To lighten the single processor's burden and meet real-time requirements in embedded systems, the authors propose using emulated multithreading to reduce overhead and allow faster fault detection.

Jörg Keller
Andreas Grävinghoff
FernUniversität Hagen

Embedded systems play an important role in our daily lives: We interact, sometimes unknowingly, with embedded systems in applications ranging from cars, trains, and aircraft to washing machines and other household appliances. In an increasing number of applications, the system's ability to detect or tolerate transient and permanent faults is important. To achieve this goal, a system requires redundancy, or multiple resources. In the past, designers have used some form of duplex system, or structural redundancy, for these applications.

However, the cost associated with duplicated hardware resources is a major drawback of duplex systems. For embedded systems, which are usually deployed in very high volumes in highly competitive products such as household appliances, there is a strong motivation to lower costs. In this article, we present a new approach, which avoids the high costs associated with traditional duplex systems without sacrificing the ability to detect and tolerate faults.

The basis of our approach is virtual duplex systems.[1] Virtual duplex systems use temporal rather than structural redundancy to detect faults. Experimental results show that such systems provide excellent fault detection in the case of transient failures. For permanent failures, the use of systematic diversity in combination with design diversity produces encouraging results.[2] A virtual duplex system requires only a single processing node, whereas a traditional duplex system needs two. To match the performance of a traditional duplex system utilizing more than 50 percent of each processor's performance, a virtual duplex system must use a faster processor. However, one faster processor is usually cheaper than two slower ones. In addition, a virtual duplex system requires only one instance of supporting circuitry (memory, I/O, and so forth) and uses a smaller printed circuit board.

Because virtual duplex systems use temporal redundancy, their use might interfere with embedded applications' real-time requirements. For example, consider an autonomous guided vehicle that uses a camera to control its movements: The embedded system controlling the vehicle must issue instructions at a rate proportional to the vehicle speed; that

Published by the IEEE Computer Society

0272-1732/05/$20.00 © 2005 IEEE

is, there is little time for fault detection because instructions are issued only after the absence of faults has been asserted.

Virtual duplex systems use several processes, so the duration of context switches between processes is a constraint on fast fault detection: The percentage of context-switching time increases relative to user time as the time available for fault detection decreases. Context-switching overhead is a well-known operating-system problem. Modern operating systems, therefore, support threads, which share resources (such as address space and open files) to decrease context-switching time. Even with threads, however, a context switch is an expensive operation, and no embedded processors support hardware multithreading. Therefore, we propose virtual duplex systems using emulated multithreading. Originally developed to hide memory latency in massively parallel computers,[3,4] emulated multithreading provides very fast context switches and thus enables very small grain sizes.

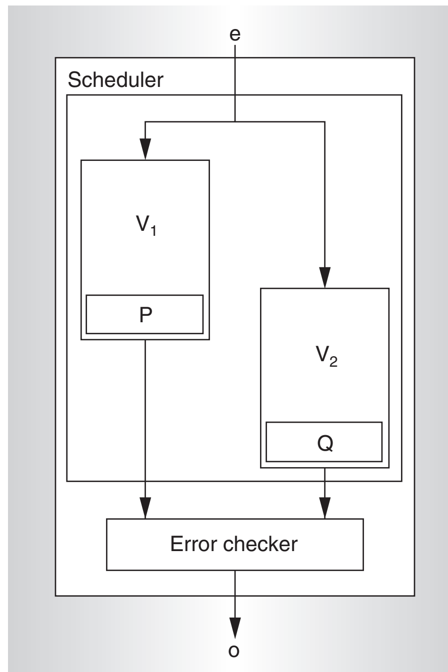

Figure 1. Basic structure of virtual duplex system.

Virtual duplex systems

Structural redundancy in the form of duplex systems is a common method of supporting fault detection. During the past two decades, the use of temporal redundancy emerged, especially in circuit design areas such as alternating circuits, alternate data retry, and recomputing with shifted operands. Designers also have used temporal redundancy in connection with design diversity to detect design faults through self-reducible functions, certification trails, and program checking. Echtle, Hinz, and Nikolov discuss using temporal redundancy in connection with design diversity to detect permanent as well as transient hardware failures.[1] We describe the virtual duplex systems introduced in their work in the next paragraphs. In their experiments, such systems detected design faults and transient hardware failures with very good fault coverage. These systems also provided limited coverage of permanent hardware failures. On the basis of this work, Lovric used systematic diversity and diverse error-correcting codes to enhance the fault detection capability of virtual duplex systems.[2,5,6] This combination of techniques yielded good results even in the presence of permanent hardware failures.

