# Exploiting Multicore Technology in Software-Defined GNSS Receivers

Todd E. Humphreys, Jahshan A. Bhatti, *The University of Texas at Austin, Austin, TX*
Thomas Pany, *IFEN GmbH, Munich*
Brent M. Ledvina, *Coherent Navigation, San Mateo, CA*
Brady W. O'Hanlon, *Cornell University, Ithaca, NY*

## BIOGRAPHIES

Todd E. Humphreys is an assistant professor in the department of Aerospace Engineering and Engineering Mechanics at the University of Texas at Austin. He received a B.S. and M.S. in Electrical and Computer Engineering from Utah State University and a Ph.D. in Aerospace Engineering from Cornell University. His research interests are in estimation and filtering, GNSS technology, GNSS-based study of the ionosphere and neutral atmosphere, and GNSS security and integrity.

Jahshan A. Bhatti is pursuing a Ph.D. in the Department of Aerospace Engineering and Engineering Mechanics at the University of Texas at Austin, where he also received his B.S.. His research interests are in development of small satellites, software-defined radio applications, and GNSS technologies.

Thomas Pany works for IFEN GmbH as a senior research engineer in the GNSS receiver department. In particular, he is concerned with algorithm development and C/C++/assembler coding. He was for six years assistant professor (C1) at the University FAF Munich and for four years research associate at the Space Research Institute of the Austrian Academy of Science. He research interests include GNSS receivers, GNSS-INS integration, signal processing and GNSS science.

Brent M. Ledvina is Director of New Business and Technology at Coherent Navigation in San Mateo, CA. He received a B.S. in Electrical and Computer Engineering from the University of Wisconsin at Madison and a Ph.D. in Electrical and Computer Engineering from Cornell University. His research interests are in the areas of ionospheric physics, space weather, estimation and filtering, and GNSS technology and applications.

Brady W. O'Hanlon is a graduate student in the School of Electrical and Computer Engineering at Cornell University. He received a B.S. in Electrical and Computer Engineering from Cornell University. His interests are in the areas of ionospheric physics, space weather, and GNSS technology and applications.

## ABSTRACT

Methods are explored for efficiently mapping GNSS signal processing techniques to multicore general-purpose processors. The aim of this work is to exploit the emergence of multicore processors to develop more capable software-defined GNSS receivers. It is shown that conversion of a serial GNSS software receiver to parallel execution on a 4-core processor via minimally-invasive OpenMP directives leads to a more than 3.6x speedup of the steady-state tracking operation. For best results with a shared-memory architecture, the tracking process should be parallelized at channel level. A *post hoc* tracking technique is introduced to improve load balancing when a small number of computationally-intensive signals such as GPS L5 are present. Finally, three GNSS applications enabled by multicore processors are showcased.

## I. INTRODUCTION

Single-CPU processor speeds appear to have reached a wall at approximately 5 GHz. This was not anticipated. As recently as 2002, Intel, the preeminent chip manufacturer, had road maps for future clock speeds of 10 MHz and beyond [1]. As more power was poured into the chips to enable higher clock speeds, however, it was found that the power dissipated into heat before it could be used to sustain high-clock-rate operations [2]. Other performance limitations such as wire delays and DRAM access latency also emerged as clock speeds increased, and more instruction-level parallelism delivered ever-diminishing returns [3].

Interestingly, the current limitation of single-CPU processor speeds has not been the cause, nor the effect, of an abrogation of Moore's law. The number of transistors that can be packed onto a single chip continues its usual doubling every 24 months. The difference now is that, instead of allocating all transistors to a single CPU, chip designers are spreading them among multiple CPUs, or "cores", on a single chip.

The emergence of multicore processors is a boon for software-defined radios in general, and for software-defined GNSS receivers in particular. This is because the data processing required in software radios naturally lends itself to parallelism. Software radio is a special case of what are



**Preprint of the 2009 ION GNSS Conference Savanna, GA, September 22–25, 2009**

known as streaming applications, or applications designed to process a flow of data by performing repeated identical operations within strict latency bounds. Streaming applications are perhaps the most promising targets for performance improvement via multicore processing [4].

The goals of this work are (1) to investigate how to efficiently map GNSS signal processing techniques to the multicore architecture and (2) to explore software GNSS applications that are enabled by multicore processors. Investigating efficient mapping of GNSS signal processing tasks to a multicore platform begins with the following top-level questions, to which this paper offers answers:

1. How invasive will be the changes required to map existing serial software GNSS receiver algorithms to multiple cores?
2. Where should the GNSS signal processing algorithms be partitioned for maximum efficiency?
3. What new GNSS processing techniques will be suggested by multicore platforms?

