White Paper

Six Design and Development Considerations for Embedded Multi-Processor Systems

By Dr. Kelvin Nilsen,
CTO, Atego

July 21, 2009

Six Design and Development Considerations for Embedded Multi-Processor Systems

Established over 40 years ago, Moore's law explains that processor capacity roughly doubles every two years. The doubling of capacity can be seen as doubling of memory sizes and processing speeds. In recent years, chip companies have found that the most effective way to profit from today's wealth of transistors is to place multiple processing cores on the same chip. This paper discusses some of the issues associated with targeting symmetric multi-processor (SMP) platforms for embedded and real-time software systems. Most modern multi-core chips are structured as SMP systems, with very efficient (on-chip) communication between processors.

1. Industry Trends Regarding Multi-core Hardware

Though Moore's law is usually expressed in terms of raw computing capacity, the theoretical origins of the law are most directly related to the transistor capacity of a chip. A chart of the last 40 years of microprocessor evolution highlights this point. (See Figure 1). The straight line represents Moore's law, calibrated to double transistor counts every two years.

With more transistors, computer engineers are able to pack more sophisticated logic and larger caches on the chip. Over the years, these transistors have been used to increase typical data path size from 4 bits to 64 bits, add math coprocessors and specialized math instructions, implement multiple instruction dispatch and dynamic instruction scheduling, implement deep pipelines and superscalar operation, and implement speculative execution and branch prediction. These transistors have also been used to expand the sizes of memory caches. Most of these hardware optimizations have served to automatically identify and exploit opportunities for parallel execution that is inherent in a single application's instruction stream.

As more transistors are placed on a single chip, the connecting wires between adjacent transistors also shrink. This means less time is required to propagate a signal from one transistor to the next. This enables processors to run at higher clock frequencies, meaning less time is required to execute each instruction.

Industry Trends Regarding Multi-core Hardware

Until about the year 2000, nearly all high-volume microprocessors were single core solutions, drawn blue in figure 1. Note that starting in about 1995, single-core microprocessors have typically fallen below the Moore's law prediction. In some cases, they are significantly below it. There are several reasons for this trend.

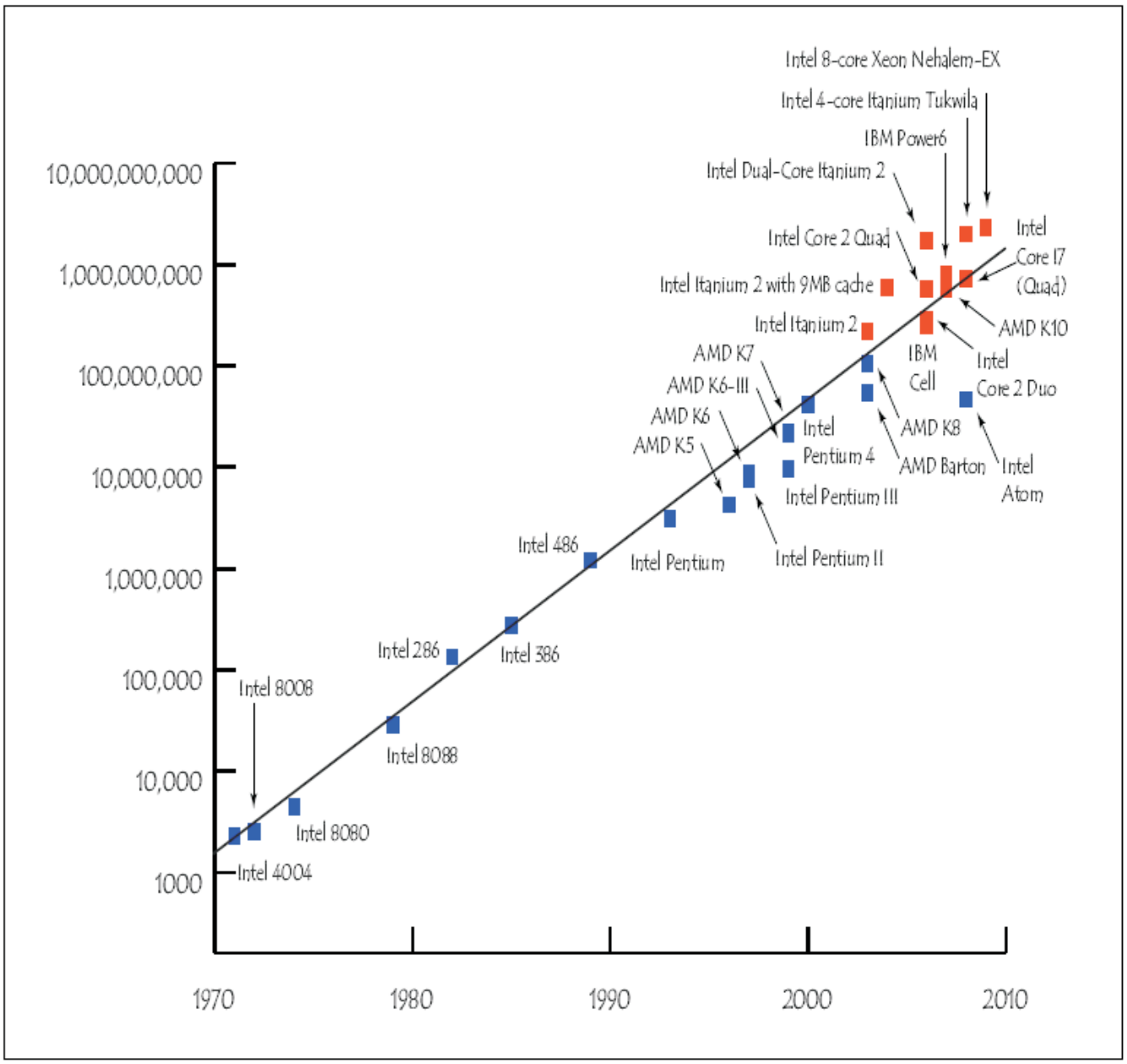

Figure 1. Transistors versus year of microprocessor

1. With regards to adding more transistors to expand memory caches and/or increase the sophistication of on-line instruction scheduling and speculative execution, CPU designers have reached the point of diminishing returns. Certainly, CPU designers could have continued adding larger caches in order to sustain Moore's law, but these larger caches would not have returned benefits consistent with their costs.

2. Power consumption is proportional to clock frequency and to the square of a processor's supply voltage [2]. With higher clock frequency, a processor's supply voltage must also increase. Though power considerations have traditionally been the domain of embedded device developers, the issues of power consumption and heat dissipation are of increasing concern in the realm of high-end server farms as well. Until the shift towards multi-core processing, power consumption has scaled proportional to advances in performance. This means each doubling of performance has doubled power and heat dissipation costs. Projected costs for large server farms, such as those used by Google, revealed that over the lifetime of a particular network of servers, the cost of electricity will soon exceed the cost of the hardware itself [3]. Switching to multi-core architectures changes this dynamic. The dual-core AMD Opteron 275, for example, runs 1.8 times faster on certain benchmarks than its single-core equivalent, the Opteron 248. The dual-core processor uses only 7% more power. Thus, the performance-per-watt rating of the dual-core architecture is roughly 68% better than the single-core architecture [3]. More aggressive results are possible with more radical departures from traditional single-core approaches.

3. Related to the objective of reducing power consumption when the system is operating at full speed, another design objective especially relevant to embedded developers with size, weight, and power constraints is the desire to efficiently reduce power consumption when the processing demands on the system are lower, such as during standby operation. Multi-core architectures provide options to completely turn off certain cores to reduce total power consumption without totally disabling the system [4].

4. A fourth motivating consideration is that as clock rates increase, the difference between CPU speeds and off-chip speeds expands, requiring additional circuitry (caches, hyperthreading, speculative prefetching) to mitigate this speed differential [5]. The typical multi-core system runs at a slower clock rate than the most modern single-core solutions, and achieves its higher performance by exploiting explicit parallelism manifest in the application's structure. By running the individual cores at lower clock frequencies, less circuitry is required to

manage the relatively slow interfaces to memory and I/O devices.

The motivations for the move towards multi-core architectures are easy to understand and the industry trends cannot be denied. For the foreseeable future, new processor designs will almost always feature multi-core capabilities. And single-core architectures are being phased into obsolescence. Like it or not, software engineers are going to have to befriend these new multiprocessing platforms.

2. Multi-core Demands on Software

Developing software for deployment on multi-processor platforms is very different than traditional development for uniprocessors. Kunle Okukotun, Director of the Pervasive Parallelism Lab at Stanford University, recently explained “Anything performance-critical will have to be rewritten. Eventually, you either have to rewrite that code or it will become obsolete [6].” Alex Backmutsky, a system architect and author of an upcoming book on telecommunication system design, adds “This is one of the biggest problems telecom companies face today. Their apps were not written for multiple cores and threads, and the apps are huge; they have millions and tens of millions of lines of code [6]." The challenges are several fold. Among them, software engineers targeting a multi-processor must:

1. Assure that the application is represented as a collection of independent threads or tasks, with minimal information sharing and coordination requirements between these tasks.