Figure 1 shows a virtual duplex system's basic structure. The figure does not include fault-tolerant I/O devices. Following Lovric's model,[7] we make the following assumptions: The system performs input and output by reading and writing, respectively, designated memory areas. Furthermore, computation proceeds in rounds, with the result of a deterministic function F being calculated in each round. To facilitate the detection of crash faults, we place an upper limit on the time required to calculate F. Using these assumptions, we identify the following components of a virtual duplex system:

- V_1 and V_2 are two different implementations that compute function F on input e. The implementations should be diverse; that is, they should originate from independent development teams, also a requirement for conventional duplex systems. Systematic diversity further enhances diversity. The two implementations encode result $F(e)$ with different error-correcting codes and deliver both encoded and both unencoded results to the error checker. Result $F(e)$ is a hash value over the state of the respective process. The states of V_1 and

V_2 are denoted by P and Q.

- The error checker compares results from V_1 and V_2 and signals a fault on non-equality. First, the error checker ensures the integrity of the two coded results by checking that both results are valid code words. Next, it compares the two results and signals possible differences as faults.
- The scheduler arranges execution of V_1 and V_2. In the absence of crash faults, V_2 executes after V_1 has finished. In the case of crash faults in V_1 or V_2, S signals an error to the error checker.

Systematic diversity, an extension of traditional design diversity, maintains a program's semantics. Lovric proposed mechanisms to improve diversity in design (specification), tools (compiler switches), and source code modification (high-level language and assembler). Some of these mechanisms are system dependent because they use characteristics of processor hardware and development tools. An evaluation of systematic diversity on two medium-complexity programs (quality control and interconnection network management) yielded good results; on average, all transient as well as 99.94 percent of permanent hardware failures were detected.[2] Lovric's work did not include fault recovery. However, well-known techniques uch as checkpointing can achieve fault recovery in virtual duplex systems and duplex systems alike. A simple variant is to store the state from time to time. In case of a fault—that is, in case of different states—a third instance of F is executed, starting from the last stored (valid) state. Then a majority vote over the three state encodings selects two identical states from which the application can continue.

Emulated multithreading

Until now, virtual duplex systems have used traditional heavyweight processes associated with operating systems. Because switching between these processes is costly, the typical grain size is on the order of tens of thousands of executed instructions. Modern operating systems also support lightweight processes—threads—which share some of the resources associated with heavyweight processes, such as the address space. This resource sharing makes switching between lightweight processes less costly than between heavyweight processes, so that grain sizes on the order of thousands of instructions are possible. Posix threads are a popular form of lightweight processes supported by almost all modern operating systems.[8] Mueller reports context switch times of 1,480 and 4,920 cycles, respectively, for Posix threads and processes under the Solaris operating system, measured on a Sparcstation IPX.[9]

However, these threads' overheads and corresponding grain sizes are still too high for our purposes. Fine-grained multithreading—grain sizes between one and hundreds of instructions—requires low-overhead context switches, as provided by emulated multithreading.[3,4]

In emulated multithreading, each thread is defined by its context and the program counter—that is, the thread's current location in the program code. The context is defined as the thread's processor state—the registers visible to the application programmer. The context typically contains all general-purpose registers as well as any special registers such as condition codes. The program code is divided into instruction blocks ranging in size from one to several hundred instructions. These instruction blocks are modified during compile time as follows: Each modified instruction block contains the necessary instructions to save or restore the part of the context that is modified or read, respectively, by the instructions in the original instruction block. In contrast, traditional thread packages save or restore the whole context when execution switches between threads. To execute a given thread, a scheduler program calls and executes the modified instruction block pointed to by the thread's program counter; execution then switches to the next thread. Virtual duplex systems don't require complex scheduling algorithms, thus ensuring low overhead. Therefore, the main routine from which all threads execute consists of a simple loop that calls the modified instruction blocks in a round-robin manner.