The general topic of mapping applications to multicore processors has been treated extensively over the past decade (see [4] and references therein). The particular case of mapping software-defined GNSS applications to multicore platforms has been treated at an architectural level in [5]. The current paper treats architectural issues, but also reports on an actual multicore software GNSS receiver implementation and discusses the challenges revealed and adaptations suggested by such an implementation.

The remainder of this paper is divided into seven sections. These are listed here for ease of navigation:
II: Parallel Processing Alternatives
III: Efficient Mapping to the Multicore Architecture
IV: Experimental Testbed
V: Testbed Results
VI: *post hoc* Tracking to Relax the Sequential Processing Constraint
VII: Applications of Multicore Software-Defined Radios
VIII: Conclusions

## II. PARALLEL PROCESSING ALTERNATIVES

While it is true that the emergence of multicore processors is promising for software-defined GNSS receivers, it is also true that there exist viable alternatives to the coarse-grained hardware parallelism of standard multicore processors. Hardware parallelism—that is, the hardware features that support parallel instruction execution—is best thought of as a continuum, with field-programmable gate arrays (FPGAs) on the one end and coarse-grained multicore processors on the other (see Fig. 1).


Fig. 1. Hardware parallelism granularity as a continuum.

### A. Field-Programmable Gate Arrays (FPGAs)

FPGAs, programmable logic devices composed of regular arrays of thousands of basic logic blocks, offer the finest grade of hardware parallelism: gate-level parallelism. Streaming applications can take advantage of the enormous throughput this fine-level parallelism offers. As FPGAs become denser and high-level programming tools mature, FPGAs are becoming an attractive target for full-scale GNSS receiver implementation [6, 7].

### B. Massively Parallel RISC Processors

The newest addition to the hardware alternatives for digital signal processing are massively parallel processors composed of hundreds of reduced instruction set computer (RISC) cores. For example, the PC102 from picoChip (www.picochip.com) is a software-programmable processor array that offers 308 heterogeneous processor cores and 14 co-processors, all running at 160 MHz [8, 9]. As far as the authors are aware, no GNSS receiver has yet been implemented on a massively parallel processor, though such a processors could no doubt support an implementation.

### C. Multicore General-Purpose Processors

Multicore general-purpose processors (GPPs) such as the Intel Core line and the Texas Instruments (TI) TMS320C6474 offer coarse-grained hardware parallelism. The multiple cores in these chips—typically from 2 to 4 cores—are large cores with rich instruction sets like those found in legacy single-core processors. In addition to core-level parallelism, these chips typically offer instruction-level parallelism, with each core supporting simultaneous instructions in one clock cycle. Instruction-level parallelism can be used to great advantage in GNSS receiver implementations. For example, the NavX-NSR 2.0 software GNSS receiver, (discussed in Section VII-A) exploits Intel SSSE3 commands to perform 16 parallel 8-bit multiply-and-accumulate (MAC) operations per core per clock cycle—a remarkable total of 64 parallel MACs on the preferred 4-core platform. Hence, the impressive performance of the NavX-NSR 2.0 is dependent on both instruction-level and core-level parallelism. Likewise, the TI TMS320C6474 offers 3 cores, each of which can support eight 8-bit MACs per cycle.

### D. Performance and Ease-Of-Use Comparison

A comparison of the foregoing three parallel processing hardware alternatives reveals two kinds of gaps: (1) a throughput gap that favors FPGAs over RISC arrays and multicore GPPs, and (2) an ease-of-use gap that favors multicore GPPs over RISC arrays and FPGAs. The throughput gap is evident in Table I, which is based on the benchmarking results given in [8] with results for the single-core 'C6455 extrapolated to the three-core 'C6474. By measure of total channels supported, cost per channel, or power consumption per channel (not shown in Table I), the FPGA far outstrips the other two platforms.

The ease-of-use gap is more difficult to benchmark. FPGAs designs have historically been crafted in hardware description languages such as Verilog or VHDL. While powerful, these languages are less familiar to most engineers and are not as expressive, easily-debugged, or easily-maintained as high-level programming languages such as C/C++. In recent years, FPGA vendors have introduced high-level synthesis tools that allow users to generate designs from block-diagram-type representations or from variants of the C language [10, 11]. But these high-level tools typically use the FPGA resources inefficiently compared to hand-coded Verilog or VHDL, and often are inadequate to express the entire design, requiring engineers to patch together a design from a combination of source representations [7, 11].