2. Design and implement efficient and reliable task coordination mechanisms.

3. Address load distribution and load balancing issues.

4. Analyze inter-task contention and schedulability in order to demonstrate compliance with real-time constraints.

5. Structure their software solutions so that they are easily maintained and ported between distinct multi-processor implementations.

One of the challenges with porting software to a multi-processor platform is that the maximum speedup available through running an application in parallel on multiple processors is limited by Amdahl’s law [8, 9]. For any given application, certain parts of the application are inherently sequential and cannot run in parallel on multiple

processors. Let Q represent the percentage of the application that is inherently sequential. Let N represent the number of processors comprising the targeted multi-processor. Amdahl's law states that the maximum speedup, S, available on this multi-processor is limited by the following mathematical expression:

$$S \leq \frac{1}{Q + (1 - Q) / N}$$

Suppose, for example, that Q = 5% and N = 4. Applying Amdahl's law, we find that the maximum speedup, S, is 3.5. Increasing N to 8 changes the limit to 5.9. Though Amdahl's law may limit overall speedup, it does not necessarily limit processing capacity. Consider, for example, a software-implemented telephone switch which involves a significant amount of sequential work to set up each call, and then a comparable sequential amount of work to tear down the call. Running this telephone switch software on a multi-processor will not decrease the time required to set up or tear down each call, but it will very likely increase the number of calls that can be initiated and terminated in any given time span.

In the quest to achieve the theoretical speedup limits of Amdahl's law, most legacy software must be restructured to expose the potential for parallel execution paths. Tools that automatically detect parallelism inherent in numeric matrix algorithms provide limited success with automatic generation of parallel code for those specialized algorithms [7]. However, the generality required by typical information processing applications, with complex linked data structures and data structure traversals, makes it very difficult to do automatic parallelization of this code. Thus, most experts believe that manual translation of sequential algorithms is necessary in order to effectively exploit multi-processor parallelization of software systems. When manually translating sequential programs into parallel execution paths, software engineers need to target a programming language that allows them to express parallel constructs in concise notations that are robust, easily maintained, and scalable across a broad spectrum of multi-processor hardware configurations. Since the Java language was invented after the importance of parallel execution on multi-processors was well understood, it has features designed to facilitate multi-threaded programming.

"The ubiquitous C language is the worst [tool] because it is inherently sequential and obscures any parallelism inherent in an algorithm," said Jeff Bier, President of DSP consulting firm Berkeley Design Technology [6]. Both C and C++ suffer because the design of these languages never considered the special needs of multiprocessing. Various realtime operating system (RTOS) vendors have found ways to extend C into the domains of concurrent and fully parallel execution. However, this approach is hindered

because the extensions are bolted on to the language in the form of libraries, without any standards-based support from the language syntax or the compilers.

Over the years, C and C++ language compilers and the language definition itself have evolved to extract high performance from modern uniprocessors. Thus, the language allows caching values in registers and speculative and out-of-order execution. One important consideration, from the perspective of writing code for execution on multi-processors, is there is no clean way to disable these optimizations in selected contexts that might involve coordination with other processors. If there exist mechanisms to implement reliable information sharing between processors using particular C compilers with particular hardware and particular RTOS, those mechanisms are highly nonportable. Great difficulty is encountered when developers attempt to move the code to another RTOS, CPU, or C compiler.

3. Java Concurrency Constructs

The Java language was designed by Sun Microsystems during the early 1990s and released to the general public in 1995. At the time, Sun Microsystems had already made the shift to multi-processor architectures. It would appear that Sun Microsystems felt it was important that Java improve the ease with which multi-processor software could be developed and maintained. Various multiprocessing features were designed into the Java language. These features integrate at all levels of the Java implementation, manifesting themselves within the programming language syntax and source compilers, in special byte codes and constraints on implementation of the Java virtual machine, and throughout the standard libraries. Unlike C and C++, Java provides a strong foundation for building efficient, portable, and scalable applications and software components that fully exploit whatever multiprocessing capabilities are provided by the underlying hardware.

The concurrency features of Java are defined to provide compatible behavior on both uniprocessors and multi-processors. On uniprocessors, concurrency is implemented by time slicing and priority preemption. On multi-processors, multiple concurrent threads execute in parallel on different processor cores.

The remainder of this section provides an overview of the key concurrency features built into the Java platform. Many resources are available to explore more thoroughly. A particularly good source for studying concurrency topics related to Java is reference 10.

THE THREAD TYPE

A Java application is generally comprised of one or more independent threads of xecution. Each thread represents an independent sequence of instructions. All threads within a particular Java application can see the same memory. In the vernacular of POSIX [11], all of the threads running within a single Java virtual machine reside within the same process.

Defining a new thread in Java is straightforward. Since Java is an object-oriented language, the developer simply introduces a new Class extending the java.lang.Thread class, overriding the run() method with the body of code to be executed within this thread. See the excerpt below for an example.

```
public class MyThread extends java.lang.Thread {
    final private String my_id;
// constructor
public MyThread(String name) {
    my_id = name;
}
public void run() {
    int count = 0;
    while (true) {
        System.out.println("MyThread " + my_id + " has iterated " + count + " times");
        count++;
    }
}
}
```

To spawn this thread's execution, the program simply instantiates a MyThread object and invokes its start() method. The spawned thread will run at the default priority of 5. Change the priority, if desired, by invoking Thread.setPriority(). This is demonstrated in the following code sequence.

```
class MyMain {
    public static void main(String[] thread_names) {
        for (int i = 0; i < thread_names.length; i++) {
            MyThread a_thread = new MyThread(thread_names[i]);
            a_thread.start();
        }
}
}
```

For an invocation of this main program with command-line arguments bob fred sally, the first several lines of output were:

```
MyThread bob has iterated 0 times
MyThread bob has iterated 1 times
MyThread bob has iterated 2 times
MyThread bob has iterated 3 times
```

```
MyThread bob has iterated 4 times
MyThread fred has iterated 0 times
MyThread sally has iterated 0 times
MyThread bob has iterated 5 times
MyThread fred has iterated 1 times
MyThread sally has iterated 1 times
MyThread bob has iterated 6 times
...
```

SYNCHRONIZED STATEMENTS

Whenever multiple threads share access to common data, it is occasionally necessary to enforce mutual exclusion for certain activities, enforcing that only one thread at a time is involved in those activities. Consider, for an illustrative example, a situation where one thread updates a record representing the name and phone number of the person on-call to respond to any medical emergencies that might arise. Many other threads might consult this record in particular situations. An (incorrect) implementation of this record data abstraction is shown below:

```
public class OnCallRecord {
   private String name, phone;

   public void overwrite(String new_name, String new_phone) {
   name = new_name;
   // placing print statement here increases probability that we'll
   // experience a context switch after incompletely updating record
   System.out.println("overwriting OnCallRecord with: " + new_name +
   ", phone: " + new_phone);
   phone = new_phone;
}

public OnCallRecord clone() {
   OnCallRecord copy = new OnCallRecord();
   copy.name = this.name;
   copy.phone = this.phone;
}

public String name() {
   return name;
}

public String phone() {
   return phone;
}
}
```

The problem with the above code is that the overwrite() method may be preempted before the entire record has been updated. If another thread performs a clone() during this preemption, the other thread will see inconsistent data, perhaps obtaining one person's name paired with a different person's phone number. An excerpt of an

execution trace for an actual run involving one updating thread and one inquiring thread shows that this possibility is real.

```
[1]     overwriting OnCallRecord with: Mary, phone: 222-3333
[2]     Responsible party is: Mary, phone: 111-2222
[3]     overwriting OnCallRecord with: Mark, phone: 444-5555
[4]     Responsible party is: Mark, phone: 222-3333
[5]     overwriting OnCallRecord with: Sally, phone: 666-7777
[6]     Responsible party is: Sally, phone: 444-5555
[7]     Responsible party is: Sally, phone: 444-5555
[8]     Responsible party is: Sally, phone: 444-5555
[9]     Responsible party is: Sally, phone: 444-5555
[10]    Responsible party is: Sally, phone: 444-5555
[11]    overwriting OnCallRecord with: Sue, phone: 888-9999
[12]    Responsible party is: Sue, phone: 666-7777
```

Note that, for this execution trace, the phone number is consistently wrong. The fix for this problem is to add the synchronized keyword to each of the methods in the definition of OnCallRecord.

```
public class OnCallRecord {
   private String name, phone;

   public synchronized void overwrite(String new_name, String new_phone) {
      name = new_name;
      // placing print statement here increases probability that we'll
      // experience a context switch after incompletely updating record
      System.out.println("overwriting OnCallRecord with: " + new_name +
                         ", phone: " + new_phone);
      phone = new_phone;
   }

   public synchronized OnCallRecord clone() {
      OnCallRecord copy = new OnCallRecord();
      copy.name = this.name;
      copy.phone = this.phone;
   }

   public synchronized String name() {
      return name;
   }

   public synchronized String phone() {
      return phone;
   }
}
```

With this change, a similar execution trace shows much better behavior:

```
[1]  overwriting OnCallRecord with: Bob, phone: 123-4567
[2]  Responsible party is: Bob, phone: 123-4567
[3]  overwriting OnCallRecord with: Nancy, phone: 321-9876
[4]  Responsible party is: Nancy, phone: 321-9876
[5]  overwriting OnCallRecord with: George, phone: 111-2222
[6]  Responsible party is: George, phone: 111-2222
[7]  Responsible party is: George, phone: 111-2222
[8]  Responsible party is: George, phone: 111-2222
[9]  Responsible party is: George, phone: 111-2222
[10] Responsible party is: George, phone: 111-2222
[11] overwriting OnCallRecord with: Mary, phone: 222-3333
[12] Responsible party is: Mary, phone: 222-3333
[13] overwriting OnCallRecord with: Mark, phone: 444-5555
[14] Responsible party is: Mark, phone: 444-5555
[15] overwriting OnCallRecord with: Sally, phone: 666-7777
[16] Responsible party is: Sally, phone: 666-7777
```

One effect of adding the synchronized qualifier to a method is to cause the Java compiler to insert code into the method's prologue and epilogue to acquire a mutual exclusion lock associated specifically with the object to which the method is attached. Some comparisons with C (and C++) are appropriate:

1. Note that the implementor of the OnCallRecord has declared the name and phone fields to be private. This encapsulates access, assuring that the only way to modify or view the values of these fields is by means of the public methods associated with this data type. Thus, the implementor of OnCallRecord can assure that access to its internal data is properly synchronized. The Java compiler enforces that there is no way for other code outside the OnCallRecord abstraction to violate these constraints.