In addition, the compiler can spread the operations to save and restore registers along the instruction block, possibly filling empty instruction slots: Register saving can be performed directly after the last write to that register, whereas register restoring can be

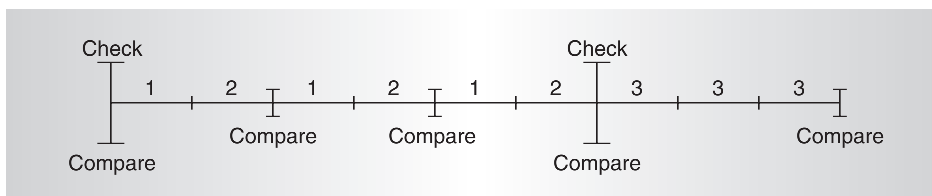

Figure 2. Thread scheduling in our approach.

postponed until the first read of that register. An examination of programs from NASA's NAS Parallel Benchmarks 2.0 showed that on average a register was used last after 70 percent of the instruction block length.[10] We can further decrease the number of save/restore operations; for example, it is not necessary to restore a register if the first access to that register is a write.

Our current implementation uses several tools that enable emulated multithreading:

- *hllconv*, a high-level language converter that processes the high-level language (C or Fortran) source code to identify procedures that must be modified;
- *asmconv*, an assembler converter that modifies the corresponding assembler instruction sequences; and
- *emulib*, an emulation library that contains a lightweight thread package and is linked to all programs after modification by the two converters.

In addition to these tools, we use the standard development tools (compiler, assembler, and linker) in the design flow.

Embedded virtual duplex systems

In embedded environments, virtual duplex systems should work as follows: As in conventional virtual duplex systems, a single node alternately executes two different implementations of function *F*. Instead of processes, the system uses threads in the form of emulated multithreading. In total, we provide three different implementations instead of two because we want to tolerate faults. All implementations are modified as described earlier. We ensure diversity between different implementations in several ways: First, we use traditional design diversity as well as systematic diversity during the design phase. If the implementations already use emulated multithreading (that is, if they contain calls to the emulation library), no modifications of the high-level language source code are necessary. Then, we use systematic diversity techniques such as transformations of the high-level language code to generate assembler sources. We modify these sources as described in the preceding section.

Next, we apply systematic diversity techniques at the assembler language level and generate executables. As Lovric proposes, all implementations use error-correcting codes and send encoded as well as unencoded data to the error checker. In addition, a simple signature function, such as the sum of all integers modulo 256, calculates a signature of the thread's state at the end of each instruction block. The system uses these signatures later to detect faults in a single round. Therefore, we must use a common subset of the state to calculate the signature. The signature also serves in checking whether a faulty thread has modified another thread's data, which is possible because there is no memory protection between threads as there is between processes.

Figure 2 depicts thread scheduling in our approach. Instruction blocks from the first two implementations (labeled 1 and 2) execute alternately. After execution of both instruction blocks, a signature check (*labeled Compare*) compares the state of the two threads. After a selectable number of instruction blocks, a checkpoint (*labeled Check*) is generated in addition to the normal signature check Compare; that is, both threads' states are saved to disk to enable fault tolerance. Thus, the system frequently compares the two states' threads without always incurring checkpoint generation overhead.

In an application mentioned earlier, the embedded system controls an automated guided vehicle that uses a camera to track its movements. To keep the vehicle on track, the embedded system constantly issues control instructions based on the camera input. In the

presence of real-time requirements and mission-critical control operation, the system issues the next control instruction only in the absence of faults. Therefore, the time between signature checks is bounded by the maximum time between two control instructions—that is, by the vehicle speed. This restriction requires small block sizes, fast context switches, and fast signature generation and comparison. Decreasing the block size while maintaining the same context-switching time leads to a proportional overhead increase, which is usually undesirable.

Frequent comparison leads to faster fault detection since faults can be detected only at the end of instruction blocks. Because checkpointing is an expensive operation, the frequent use of checkpoints could lead to unacceptably high overhead. This might even be true of diskless checkpointing, which trades checkpointing overhead for memory consumption. In the event of errors, the system enables the third implementation (labeled 3 in Figure 2) and rolls back its state to the last known checkpoint. After the thread reaches the point where the previous error was detected, a majority vote disables the faulty thread. Because all three threads are diverse implementations, the remaining two threads are still diverse and can continue operation.