In short, under current practices, implementing a digital signal processing application on an FPGA typically takes considerably more effort—perhaps up to five times more—than implementing the same application on a single-core DSP [11]. Thus there exists a wide ease-of-use gap between FPGAs and single-core GPPs.

But the ease-of-use gap narrows as single-core GPPs give way to multicore GPPs. The added complexity in synchronization and communication for applications ported to multicore GPP platforms makes all stages of a design life cycle—from initial layout to debugging to maintenance—more difficult. One of the goals of this paper is to evaluate just how much the ease-of-use gap narrows with the transition to multicore GPP platforms.

One might think that massively parallel RISC arrays such as the picoChip PC102 would fall somewhere between FPGAs and multicore GPPs in regard to ease-of-use. This does not appear to be the case. In fact, it appears that programming RISC arrays has proven so challenging for users that vendors such as picoChip no longer offer general-purpose development tools for their hardware. Instead, users are limited to choosing from among several pre-packaged designs. Hence, in general, RISC array ease-of-use is far worse than that of FPGAs or multicore GPPs.

Because each designer evaluates the trade-off between performance and ease-of-use differently, and differently for each project, the right hardware platform is naturally designer- and application-specific. For leading-edge research into GNSS receiver technology, especially at research institutions where projects are handed off from one student to the next, ease-of-use is weighted heavily over performance. Moreover, given that many exciting GNSS applications are well within the performance capability of high-performance multicore GPPs (as will be shown in later sections of this paper), multicore GPPs remain the authors' platform of choice. However, the trend lines appear clear: with outstanding performance and ever-more-powerful design tools, FPGAs are positioned to become the future platform of choice for software-defined GNSS receivers.

## III. EFFICIENT MAPPING TO THE MULTICORE ARCHITECTURE

The challenge of mapping an application to a multicore architecture is one of preventing the gains from parallel execution from being squandered on communication and synchronization overhead or poor load balancing. These are the basic problems of concurrency.

### A. The Fork/Join Execution Model

A software-defined GNSS receiver, a general block diagram of which is shown in Fig. 2, is an inherently parallel application. The two-dimensional acquisition search can be parallelized along either the code phase or Doppler shift dimension and can be further parallelized across the unique signals to be searched. Once signals are acquired, the tracking channels run substantially independently, and thus are readily parallelizable (one exception to this are vector tracking loop architectures, whose correlation channels are interdependent).

Both parallel acquisition and parallel tracking are punctuated with synchronization events by which all parallel task execution must be completed. For acquisition, the synchronization event is the moment when a decision must be made about whether a signal is present or not. For traditional scalar-type tracking, the synchronization event is the computation of a navigation solution.

The parallel processing from one synchronization event to the next can be represented by the fork/join execution model (Fig. 3). At the fork, a master thread may create a team of parallel threads. Alternatively, if threads exist statically and the memory architecture is distributed, the fork may consist of data being distributed to the separate cores for processing. In any case, the fork marks the beginning of parallel processing of a block of tasks. As defined here, the task assigned to each core within a fork/join block is the sum of all work the core must complete within the

TABLE I

Performance and cost comparison of alternative parallel processing platforms

| Chip | Clock Speed | Chip Cost | Channels Supported | Cost per Channel |
|---|---|---|---|---|
| picoChip PC102 | 160 MHz | \$95 | 14 | \$6.8 |
| TI TMS320C6474 | 1 GHz | \$170 | $\leq 6$ | $\geq$ \$28.3 |
| Xilinx Virtex-4 FX140 | -11 grade | \$1286 | 432 | \$3 |

block, no matter how many separate execution threads are involved. Therefore, a core may service several threads in completing its task within a fork/join block. The outputs of each parallel task are joined at the join event.

The most computationally expensive of the parallel tasks in a fork/join block is called the critical task. To meet real-time deadlines, the critical task must complete within the fork/join block. For maximum efficiency, the critical task should not extend prominently beyond any other parallel task. This objective is termed load balancing.