2. As a high-level programming abstraction, much of the low-level detail associated with managing synchronization locks is hidden from the developers. The Java virtual machine automatically decides when to allocate and deallocate lock data structures and has full control over when and how to coordinate with the underlying operating system.

3. Since the synchronized keyword applies to entire methods and statements, it is block oriented. The Java compiler enforces that the lock is acquired upon entry to the block and is released upon exit from the block. Unlike C, there is no way for a programmer to forget to release the lock. The Java platform assures that the lock will be released even if this block of code is exited abnormally, such as when an exception is thrown or a break statement leaves an inner-nested loop.

4. Since it is known by the Java run-time environment that this lock is associated

with mutual exclusion and nested locks are released in LIFO order, the Java run-time is able to reliably implement priority inheritance and even priority ceiling emulation for the synchronization locks. Priority inversion avoidance is not required by the Java language specification, but it is implemented by several real-time virtual machines.

5. Because the synchronized keyword is built into the language itself, the compiler and run-time environment are able to cooperate in providing very efficient implementations. The typical implementation of Java lock and unlock services is much more efficient than the RTOS system calls that are required to implement these services in typical C code.

WAIT AND NOTIFY

Within a synchronized method, Java programs have access to implicit condition variables associated with the object's synchronization lock. The typical concurrent Java programmer acquires an object's mutual exclusion lock by entering a synchronized method, checks for a certain condition and, if that condition is not satisfied, blocks itself by invoking the wait() service. When a thread executes wait(), it is placed in a blocked state on a special queue associated with the enclosing object. While on this queue, the thread temporarily releases its mutual exclusion lock.

Other threads operating on the same object will presumably make the condition true. Before making the condition true, the other thread also acquires the mutual exclusion lock. After making the condition true, but while still holding the mutual exclusion lock, the other thread performs a notify() (or notifyAll()) operation. This has the effect of unblocking one (or all) of the threads on the object's condition queue. Each unblocked thread automatically reacquires the mutual exclusion lock before it executes within the synchronized block that was running when it invoked wait(). The mutual exclusion lock will not be available until the notifying thread leaves its synchronized block.

The following abstract data type demonstrates the use of wait() and notify() services.

```
[1]   public class FIFO {
[2]         final private int buffer[]; // this FIFO buffers integers
[3]         private int first, count;
[4]
[5]         public FIFO(int size) {
[6]                     buffer = new int[size];
[7]                     first = 0; count = 0;
[8]         }
[9]
[10]        public synchronized void put(int value) {
[11]                    while (count >= buffer.length) {
```

```
[12]                        wait();
[13]                }
[14]                buffer[(first + count++) % buffer.length] = value;
[15]                if (count == 1) {
[16]                        notify();               // wake a waiting reader, if there is one
[17]                }
[18]        }
[19]
[20]        public synchronized int get() {
[21]        int result;
[22]        while (count <= 0) {
[23]            wait();
[24]        }
[25]        result = buffer[first++];
[26]        if (first >= buffer.length) {
[27]            first = 0;
[28]        }
[29]        if (count-- == buffer.length) {
[30]        notify(); // wake a waiting writer, if there is one
[31]        }
[32]    }
[33] }
```

In this sample code, a single wait queue represents two possible conditions. If the buffer is currently full, any threads on the wait queue must be waiting to insert new items. If the buffer is currently empty, any threads on the wait queue must be waiting to extract an item.

The notify() operations at lines 16 and 30 have no effect if no thread is currently waiting on the corresponding wait queue. The Java Language Specification does not specify which of multiple threads on the wait queue will be notified if multiple threads are on the queue. Certain real-time virtual machines sort the wait queue first by active thread 11 priority and second by time of arrival (FIFO), guaranteeing to always notify the highest priority thread that has been waiting the longest.

Providing the wait() and notify() services as built-in capabilities of the Java language encourages a style of programming that is easily understood and easily maintained. The common usage patterns are recognized as idioms by experienced Java developers. Idiomatic usage is desirable because it means the software engineer does not have to analyze all of the subtle thread interactions associated with each body of code. The lack of idiomatic usage for condition notification in C and C++ is one of the disadvantages of those languages for development of concurrent software.

VOLATILE VARIABLES

Like C and C++, the Java language allows programmers to declare that certain variables are volatile. One effect of declaring a variable to be volatile is that any fetch or store of this variable is forced to communicate with the memory subsystem. For variables not declared to be volatile, the variable, which might be cached in a machine register, could be updated many times by one thread while other threads consulting the variable would never see a change to its value.

The program below represents an alternative solution to the on-call physician problem. With this solution, the OnCallRecord is replaced each time a new physician is placed on call.

```
public class ImmutableOnCallRecord {
    private final String name, phone;
    public ImmutableOnCallRecord(String name, String phone) {
        this.name = name;
        this.phone = phone;
    }

    public String name() {
        return name;
    }

    public String phone() {
        return phone;
    }
}

public class ImmutableOnCallMain {
    static volatile ImmutableOnCallRecord shared_record;

    public static void main(String[] args) {
        OnCallScheduler ocs = new OnCallScheduler();
        ocs.start();

        while (true) {
            ImmutableOnCallRecord responsible = shared_record;
            if (responsible != null) {
                System.out.println("Responsible party is: " + responsible.name() +
                                                                    ", phone: " + responsible.phone());
            }
        }
    }
}
public class OnCallScheduler extends java.lang.Thread {
    public void run() {
        while (true) {
            ImmutableOnCallMain.shared_record =
                new ImmutableOnCallRecord("George", "111-2222");
            ImmutableOnCallMain.shared_record =
```

```
                new ImmutableOnCallRecord("Mary", "222-3333");
            ImmutableOnCallMain.shared_record =
                new ImmutableOnCallRecord("Mark", "444-5555");
            ImmutableOnCallMain.shared_record =
                new ImmutableOnCallRecord("Sally", "666-7777");
            ImmutableOnCallMain.shared_record =
                new ImmutableOnCallRecord("Sue", "888-9999");
            ImmutableOnCallMain.shared_record =
                new ImmutableOnCallRecord("Bob", "123-4567");
            ImmutableOnCallMain.shared_record =
                new ImmutableOnCallRecord("Nancy", "321-9876");
        }
    }
}
```

Note that the shared_record variable defined within the ImmutableOnCallMain class is declared volatile. This means that each time the variable is overwritten within the run() method of OnCallScheduler, the change is immediately visible to other threads. Further, it means that each time the variable's value is fetched within the ImmutableOnCall-Main.main() method, the value is fetched from shared memory rather than a locally cached register value. If this variable had not been declared volatile, a fully compliant Java implementation would be free to behave as if each running thread had its own private copy of this variable.

It is important to allow the Java run-time to cache shared variables, reorder instructions, and perform certain speculative operations because these behaviors are critical to achieving high performance within each thread. However, it is also very important to allow programmers to selectively disable these optimizations where doing so is necessary in order to achieve reliable sharing of information between independent threads.

One aspect of Java that is different, and more powerful, than C and C++ is its treatment of volatile variables. Not only does the volatile keyword force all uses of the variable to access shared memory, but this keyword also implies certain important semantics with regards to other non-volatile variables. In particular:

1. Since overwriting the value of a volatile variable most likely triggers communication of certain information held as part of this thread's local state to certain other threads, the operation is accompanied by a cache, memory, and instruction-ordering fence.
   a. Any actions that precede the volatile write within source code may not be reordered to actually occur after the volatile write operation.
   b. Any variables represented by machine registers which may hold values that were modified since they were last fetched from memory, must be copied back to memory before the volatile write operation is performed.
   c. If the underlying hardware allows the same shared variable to have different values in each

processor's cache of that memory location, any modified values held in this processor's cache must be copied back to memory before the volatile write operation is performed. Architectures that allow relaxed consistency generally provide explicit instructions to force cache flushes.