We can tailor our virtual duplex system to an application's requirements with several parameters: First, we can select the grain size (an instruction block's maximum length). Signature checks occur between instruction blocks, so grain size determines the frequency of those checks. We can calculate the maximum time required for each block at compile-time, assuming a worst-case scenario. Second, the ratio of checkpoints to signature checks is also selectable. More-frequent checkpoints incur more overhead but reduce the minimum time required to resolve faults. Our system uses three threads, but only two are active in the absence of errors. To resolve a fault, we don't generate a new thread but activate the third thread instead.

Because traditional Posix threads as well as our threads use resource sharing rather than processes, some problems arise. The first is that quasi-parallel execution requires synchronization between threads that modify shared data structures. Otherwise the atomicity of updates cannot be ensured. The input values used by all threads are only read and therefore require no synchronization. If shared data structures are used, the design requirements must ensure proper synchronization (locks, semaphores, and so forth) between threads. Alternatively, synchronization is not necessary if updates to shared data structures do not cross instruction block boundaries. For instance, the programmer can mark such updates, allowing the assembler-level converter to use this information while generating instruction blocks. However, there is little reason to believe that the different threads require read/write access to shared data structures, since they are diverse, not cooperating, implementations of one function *F*.

The second problem is the lack of protection between threads. Because threads can access one another's' data structures, a faulty thread can influence other threads. In our approach, we partition address space between threads as follows: We map input values used by all threads to a portion of the address space shared by all threads. Because neither the threads nor the error checker require write access to these values, the corresponding memory pages can be mapped as read-only. Therefore, every write access causes a segment violation. The same reasoning can be applied to output values; the corresponding memory pages can be mapped as write-only, but writes to wrong outputs can still occur. We cannot handle self-modifying code, so write access to the threads' program code is not required. Again, the corresponding memory pages are mapped as read-only, so that threads can only access one another's' data structures, not their program codes. Every thread, as well as the error checker and scheduler, uses its own portion of the address space, which contains heap, stack, and frame information. To protect accesses, we could separate the different portions of the address space with read-only guard pages. However, virtual duplex systems don't rely on protection between processes anyway, because they use error-correcting code.[7]

Modifications at the assembler level can increase code size by adding instructions associated with context switches to the instruction stream. The number of additional instructions depends on the grain size. Larger grain sizes lead to less frequent context

switches and therefore fewer additional instructions. However, instead of using a full-blown thread library, we use a small library that can limit the increase in code size.

Evaluation

To evaluate our approach, we pursued two paths. First, we fixed the processor's performance and calculated how much faster we could switch contexts in a virtual duplex system when we compare emulated threads and Posix threads. Second, we fixed an application's context-switching frequency and calculated the performance requirements of several types of duplex and virtual duplex system implementations. We performed both calculations in a parameterized way first and applied some realistic values to these parameters afterwards. We used the following parameters:

- t_H, time required to compute a hash value of the state;
- t_S, time required to compute a signature on unchanged data;
- t_P, time required to switch between two processes;
- t_T, time required to switch between two Posix threads;
- t_E, time required to switch between two emulated threads; and
- t, time spent on useful work.

All parameters can be expressed as multiples of t_H; that is, $t = \beta \times t_H$ and $t_X = \alpha_X \times t_H$, where X = S, P, T, E, and β and $\alpha_X \geq 0$ are reals.

Typical parameter values

Concerning the time to compute a signature, we assume that the methods used are of similar complexity to those used when computing a hash value of the state. Furthermore, we assume that the sizes of the state and the unchanged data are of the same order. Hence, we assume $t_S \approx t_H$; that is, $\alpha_S = 1$.

Concerning α_P, we start by bounding the absolute time for a context switch, which is $t_P \geq 5$ µs, even for a process with a small size.[11] So even if t_H should reach a value of 12.5 µs, which is considered large, we would still have $\alpha_P \geq 0.4$. Normally, however, we would assume $t_P \geq t_H$. That is, $\alpha_P \geq 1$ because computing the hash value consists of browsing over all memory containing part of the state and performing some arithmetic on it, while switching process context involves storing all memory contents relating to that process.