2. Since reading from a volatile variable typically represents the receipt of information communicated from another thread, this operation is also accompanied by a cache, memory, and instruction-ordering fence.
   a. If the underlying hardware allows the same shared variable to have different values in each processor's cache of that memory location, update this processor's cache by copying any variables that have not been locally modified from shared memory immediately following the read of the volatile variable. Architectures that allow relaxed consistency generally provide explicit instructions to force cache updates.
   b. Any potentially shared variables that are cached in machine registers are updated from shared memory immediately following the read of the volatile variable.
   c. Any actions that follow the volatile read within source code may not be reordered to actually occur before the volatile read operation.

Compared with C and C++, these refined semantics for dealing with volatile variables are critical for reliable sharing of information between distinct threads. It is quite difficult to precisely implement these semantics using existing C compilers and RTOS services. Even more difficult is to implement these semantics in a portable and maintainable way that would not inhibit the reuse of code on a different CPU, with a different C compiler and/or a different RTOS.

CONCURRENCY LIBRARIES

With the Java 5.0 release in 2005, the Java standard libraries included a new package named java.util.concurrent which provides support for more powerful concurrency capabilities. Included in this package and its two subpackages, java.util.concurrent.atomic and java.util.concurrent.locks, are libraries to implement thread-safe collections, semaphores, prioritized blocking queues, atomically modifiable data abstractions, and a reentrant lock data type. The reentrant lock data type offers services not available with synchronized statements, such as an API to conditionally acquire a lock only if the lock is not currently held by some other thread.

THE JAVA MEMORY MODEL

One of the more noteworthy differences between concurrent programming support in Java and C or C++ is that Java support is integrated within the language and the compiler, whereas C and C++ support is restricted to libraries, many of which are proprietary to each RTOS vendor. By integrating concurrency support within the compiler, special code generation practices enable greater efficiency without sacrificing reliable and portable sharing of information between threads.

As with reading and writing of volatile variables, entry and exit of a synchronized block has certain side effects with regards to visibility of variables that may be locally cached. In particular,

1. Exiting a synchronized block causes locally cached variables to be published for view by other threads.
   a. Any actions that are contained within the synchronized block or precede the synchronized block within source code may not be reordered to actually occur after control exits the synchronized block.
   b. Any variables represented by machine registers which may hold values that were modified since they were last fetched from memory must be copied back to memory before control leaves the synchronized block.
   c. If the underlying hardware allows the same shared variable to have different values in each processor's cache of that memory location, any modified values held in this processor's cache must be copied back to memory before control leaves the synchronized block. Architectures that allow relaxed consistency generally provide explicit instructions to force cache flushes.

2. Entry into a synchronized block causes this thread to see all data globally published by other threads.
   a. If the underlying hardware allows the same shared variable to have different values in each processor's cache of that memory location, update this processor's cache by copying any variables that have not been locally modified from shared memory immediately upon entry into the synchronized block. Architectures that allow relaxed consistency generally provide explicit instructions to force cache updates.
   b. Any potentially shared variables that are cached in machine registers are updated from shared memory immediately upon entry into the synchronized block.
   c. Any actions that are contained within the synchronized block or that follow the synchronized block within source code may not be reordered to actually occur before control enters the synchronized block.

The Java Memory Model describes a happens-before relationship between certain events which may occur in different threads. Data races may exist in Java programs that are not carefully structured so as to assure that the write operation happens before all of the read operations that are intended to see the value written. A correctly synchronized Java program will avoid race conditions by assuring that activities occuring in different threads are properly sequenced by happens before relationships. As examples of the happens-before relationships, the Java Memory Model specifies the following orderings:

1. An unlock of a monitor lock (exiting a synchronized block) happens before every subsequent locking of the same monitor lock.

2. Overwriting a volatile variable happens before every subsequent fetch of the same field's value.

3. Invocation of Thread.start() happens before every action performed within the started thread.

4. All actions performed within a thread happen before any other thread can detect termination of the thread. Other threads detect this thread's termination by invoking Thread.join() or Thread.isAlive().

5. A thread's request to interrupt another thread happens before the other thread detects it has been interrupted.

4. Migrating to Multi-core Java

For the past several decades, huge legacies of software have been created. Many embedded system industries have experienced doubling of application size every 18 to 36 months. Now, many of the companies responsible for maintaining these large legacies are faced by the difficult challenge of modernizing the code to run in multi-processor environments.

ONE OPTION: KEEP YOUR LEGACY CODE AS IS

Clearly, the path of least resistance is to keep large legacies of code as is, and not worry about porting the code to new multi-processor architectures. Most code originally developed to run as a single non-concurrent thread will run just fine on today's multi-processors, as long as there is no need to scale performance to more than one processor at a time.

Sometimes, you can scale easily to multi-processor deployment by running multiple instances of the original application, with each instance running on a different processor. This improves bandwidth by enabling a larger number of transactions to be processed in any given unit of time, but it does not generally improve latency (the time required to process each transaction).

Usually, even if it is possible to preserve large portions of an existing legacy application by replicating the application's logic across multiple processors, some parallel processing infrastructure must be developed to manage the independent replicas. For example, a supervisor activity might need to decide which of the N replicas is responsible for each transaction to be processed. And at a lower level, it may be necessary for all N replicas to share access to a common data base to assure that the independent processing activities do not create integrity conflicts.

In general, it is good to reuse time-proven legacy code as much as is feasible. However, in most cases, significant amounts of code must be rewritten and/or added in order to fully exploit the parallel processing capabilities of the more modern hardware. We recommend the use of Java for new code development for several reasons:

1. In general, Java developers find themselves to be about twice as productive as C and C++ developers during the creation of new (sequential) functionality.
2. During software maintenance and reuse activities, systems implemented with the Java language are maintained at one-fifth to one-tenth the cost of systems implemented with C or C++.
3. Java has much stronger support than C and C++ for concurrent and parallel processing.
4. There is a greater abundance of readily reusable off-the-shelf software components written in the Java language.
5. Most recently, it is much easier to recruit competent Java developers than developers expert in the use of C and C++.

BUG-FREE JAVA CODE

Since Java is designed for concurrent and parallel programming, many off-the-shelf software components and many legacy Java applications are already written to exploit concurrency. If you are lucky enough to inherit a legacy of Java application software, migration to a multi-processor Java platform may be straightforward.

A word of caution: even though Java code may have been written with concurrency in mind, if the code has never been tested on a true multi-processor platform, it may be that the code is not "properly synchronized". This means certain race conditions may

manifest on the multi-processor system even though they did not appear when the same code ran as concurrent threads on a uniprocessor.

Budgeting time to test and fix these sorts of bugs is advisable. As Java code is migrated from uniprocessor to multi-processor platforms, code review or inspections by peers may help identify and address shortcomings more efficiently than extensive testing. Enrolling your entire staff of uniprocessor-savvy Java programmers for training on multi-processor issues may be an investment that pays for itself many times over as part of the preparation for migrating existing Java code from uniprocessor to multi-processor platforms.

The remainder of this section discusses issues relevant to creating Java code for or porting code into a multi-processor environment.

FINDING OPPORTUNITIES FOR PARALLEL EXECUTION

To exploit the full bandwidth capabilities of a multi-processor system, it is necessary to divide the total effort associated with a particular application between multiple largely independent threads, with at least as many threads as there are processors. Ideally, the application must be divided into independent tasks, each having the relatively rare need to coordinate with other tasks. For efficient parallel operation, an independent task would execute hundreds of thousands of instructions before synchronizing with other tasks, and each synchronization activity would be simple and short, consuming no more than a thousand instructions at a time. Software engineers should seek to divide the application into as many independent tasks as possible, while maximizing the ratio between independent processing and synchronization activities for each task. To the extent these objectives can be satisfied, the restructured system will scale more effectively to larger numbers of processors.

Serial code is code that multiple tasks cannot execute in parallel. As software engineers work to divide an application into independent tasks, they must be careful not to introduce serial behavior among their “independent tasks” because of contention for shared hardware resources. Suppose, for example, that a legacy application has a single thread that reads values from 1,000 input sensors. A programmer might try to parallelize this code by dividing the original task into ten, each one reading values from 100 sensors. Unfortunately, the new program will probably experience very little speedup because the original thread was most likely I/O bound. There is a limit on how fast sensor values can be fetched over the I/O bus, and this limit is almost certainly much slower than the speed of a modern processor. Asking ten processors to do the work previously done by one will result in ineffective use of all ten processors rather than just one. Similar I/O bottlenecks might be encountered with file subsystem

operations, network communication, interaction with a data base server, or even a shared memory bus.

Explore pipelining as a mechanism to introduce parallel behavior into application that might appear to be highly sequential. See Figure 2 for an example of an application that

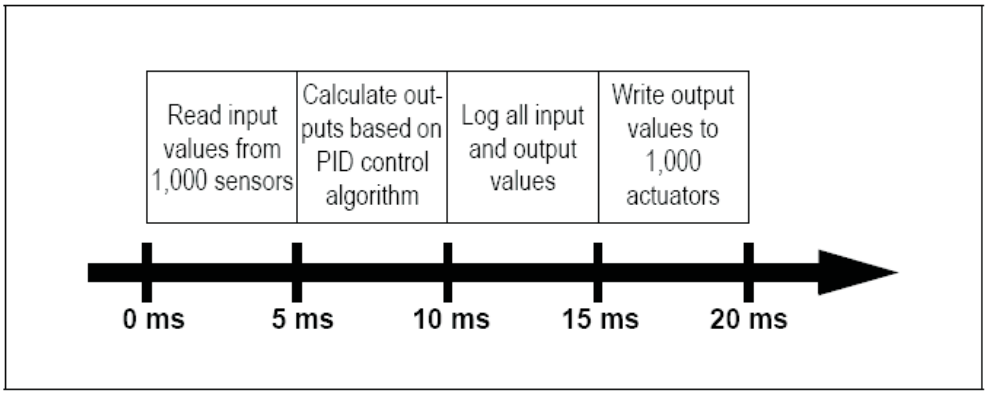

Figure 2. Soft PLC Implementation of Proportional Integral Derivative (PID) Control

appears inherently serial. This same algorithm can be effectively scheduled on two pro-processors as shown in Figure 3. Pipelined execution does not reduce the time required

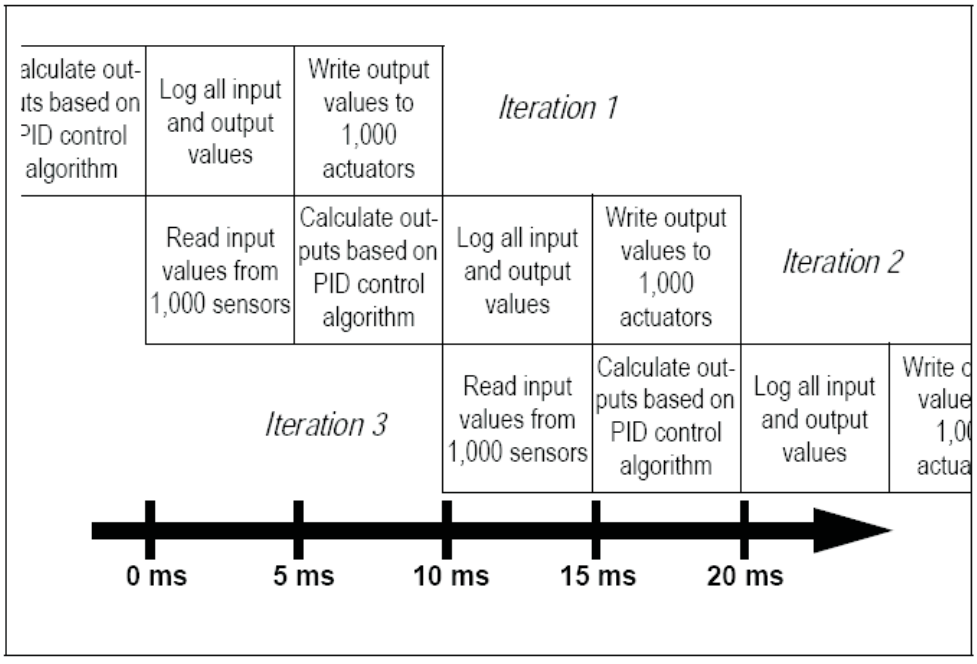

Figure 3. Two-Processor Pipelined Implementation of Soft PLC

to process any particular set input values. However, it allows an increased frequency of processing inputs. In the original sequential version of this code, the control loop ran once every 20 ms. In the two-processor version, the control loop runs every 10 ms.

This particular example is motivated by discussions with an actual customer who was finding it difficult to maintain a particular update frequency using a sequential version of their software Programmable Logic Controller (PLC). In the two-processor version of this code, one processor repeatedly reads all input values and then calculates new outputs, while the other processor repeatedly logs all input and output values and then writes all output values. This discussion helped them understand how they could use multiprocessing to improve update frequencies even though their algorithm itself was inherently sequential.

The two-processor pipeline was suggested based on an assumption that the same I/O bus is fully utilized by the input and output activities. If inputs are channeled through a different bus than outputs, or if a common I/O bus has capacity to handle all inputs and outputs within a 5 ms time slice, then the same algorithm can run with a four-processor pipeline, yielding a 5 ms update frequency, as shown in Figure 4.

Remember that a goal of parallel restructuring is to allow each thread to execute independently, with minimal coordination between threads. Note that the independent threads are required to communicate intermediate computational results (the 1,000 input values and 1,000 output values) between the pipeline stages. A naive design might pass each value as an independent Java object, requiring thousands of synchronization activities between each pipeline stage. A much more efficient design would save all of the entries within a large array and synchronize only once at the end of each stage to pass this array of values to the next stage.

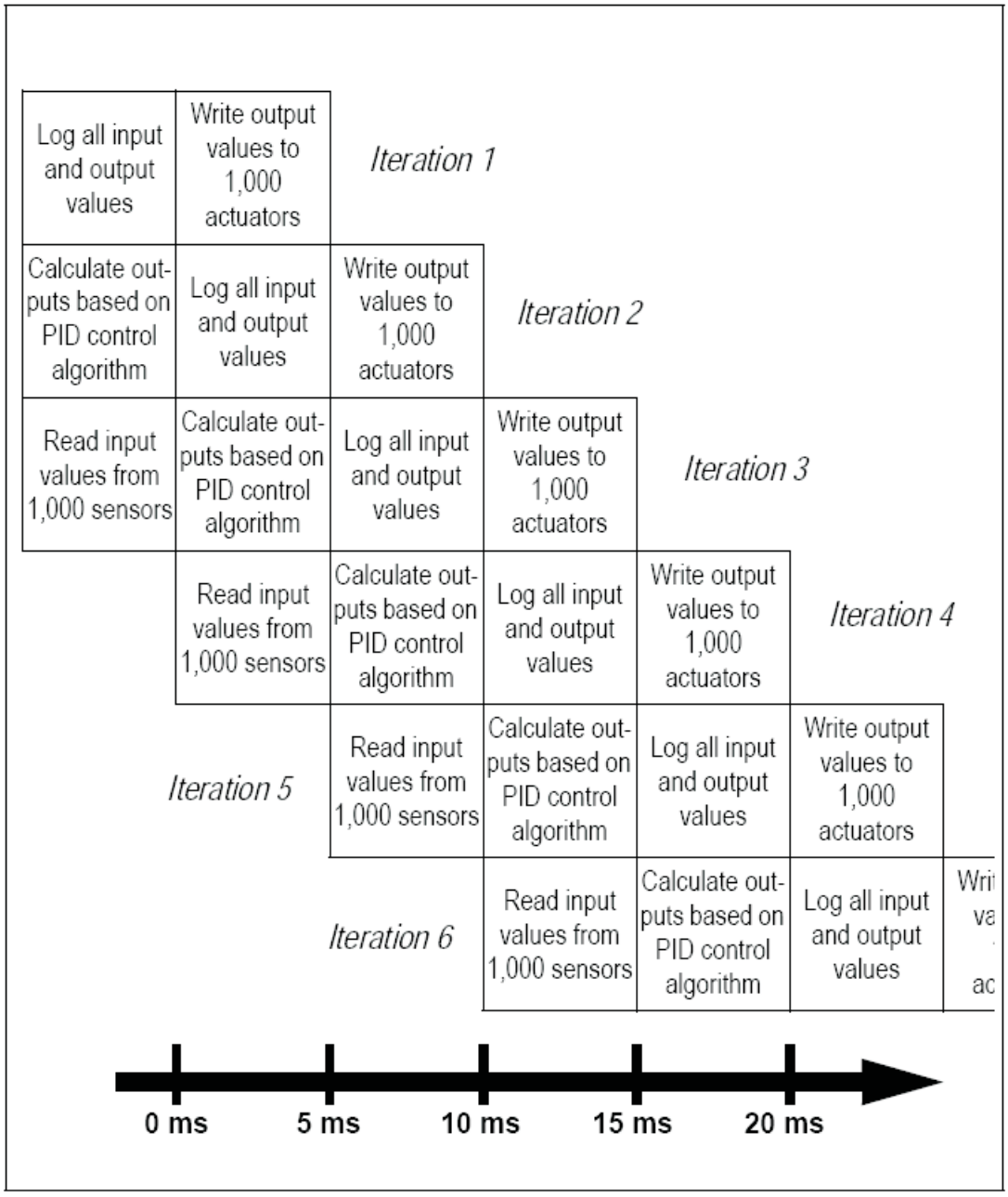

Figure 4. Four Processor Pipelined Implementation of Soft PLC

IMPLEMENTING PARALLEL ACTIVITIES

Java's built-in support for concurrent programming makes implementation of parallel code straightforward. However, there are various issues which require appropriate attention at architecture and design time in order to assure robust and scalable operation of the modernized application. Some of these topics are discussed further below.

Thread pools. Many real-time workloads are much more dynamic than the rigidly scheduled software PLC application described above. Events are signaled asynchronously and a prescribed response must be provided for each event within a specific deadline associated with that event.

1. Because starting and stopping threads are relatively costly operations, it is preferable to start all threads during application boot time. Each thread iterates, waiting for an appropriate event to occur, providing a response to that event, and repeating the sequence.