Concerning the context-switching time for Posix threads, we assume $t_T \leq t_P$ because switching between threads does not involve more operations than switching between processes. In Linux, we have $t_T = t_P$ because Linux implements each Posix thread as a process of its own.[11] Under Solaris, a value reported is $t_P \approx 4 \times t_T$.[12] Hence, we assume $\alpha_T = \alpha_P$ or $\alpha_T = 0.25 \times \alpha_P$.

Regarding our own implementation of threads, the overhead is so small compared with the times t_P and t_T, that we dare to assume $t_E << t_H$, and thus we assume $\alpha_E = 0$.

Parameter β models the relationship between useful work (t) and overhead (t_H) in a duplex system. We assume $\beta = 10$, a 10 percent overhead.

Fixed processor

Let's assume a processor running a virtual duplex system with Posix threads, in which a context switch occurs after a thread has operated for time t. The time needed to compare states and switch contexts is $t_1 = t_H + t_S + t_T$. When we employ a virtual duplex system with emulated threads instead, this value changes to $t_2 = t_H + t_S + t_E$. The former system can switch once in time $t + t_1$, whereas the latter system can switch once in $t + t_2$. Hence, in $t + t_1$, the latter system can switch A times, where A is

$$\begin{aligned} A &= (t + t_1)/(t + t_2) \\ &= (t + t_H + t_S + t_T)/(t + t_H + t_S + t_E) \\ &= (\beta + 1 + \alpha_S + \alpha_T)/(\beta + 1 + \alpha_S + \alpha_E) \end{aligned} \quad (1)$$

A is the context switch frequency of a virtual duplex system with emulated threads, expressed as a multiple of the context switch frequency of a virtual duplex system with Posix threads. We assume that $t_S \approx t_H$, or that $\alpha_S = 1$; that t_E is close to 0, or that $\alpha_E = 0$; and that $\beta = 10$; then Equation 1 simplifies to

$$A = 1 + (\alpha_T/12) \quad (2)$$

Figure 3 depicts Equation 2. For values $\alpha_T \geq 3$, there is at least a 25 percent gain. Even for a Posix thread system whose context switch is quite fast compared with the state comparison, or stated the other way around, whose

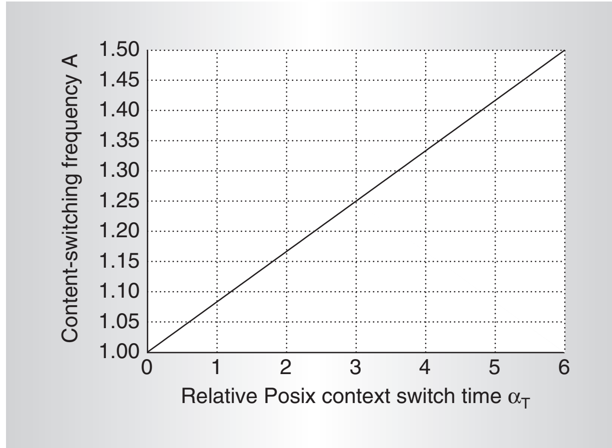

Figure 3. Relative context-switching frequency *A* of virtual duplex systems on fixed processors.

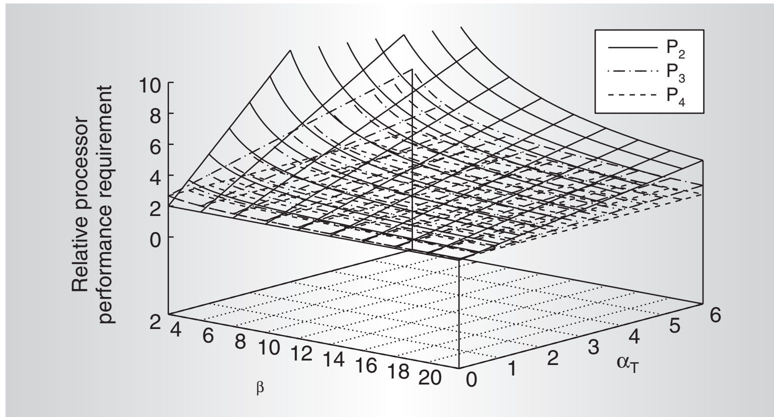

Figure 4. Relative processor performance requirements for various virtual duplex systems with fixed context-switching frequencies.

state comparison takes much longer than a context switch, there is still a noticeable 3.3 percent gain; for $\alpha_T = 0.4$ we obtain $A = 1.03$.