2. In some cases, certain threads may be dedicated as servers, capable of responding to a variety of different events, depending on the application's most urgent requirements. A typical server thread implementation might be represented by the the following run() method:

```
public void run() {
   WorkAssignment a_task;

   while (true) {
      synchronized (this) {
            while (my_work_assignment == null) {
               wait();
               }
               a_task = my_work_assignment;
               my_work_assignment = null;
               // Since the server thread is running, the only possible waiters are other
               // threads wanting to assign work to me.
               notify();
            }
            // do the work after releasing the lock, allowing other threads to offer a new
            // work assignment while I'm completing the previous assignment.
            a_task.doWork();
      }
   }
```

This code presumes existence of a WorkAssignment interface with a doWork() method. Elsewhere within this server thread's class definition, a different method would enable other threads to make work assignments to this server thread. That code might resemble the following:

```
public synchronized void assignWork(WorkAssignment wa) {
   while (my_work_assignment != null) {
   wait();
   }
   my_work_assignment = wa;
   // I have just assigned work. It may be that the condition queue holds other threads
```

```
        // wanting to assign work in addition to the server thread waiting to receive work. I'll
        // wake them all just to be sure that the server gets a notification. This is reasonably
        // efficient under the assumption that this system is balanced so that the server is
        // generally keeping up with all requests so there would not normally be other threads
        // waiting to assign work which would be unproductively notified. Note that any thread
        // that is unproductively notified will simply place itself back on the wait queue.
        notifyAll();
    }
```

In order to effectively manage response-time constraints, the recommendation is that threads with shorter deadlines are assigned higher priority. This is known as "rate-monotonic assignment of priorities". Analysis of schedulability to satisfy real-time constraints is discussed below.

Note that if a server thread's priority is assigned in rate-monotonic order, then it is necessary for all of the work tasks assigned to a given thread to have the same relative deadline. Note also that exploiting the full capacity of an N-processor computer may require N running instances of each server thread.

Deadlock prevention. Deadlock occurs if one thread is waiting for another thread to accomplish a critical task while that other thread is unable to accomplish the critical task because it is waiting (possibly indirectly) for the first thread to perform some other critical activity. It is the software equivalent of two overly polite individuals stepping up to a door with the first person saying "You first," and the second responding, "No, after you." People are able to break this deadlock by observing other visual queues.

Unfortunately, software tends to focus on one objective at a time, and is totally oblivious to the bigger picture. Thus, two software threads encountering a similar situation might both wait forever for the conflict to be resolved. A symptom of this behavior is that the computer system becomes catatonic, and eventually, some human operator decides to reboot it.

One very real risk when moving code from a uniprocessor to multi-processor environment is that deadlocks never seen in the uniprocessor code begin to manifest in the multi-processor version. This is because certain coordination algorithms that work fine on a single processor do not work reliably in the multi-processor context. An example of one such algorithm is a non-blocking priority ceiling emulation (PCE) algorithm for mutual exclusion. This protocol specifies that when a thread enters a particular critical section of code, it immediately raises its priority to the ceiling priority associated with that code. By design, any thread requiring mutually exclusive access to this same critical region of code must have a priority no greater than the ceiling priority. On a uniprocessor system, no semaphores or queues are required to implement the PCE

protocol as long as threads holding a priority ceiling emulation "lock" do not suspend execution while holding the lock. This is because a thread holding the lock is known to run at a higher priority than all other threads that may attempt to acquire the lock.

The PCE mechanism also works on a multi-processor but a lock and queue may be required in the implementation because there is no guarantee on a multi-processor that other threads will not be running in parallel at the same or even lower priorities than this thread. Even though PCE generalizes to a multi-processor environment, there is a risk that using this protocol on a multi-processor will result in deadlock even though the same code works reliably on a uniprocessor without deadlock.

Deadlock may occur if one thread uses PCE to lock resource A and a different thread uses PCE to lock resource B. If the first thread subsequently attempts to lock resource B using PCE while still holding its PCE lock on resource A and the second thread attempts to lock resource A using PCE while still holding its PCE lock on resource B, deadlock results. It would not be possible for this sequence of events to occur on a uniprocessor system so the uniprocessor implementation is deadlock free.

A common algorithm to avoid deadlock is to require that all threads acquire nested locks in the same order. A PCE practice that assures absence of deadlock even on multi-processors is to require that inner-nested PCE locks have a higher ceiling than outernested PCE locks. Adopting this practice for all PCE code will assure that the code is deadlock free both on uniprocessor and multi-processor platforms.

Data structure considerations. Often, mutual exclusion locks associated with shared data structures surface as the bottleneck that prevents effective parallel execution of multiple threads executing on different processors. If only one processor at a time needs access to a complex data structure, then a single lock at the outermost interfaces to this data structure may be an effective way to assure the data structure's integrity. However, if many concurrent threads seek to read and even modify the data structure in parallel, alternative locking mechanisms are necessary. Consider the following common approaches:

1. Reorganize the data structures so that physical proximity correlates with temporal proximity for any given thread. In other words, if a thread's visit to a particular part of the data structure predicts with high probability that another part of the data structure will also be visited in the very near future, then it is desirable that these two parts of the data structure be near to each other within the in-memory data structure.

2. Restructure the locking protocols associated with the data structure.
   a. Rather than a single global lock on the entire data structure, introduce multiple locks, each guarding a particular portion of the data structure. Data that has close physical proximity would be guarded by the same synchronization lock. The objective is to allow different processors to manipulate different parts of the data structure in parallel.
   b. Consider a locking mechanism that allows many concurrent readers but only one writer at a time to lock each portion of the data structure.

3. Consider replicating certain data structures so that particular threads can access local copies of certain data without any synchronization at all. This might require that algorithms be adjusted to tolerate the possibility that localized data structure copies may hold "stale" information.

TO AFFINITY AND BEYOND

When multiple processors work together to satisfy a particular workload, the goal to minimize coordination activities may conflict with the need to assure that each processor has plenty of worthwhile work to perform. Coordination is often required to check the status of each processor's ongoing activities, as the work completed by one processor may affect the work assignments given to others.

Individual processors are most efficient if they can focus their efforts on a small set of tasks which remain resident in that processor's local memory and cache. However, aligning the efforts of each processor with globally important objectives may require frequent redirection of each individual processor's efforts. Trading off between "productivity" of individual processors and satisfaction of global objectives is an intricate balancing act. This section discusses several alternative approaches.

Throughout this paper, we assume real-time operation is an important consideration. Thus, there is an expectation that priorities are honored by the Java virtual machine and underlying operating system. Note that non-real-time virtual machines do not necessarily honor thread priorities and will sometimes schedule a lower priority thread to run even though higher priority threads are ready. Real-time virtual machines are generally configured to schedule in full accordance with thread priorities.

One JVM for each processor. With a four-way multi-processor, you can start up four independent Java virtual machines, each one bound to a different processor. One advantage of this approach is that each JVM sees itself as running on a uniprocessor, and can thus use simpler synchronization protocols. Another advantage is that there is strong isolation of behavior between the code running on the independent processors. Each instance of the JVM has independent memory and CPU budgets, with very little

potential for interference from one JVM to the next.

The key disadvantage of this approach is that it is much more difficult to share information between threads running in distinct JVMs. The typical communication mechanism would use Java Remote Method Invocation (RMI). This is orders of magnitude more costly than synchronized access to shared objects residing in the same process space. This high cost for information sharing makes it especially difficult to balance load because this would typically require copying of large amounts of data and possibly code from one JVM to another.

Single JVM with global scheduling of Java threads. An alternative configuration runs a multi-processor-aware JVM across all (or some subset of) available processors. Individual threads are free to migrate automatically between processors. The JVM uses a global scheduling technique, assuring at all times that the N highest priority threads in the Java application are running on the N available processors.

This approach makes information sharing between threads (and processors) very easy because all objects reside in the same global address space. Load balancing is automatically performed by the global scheduler. The disadvantages of this are that (1) the isolation barriers around independent components are less rigid, making the application vulnerable to CPU or memory starvation from misbehaving components; (2) frequent global synchronizations are required to maintain the data structures required to support global scheduling decisions; and (3) scheduling the N highest priority ready threads in the system requires frequent migration of threads from one processor to the next, resulting in poor cache locality.

Single JVM with threads bound to individual processors. Standard edition Java does not offer an API to set the affinity of particular threads to particular processors. However, some virtual machine's offer special support for this capability. The idea is that certain threads can be bound to particular processors, thereby improving the cache locality and reducing the burden on a global scheduler. Binding threads to particular processors also improves predictability of scheduling, enabling analysis of schedulability to guarantee that threads will satisfy their real-time constraints.

Since all of the threads are running within the same virtual machine, they all share access to the same objects. This makes information sharing and load balancing straightforward. Note that load balancing must be performed by application software logic and does not happen automatically, unlike the globally scheduled approach.

One difficulty with assigning thread affinity to processors is that this complicates the

implementation of priority inheritance. Suppose, for example, that thread α, bound to processor 1 and running at priority 6, holds a lock on resource R. Suppose further that there is a higher priority thread β, also bound to processor 1, running at priority 10. Now suppose that thread γ, running on processor 2 at priority 8, attempts to lock resource R. Since the resource is already locked by a lower priority thread, it will endow its priority to thread α because it currently holds the lock. However, raising thread α to priority 8 does not allow it to run, because it is bound to processor 1 and processor 1 is consumed by the demands of thread β, which is running at priority 10.

If you are going to pursue an approach that binds individual threads running within a single JVM to different processors, it is important to know how the virtual machine deals with priority inheritance and take this into consideration when performing schedulability analysis. One JVM approach is to ignore a thread's specified affinity whenever the thread is running with inherited priority. Another approach is to inherit not only thread priority but also thread affinity. A third approach is for the JVM to strictly honor thread affinity and simply document that priority inheritance does not work reliably when shared synchronization objects are accessed from threads bound to different processors.

Hybrid approaches. Various hybrid approaches are possible, each providing different tradeoffs between strong isolation of resources, ease of information sharing, and support for dynamic load balancing. A uniprocessor JVM may be bound to one processor, while a multi-processor JVM may be bound to certain other processors. Each multi-processor JVM may support a combination of global scheduling for certain threads and localized scheduling for threads bound to particular processors. In this sort of hybrid configuration, an unbound thread would be scheduled on any processor for which the highest priority bound ready thread has lower priority than the unbound thread's priority. Note that allowing unbound threads to freely migrate between processors that are also scheduling bound threads adds complexity to the scheduling analysis performed on each processor. Address this complexity by limiting which processors an unbound thread is allowed to run on, or by assigning lower priorities to unbound threads.

ANALYSIS OF REAL-TIME CONSTRAINTS

An important activity in the development of any real-time system is analysis of compliance with real-time constraints. This activity, often described as schedulability analysis, requires a good understanding of the CPU time consumed by every real-time task, the frequency with which each task is triggered for execution, the maximum amount of time each task might be blocked contending for access to shared resources, and the deadline for completion of the task's work measured from the moment it is

triggered. Many thousands of research papers have reported on varous aspects of real-time schedulability analysis. One of the most pragmatic and common approaches, especially for software components that are independently developed and maintained, is rate monotonic analysis. Other scheduling techniques are able to extract higher utilization from the CPU, but these other techniques often incur higher run-tim overheads, may require greater degrees of integration between the independently developed software components, and most do not deal as gracefully with transient work overloads as does the rate monotonic technique.

In hard real-time systems, which are typically characterized by an expectation that the system is proven to always meet all timing constraints with no tolerance for the slightest deviation from scheduling constraints, programmers are expected to theoretically analyze all of the worst case behaviors of each task and feed these worst-case numbers in as parameters to the schedulability analysis. In soft real-time systems, the expectation is less rigorous. Approximations of task behaviors based on various measurements are accepted as inputs to the schedulability analysis. The expectation is that scheduling analysis based on approximations of task behavior will usually satisfy all deadlines, but may occassionally miss certain deadlines by relatively small amounts of time.

Traditional uniprocessor rate monotonic analysis. Rate monotonic analysis is much simpler, and provides higher guaranteed utilization of available CPU resources, on a uniprocessor. The basic approach of rate monotonic analysis is to assign task priorities in order of decreasing deadlines, so the tasks with the shortest deadlines get the highest priority. In the absence of blocking behaviors, rate monotonic analysis promises that if the total CPU utilization of a workload for which task priorities have been assigned I rate monotonic order is less than 69%, all of the tasks will always satisfy their real-time constraints. Utilization is computed by multiplying every task's worst-case execution time by its maximum execution frequency and summing the results. The analysis is more complex when tasks contend for exclusive access to shared resources. For a ful description, see reference 12.

Rate monotonic analysis for globally scheduled multi-processor. When an N-way multi-processor is configured to always run the N highest priority read threads in parallel on the N available processors, we call this global scheduling. Rate monotonic theory has been generalized to globally scheduled multprocessors [13]. An interesting result is that the globally scheduled multi-processor system needs much more slack than a uniprocessor in order to guarantee compliance with real-time constraints. One form of the result for globally scheduled multi-processors, is given by the following:

Corollary 1. Any periodic task system in which each task's utilization is no more

than one-third and the sum of the utilization of all the tasks is no more than N/3 is successfully scheduled by the Rate Monotonic Scheduling Algorithm upon N unitcapacity processors.

Note that the rate monotonic scheduling guarantee in this context applies only if the total utilization is less than 33.3% of the total capacity of all N processors. This is much lower than the 69% utilization that can be effectively scheduled on a uniprocessor. When applying this approach, it is also important to consider that the frequent migration of tasks from one processor to another is likely to increase the expected execution times of individual tasks compared to repeated execution of the tasks on the same processor because of loss of cache locality benefits.

Rate monotonic analysis for locally scheduled multi-processors. When threads are bound to particular processors, the schedulability analysis for each processor is the same as the uniprocessor version of rate monotonic analysis. The greatest difficulty with this approach is deciding on the most optimal partitioning of threads between available processors as this is a form of the bin packing problem, known to be NP-hard. A 1998 paper describes a simple first-fit-decreasing partitioning heuristic and includes an analysis demonstrating that the use of this heuristic to assign threads to particular processors will result in a system that is rate-monotonic schedulable as long as:

$$U \leq N(\sqrt{2} - 1)$$

where U is the total workload utilization and N is the number of processors [14]. Simplifying the math, this approach guarantees to satisfy real-time constraints as long as the total utilization is no greater than 41% of the multi-processor capacity.

5. Real-Time Multi-core Garbage Collection

As with uniprocessor real-time garbage collection, the key requirements for robust and reliable operation of real-time garbage collection are that it be

- preemptible, allowing higher priority tasks to run even when garbage collection is active;
- incremental, making appropriate forward progress during available increments of CPU time so that it does not need to restart itself whenever it is resumed following preemption;
- accurate, guaranteeing to find and reclaim all garbage in the system;
- defragmenting, relocating live objects to contiguous memory locations and coalescing contiguous free segments into larger free segments to improve reliability and allocation efficiency; and

- paced, to ensure that the garbage collector replenishes the free pool at a rate consistent with the application's appetite for new memory allocation.

The objective of garbage collection pacing is to make sure that garbage collection gets enough increments of CPU time to make sure it consistently replenishes the allocation pool before the available supply of memory has been exhausted. Figure 5 shows a simulated air traffic control workload with real-time garbage collection running under the direction of a real-time pacing agent. This particular application is running in a fairly predictable steady state as characterized by the following observations. First, the slope of the yellow chart, which represents the amount of memory available for allocation, is roughly constant whenever garbage collection is idle. This means the application's allocation rate is approximately constant. Second, the heights of the yellow chart's peaks are roughly identical. This means the amount of live memory retained by the application is roughly constant. In other words, the application is allocating new objects at approximately the same rate it is discarding old objects. Finally, the percentage of CPU time required for application processing is well behaved, ranging from about 20% to 50%.

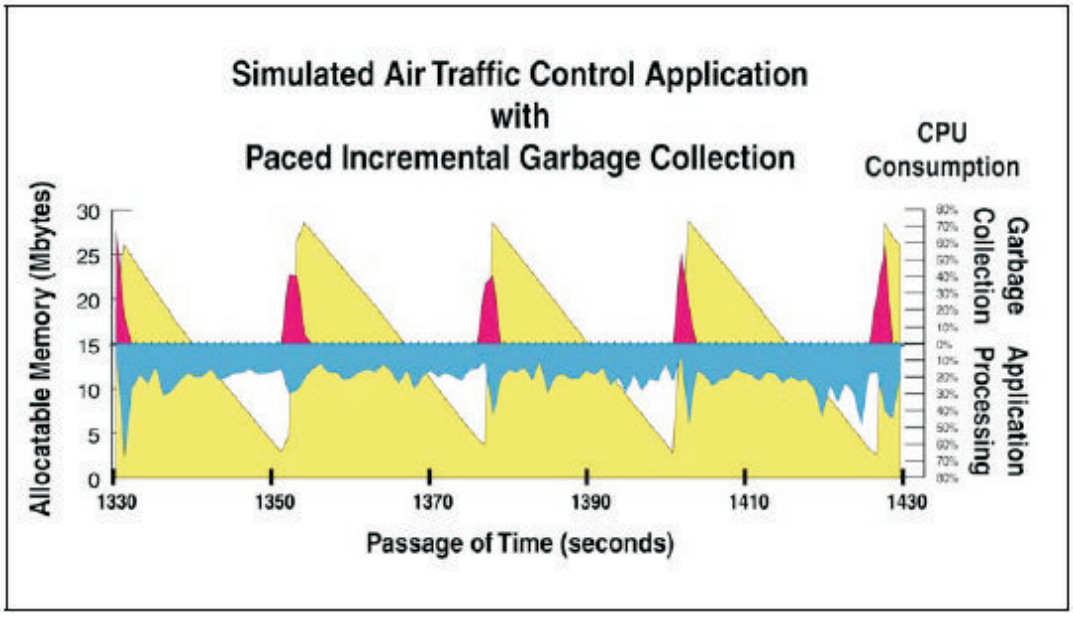

Figure 5. Allocatable Memory versus Time

Note that garbage collection is idle most of the time. As memory becomes more scarce, garbage collection begins to run. When garbage collection runs, it interferes with some,

but not all, of the real-time application processing. For this reason, you'll note a slight dip in application processing each time the garbage collector, represented by the occasional red upward spikes, runs. You'll also note a tendency for application processing to briefly increase following each burst of garbage collection. This is because the pre-empted application needs to perform a small amount of extra work to catch up with real-time scheduling constraints following each preemption. If properly configured, the pacing agent will carefully avoid delaying the application threads by any more than the allowed scheduling jitter.