Fixed context-switching frequency

To perform this evaluation, we start with a duplex system consisting of two processors. Here, the overhead consists of computing a state hash value. To get a valid starting point, we must relate the overhead to the amount of work done between state checks. Therefore, we specify that in the duplex system, a state check taking time t_H shall occur after time $t = \beta t_H$ of useful work.

We express the performance of all other processors used in this evaluation in relation to the processors used here. That is, the processors used here have performance $P_1 = 1$ and perform a piece of useful work and a state check in time $T_1 = t + t_H = (\beta + 1)t_H$.

A virtual duplex system based on processes must perform, with each of two processes, the useful work, a state check, and a context switch. This takes time $T_2 = 2(t + t_H + t_P) = 2(\beta + 1 + \alpha_P)t_H$. To perform this work in time T_1, the processor needs a relative performance of $P_2 = T_2/T_1 = 2 + (2 \times \alpha_P)/(\beta + 1)$. We implicitly assume that memory bandwidth scales with processor performance, or that at least memory bandwidth is not the limiting performance factor.

For a virtual duplex system based on Posix threads, overhead increases by t_S for each thread, and the context-switching time changes from t_P to t_T, leading to

$$\begin{aligned} T_3 &= 2\left(t + t_H + t_S + t_T\right) \\ &= 2\left(\beta + 1 + \alpha_S + \alpha_T\right)t_H \\ P_3 &= \frac{T_3}{T_1} = 2 + \frac{2\left(\alpha_T + \alpha_S\right)}{\beta + 1} \end{aligned}$$

For a virtual duplex system based on emulated multithreading, context-switching time changes from t_T to t_E, leading to

$$\begin{aligned} T_4 &= 2\left(t + t_H + t_S + t_E\right) \\ &= 2\left(\beta + 1 + \alpha_S + \alpha_E\right)t_H \\ P_4 &= \frac{T_4}{T_1} = 2 + \frac{2\left(\alpha_E + \alpha_S\right)}{\beta + 1} \end{aligned}$$

We already see that under Linux, where $t_T = t_P$, an implementation based on Posix threads will be slower than one based on processes. When we use typical values $\alpha_S = 1$, $\alpha_E = 0$, and $\alpha_P = 4\alpha_T$, we get

$$\begin{aligned} P_2 &= 2 + \frac{8\alpha_T}{\beta + 1} \\ P_3 &= 2 + \frac{2\alpha_T + 2}{\beta + 1} \\ P_4 &= 2 + \frac{2}{\beta + 1} \end{aligned} \qquad (3)$$

Figure 4 depicts the curves from Equation 3. Figure 5 shows a cut at $\beta = 10$, our typical

value. For this value, the denominators in Equation 3 have a value of 11.

Hence, for our typical values, a virtual duplex system based on emulated multithreading needs a processor only about 2.2 as fast as the processor a duplex system requires. However, the duplex system would need two processors! For a virtual duplex system based on Posix threads, for $\alpha_T \leq 5$, we have $P_3 \leq 3$. Even this may be an acceptable choice. We present an overview of the available options in the next subsection.

The gap between duplex systems and virtual duplex systems narrows if we also consider performance in the event of an error. Assuming that state is saved after each comparison, the presented solutions lose a performance factor of 2 during error recovery time in a duplex system and 1.5 in a virtual duplex system. The reason for this is that in both systems, a third instance must be run from the last checkpoint to reach a valid state again before normal work can continue.

If we require both systems to recover with equal speed, P_1 increases by a factor of 2/1.5 = 4/3, and the relative processor performance requirement of a virtual duplex system based on emulated multithreading shrinks to about $P_4' = 1.64$. This means that with emulated multithreading, we need one processor that must be less than twice as fast as the processor in a duplex system, which needs two processors. For a virtual duplex system based on Posix threads, we obtain $P_3' \leq 2$ for $\alpha_T \leq 8/3$. Hence, for thread-switching times that are not too large compared with state-checking times, these virtual duplex systems also require one processor less than twice as fast as a processor in a duplex system.