With multi-processor real-time garbage collection, certain additional issues arise. Consider the following:

1. An N-processor system allows N times as much garbage to be created in a given unit of time. Thus, garbage collection for an N-processor system must perform N times as much work as the uniprocessor version.
   a. An N-way multi-processor can accomplish N times as much garbage collection work as an equivalent uniprocessor in the same amount of time only if the garbage collection algorithms scale well to multi-processor architectures. When migrating from a uniprocessor to multi-processor Java virtual machine, it is important to understand the performance characteristics of the multi-processor garbage collector as its behavior may not scale linearly in the number of processors.
   b. System architects might be tempted to dedicate one processor to garbage collection and the other N-1 processors to application processing. This may work for small numbers of processors but does not scale to large numbers of processors. Suppose, for example, that your uniprocessor application spends 15% of CPU time in garbage collection. On a 4-way multi-processor, 25% of total CPU time would be available to garbage collection, which should be sufficient. However, on an 8-way multi-processor, only 12.5% of total CPU time would be available to garbage collection which is inadequate. Another disadvantage of this approach is that it does not make effective use of the Nth processor during times when garbage collection is idle.
2. Incremental parallel but nonconcurrent real-time garbage collection offers the same benefits on a multi-processor system as on a uniprocessor. During certain time spans, all N processors are dedicated to application code. During other time spans, all N processors are dedicated to small increments of garbage collection.

3. Fully parallel and concurrent garbage collection offers greater scheduling flexibility than parallel but nonconcurrent real-time garbage collection. In particular, certain processors can be running increments of garbage collection in

parallel with execution of application code on other processors. This form of real-time garbage collection is more complex and may impose a larger overhead on the execution of the application threads because those threads must coordinate with finer granularity with garbage collection activities.

6. Example Software Migration from C to Multi-processor Java

Calix is a supplier of simplified services platforms designed to facilitate all aspects of voice, data, and video service delivery to business and residential subscribers for local exchange carriers of all sizes. The company was founded in 1999. By 2007, they had become the number one supplier of DSL broadband loop carriers, accounting for over 41% of DSL/voice ports shipped on broadband loop carriers in North America. In 2001, Calix faced difficult challenges. Their management plane software, which had been implemented in C, lacked key features required by the market and the product had a number of significant but elusive bugs. One of the reasons it had been so difficult to manage the C management plane was because the problem domain needed multiple threads but the C language did not provide support for threading. Calix engineers had emulated multiple threads by using cooperative coroutines, implemented in C. This code was difficult to understand and maintain. Engineering management struggled to deal with these challenges. Eventually, they chose to rewrite the entire management plane as concurrent Java code.

The Java implementation of the Calix C7 multi-service access platform is designed to support asymmetric multiprocessing with one copy of the JVM running multiple concurrent threads on each processor. The benefit of multiprocessing in this application is to allow scalability to large numbers of I/O channels. The uniprocessor configuration supports a combined total of 480 plain old telephone system (POTS) or digital subscriber line (DSL) ports. Up to five Calix C7 processors can be combined to support a total of 2,400 ports. Each interconnection of Calix C7 processors serves the needs of one local exchange.Basic management activities such as provisioning, fault and performance monitoring, and security enforcement require a unified framework capable of providing seamless communication between the management software and the services on the Calix C7 platform. By moving the application framework of its management technology to an object-oriented platform, Calix sped application development and improved software quality. By using Java as the implementation language, Calix simplified and centralized error handling, avoided many memory management issues, and took advantage of a standardized remote debugging interface.

At the start of the migration effort, Calix engineers were not familiar with Java. They

learned Java as part of this engineering activity. Specific metrics provided by Calix regarding the conversion are enumerated below.

1. Two-fold improvement in software developer productivity compared to the previous implementation in C, even counting the time required to learn Java

2. Five-fold reduction in code size compared to the previous implementation in C

3. Fewer bugs in the Java code than in the original C code

4. More flexibility and generality in the new system, as the Java implementation of their management plane was more capable of adding key capabilities required by evolving market requirements

In evaluating their experience with this migration, Calix identified several key factors that eased the migration from C to Java:

1. Portability of Java yielded several distinct benefits
   a. Developers could test and debug most of their code on larger, faster, desktop workstations (running Windows, Solaris, or Linux) even though final employment was on a single-board computer running a real-time operating system
   b. Breadth of off-the-shelf library code, never before tested on their specialized hardware, worked out of the box with no porting effort
   c. Simplified coordination with outside consultants, who guided development but did not have access to any of their specialized hardware
2. Quality development tools
   a. Symbolic cross debugger was essential, with the user interface running on a desktop workstation and the debugger agent running on their single-board computer
   b. Remote performance profiler was essential during system tuning and optimization; profiler feedback was especially valuable as it helped educate Calix engineers new to Java development regarding the costs of particular coding practices
   c. Calix summarized the importance of access to these development tools, stating that the effort would have failed if they did not have “tools to match the capabilities” they had used during C development

Summary

Multi-processor architectures offer superior performance and more efficient and scalable use of electric power. The shift by chip vendors to abandon uniprocessor designs in favor of multi-core chips is forcing changes to the ways that embedded software systems are architected and designed. In light of the challenges associated with implementing and maintaining multi-processor software, the C and C++ languages are showing their age. Since the Java language was designed to simplify engineering of multi-processor software, it is often preferred for the major software rewrites that are required to take full advantage of modern multiprocessing capabilities. The two-fold improvement in developer productivity and the five- to ten-fold improvement in ease of software reuse and integration are other reasons to prefer Java for major software renovation activities.

Software engineers who are responsible for modernization of existing uniprocessor software legacies need to become familiar with various multiprocessing issues in order to effectively manage the modernization efforts. These issues include topics in information sharing and synchronization, parallel algorithms and data structures, resource partitioning and load balancing, real-time schedulability analysis, and choice of programming language.

Faced with the need to migrate existing legacy software to multi-processor platforms, in-house engineering teams may benefit from involvement of external experts who can provide essential training and consulting relating to multi-processor software.

Bibliography

1. Moore, G., “Cramming more components onto integrated circuits”, Electronics, vol. 38, no. 8, April 19, 1965
2. “Design Considerations for Size, Weight, and Power (SWAP) Constrained Radios”, Presented at 2006 Software Defined Radio Technical Conference and Product Exposition, Nov. 14, 2006 (http://www.thalescomminc.com/media/DesignforSWAPRadios-SDRTechConference-Nov2006-s.pdf)
3. Barroso, L. A., “The Price of Performance”, ACM Queue, pp. 48-53, Sept. 2005.
4. Qi, X., Zhu, D., “Power Management for Real-Time Embedded Systems on Block-Partitioned Multi-core Platforms”, Proceedings of the 2008 International Conference on Embedded Software and Systems, pp.110-117, IEEE, 2008
5. “Multicore for Embedded Devices”, TechOnLine, Apr. 30, 2007. (www.techonline.com)
6. Merritt, R., “Parallel Software Plays Catch-Up with Multicore”, EE Times, June 22, 2009.
7. Suess, M., “An Interview with Dr. Jay Hoeflinger about Automatic Parallelization”, Aug. 14, 2007. (http://www.thinkingparallel.com/2007/08/14/an-interview-with-drjay-hoeflinger-about-automatic-parallelization/)
8. Amdahl, G., “Validity of the single-processor approach to achieving large scale computing capabilities,” Proceedings of AFIPS Conference, 1967, pp. 483-485.
9. Krishnaprasad, S., “Uses and Abuses of Amdahl’s Law”, Journal of Computing Sciences in Colleges, Vol. 17, No. 2. Dec. 2001, pp. 288-293.
10. Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., and Lea, D., Java Concurrency in Practice, Addison-Wesley, 2006, 403 pages.
11. IEEE Std 1003.1, 2004 Edition (http://www.opengroup.org/onlinepubs/009695399/)
12. Klein, M., Ralya, T., Pollak, B.,, Obenza, R., Gonzalez Harbour, M., A Practitioner’s Handbook for Real-Time Analysis, Kluwer Academic Publishersr, 1993.
13. Baruah, S. K., Goossens, J., “Rate-Monotonic Scheduling on Uniform Multiprocessors”, IEEE Transactions on Computers, Vol. 52, No. 7, July 2003. pp. 966-970.
14. Oh, D., Baker, T., “Utilization Bounds for N-Processor Rate Monotone Scheduling with Static Processor Assignment,” Real-Time Systems: The International Journal on Time-Critical Computing, vol. 15, pp. 183-192, 1998.

To obtain more information, please contact Atego at **www.atego.com** or call one of our sales offices

North America
Phone: (888) 91-ATEGO
Fax: (858) 824-0212
E-mail: info@atego.com

France
Phone: +33 (0) 1 4146-1999
Fax: +33 (0) 1 4146-1990
E-mail: info@atego.com

United Kingdom
Phone: +44 (0) 1491 415000
Fax: +44 (0) 1491 575033
E-mail: info@atego.com

Germany
Phone: +49 7243 5318-0
Fax: +49 7243 5318-78
E-mail: info@atego.com