Generally, if we allow performance to shrink by a factor of s during recovery, where $1 \leq s \leq 2$, the performance required by a duplex system increases by a factor of $2/s$, and the performance required by virtual duplex system increases by a factor of $1.5/s$ for $s \leq 1.5$. Hence, in this case also, the performance ratio decreases by a factor of 0.75 for $s \leq 1.5$, leading to similar results as those for the preceding described systems. For $s > 1.5$, the performance requirement decreases by a factor of $s/2$. For virtual duplex systems based on Posix and emulated threads, Figure 6 depicts the relative processor performances required if performance is allowed to shrink by s during recovery. The figure reflects this discussion because the function values are independent of s in the interval [1 … 1.5], and from that point on increase linearly with s in the interval [1.5 … 2]. If we perform a cut at $s = 2$, we again obtain Figure 5, without the curve for P_2.

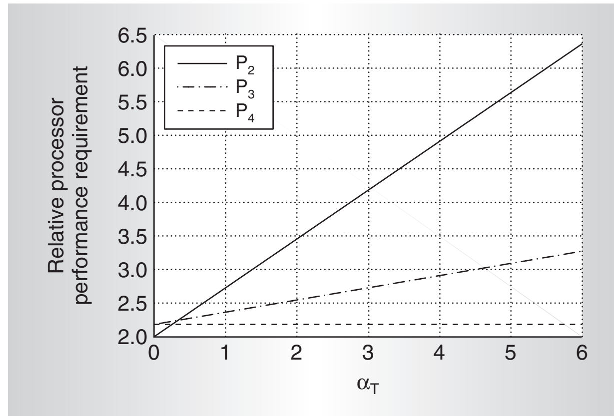

Figure 5. Relative processor performance requirements from Figure 4 for $\beta = 10$.

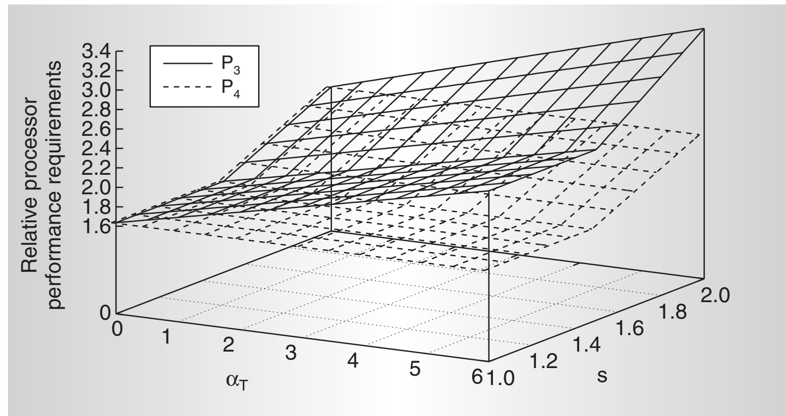

Figure 6. Relative processor performance requirements for recovery performance reduced by s.

Evaluation summary

Our evaluation indicates that in the case of a given hardware platform, a virtual duplex system based on emulated multithreading might be able to switch contexts more often than one based on Posix threads. If the speed of fault detection and recovery is the dominating design goal, emulated multithreading is preferable. However, if conformity to standards is an issue, Posix threads are an alternative if the performance difference can be

tolerated. Whether a difference is visible at all depends primarily on β, the amount of permissible overhead. If β is very large and thus the overhead very small, the gain obtainable by using emulated multithreading diminishes, and Posix threads provide an easier method. If β is small, emulated multithreading can provide performance benefits of up to 25 percent.

Our evaluation further shows that for a given context-switching frequency, several options exist, depending on the optimization goals and constraints. For a performance segment in which increasing processor performance is expensive—more than doubling the price—a duplex system is preferable because it allows for slower processors, two of which are still cheaper then one fast processor. You can derive the exact amounts the required processor performance must be increased from the P_i and P_i' curves in Figures 4 and 6. Those curves in turn depend on parameters given by the application and the platform: namely, the complexity of context switches and signatures and the permissible performance degradation during recovery.

However, for a performance segment in which increasing the processor performance is cheap, virtual duplex systems are preferable over duplex systems because one faster processor is cheaper than two slower ones. Note that price not only means the price of the processor chip but also includes power consumption and space on the printed circuit board.

The decision as to which virtual duplex system to use depends on the platform: On Linux systems, Posix threads are not expected to be faster than processes. In general, emulated multithreading requires the smallest increase in performance requirements, but it requires more software support than standardized implementations of threads. Hence, performance and programming constraints influence the decision.

The relevance of thread-based virtual duplex systems further increases with the commercial introduction of multiprocessors that support multiple threads in hardware, such as the Intel Pentium 4 Hyperthreading. Our further work tries to explore the benefits of using these processors. MICRO

Acknowledgments

We thank Klaus Echtle, Peter Sobe, and the anonymous reviewers for helpful comments.

References

1. K. Echtle, B. Hinz, and T. Nikolov, "On Hardware Fault Diagnosis by Diverse Software," *Proc. 13th Int'l Conf. Fault-Tolerant Systems and Diagnostics*, Verlag der Bulgarischen Akademie der Wissenschaften, 1990, pp. 362-367.
2. T. Lovric, "Detecting Hardware Faults with Systematic and Design Diversity: Experimental Results," *Int'l J. Computer Systems Science and Engineering*, vol. 11, no. 2, 1996, pp. 83-92.
3. A. Grävinghoff and J. Keller, "Fine-Grained Multithreading on the Cray T3E," *High-Performance Computing in Science and Engineering*, LNCS, Springer Verlag, 2000, pp. 447-456.
4. A. Grävinghoff, *On the Realization of Fine-Grained Multithreading in Software*, doctoral dissertation, Universität Hagen, Germany, 2002.
5. T. Lovric, "Systematic and Design Diversity—Software Techniques for Hardware Fault Detection," *Proc. 1st European Dependable Computing Conf.*, Springer Verlag, 1994, pp. 309-326.
6. T. Lovric, "Dynamic Double Virtual Duplex System: A Cost-Efficient Approach to Fault-Tolerance," *Proc. 5th Int'l Working Conf. Dependable Computing for Critical Applications*, IEEE Press, 1995, pp. 35-42.
7. T. Lovric, *Fehlererkennung durch systematische Diversität in entwurfsdiversitären zeitredundanten Rechensystemen und ihre Bewertung mittels Fehlerinjection (Fault detection through systematic diversity in design-diverse time-redundant computing systems, and its evaluation by fault injection)*, doctoral dissertation, Universität Essen, Germany, 1996.
8. D.R. Butenhof, *Programming with POSIX Threads*, Addison-Wesley, 1997.
9. F. Mueller, "A Library Implementation of POSIX Threads under UNIX," *Proc. 1993 Winter USENIX Conf.*, Usenix Assoc., 1993, pp. 29-42.
10. A. Grävinghoff and J. Keller, "Virtual Duplex Systems in Embedded Environments,"

Proc. Architecture of Computing Systems (ARCS 99), *Workshop Reliability and Fault Tolerance*, VDE Verlag, 1999, pp. 69-77.

11. B. Chelf, "The Fibers of Threads," *Linux Magazine*, May 2001, pp. 64-73.
12. K. Lai and M. Baker, "A Performance Comparison of UNIX Operating Systems on the Pentium," *Proc. USENIX Ann. Tech. Conf.*, Usenix Assoc., 1996, pp. 265-278.

Jörg Keller is an associate professor of computer science at FernUniversität Hagen, Germany. His research interests include parallel algorithms and architectures, fault tolerance, IT security, and new media in distance teaching. Keller has MS and PhD degrees in computer science from Universität des Saarlandes, Saarbrücken, Germany. He is a member of the IEEE Computer Society and the German Society of Informatics (GI).

Andreas Grävinghoff is a research staff member at ETAS GmbH, Stuttgart, Germany. His research interests include parallel programming systems and hardware design. Grävinghoff has an MS in computer science from Universität des Saarlandes, Saarbrücken, Germany, and a PhD in computer science from FernUniversität Hagen, Germany.

Direct questions and comments about this article to Jörg Keller, FernUniversität Hagen, Fachbereich Informatik, LG Parallelität und VLSI, 58084 Hagen, Germany; joerg.keller@fernuni-hagen.de.

For further information on this or any other computing topic, visit our Digital Library at http://www.computer.org/publications/dlib.