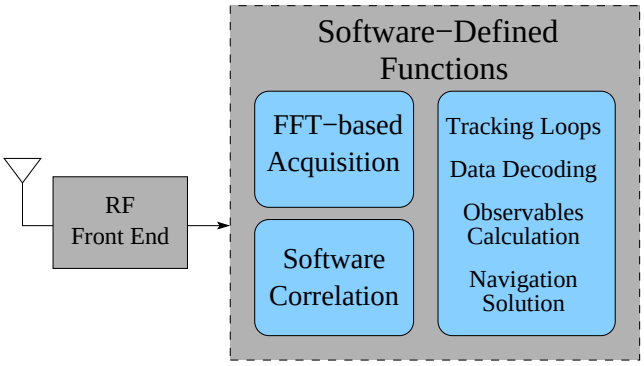


Fig. 2. Block diagram of a general software-defined GNSS receiver.

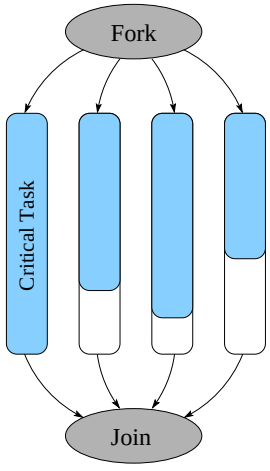


Fig. 3. The fork/join execution model. The duration of each core's task within the fork/join block is marked by blue shading. The most computationally expensive of the parallel tasks is the critical task.

### B. Memory Architecture Considerations

The speed with which each core on a multicore processor can access instructions and read and write data to memory is a crucial determinant of processing efficiency, and must be taken into account when partitioning tasks for parallel execution.

To reduce accesses to off-chip RAM, which may take several tens of clock cycles, processors have been built with a hierarchy of memory caches that temporarily store often-used instructions or data. Before performing an expensive reach into off-chip RAM, a core will first check to see if the same instruction or data are available in cache. If a "cache hit" occurs, the processor saves valuable clock cycles; otherwise, on a "cache miss" the processor must reach into off-chip RAM.

The fastest cache, called level 1 (L1) cache, is also physically closest to the processing core. Read operations from L1 can be executed in a single clock cycle. L1 cache is tied to a particular core. Level-2 (L2) cache is further from the core than L1, and read operations from L2 typically take at least ten clock cycles. L2 cache can in some cases be flexibly allocated within a unified L2 RAM/cache memory module. L2 is often shared between multiple cores, though access times to each core may differ. For example, the 3-core TI 'C6474 divides 3 MB of L2 RAM/cache among the three cores either as an equal division at 1 MB apiece or as 0.5 MB, 1.0 MB, and 1.5 MB. Each core can access its portion of the L2 memory in roughly 14 clock cycles; access to another core's memory—while permitted—takes much longer. Hence, each 'C6474 core has a high affinity for its private section of the L2 RAM/cache.

When partitioning tasks for parallel execution, one objective will be to maximize cache hits. Therefore, there should be a preference for lumping together tasks that employ identical data or instructions.

### C. Process Partitioning

There are several ways one could choose to partition processing for parallel execution. The partition should be chosen to make most efficient use of computational resources. Accordingly, the optimal partition should yield a high computation to inter-core communication and synchronization ratio, while maintaining good load balancing

between cores. Furthermore, the optimal partition must take into account the target memory architecture as described above to avoid wasting computational cycles on memory arbitration or expensive memory fetches.

Three broad types of parallelism are commonly defined: pipeline, task, and data parallelism [4]. Within each of these may exist several granularity options, from coarse to fine.

### C.1 Pipeline Parallelism

Pipeline parallelism is the parallelism of an assembly line: separate cores work in parallel on different stages of the overall task. For a software-defined GNSS receiver application, one core may be tasked with acquisition, another with tracking, and a third with performing the navigation solution and managing inputs and outputs. For a target platform whose L2 cache is not shared among cores (or is formally shared but has private fast-access sections like the 'C6474), pipeline parallelism will result in a high rate of cache hits. Unfortunately, load-balancing the pipeline across multiple cores can be challenging because the difference in computational demand among the pipelined tasks can be large and there may not be enough smaller tasks to fill in the gaps.

### C.2 Task Parallelism

Task parallelism refers to tasks that are independent in the sense that the output of one task never reaches the input of the others. In other words, task parallelism reflect logical parallelism in the underlying algorithm. Task parallelism is often implicit in the `for` loops of serial programs. The acquisition operation of a software-defined GNSS receiver can be thought of as a task-parallel operation, with Doppler search bins distributed across cores. Likewise, the tracking operation of a software GNSS receiver is task parallel. Several alternatives for task parallelism exist, as ordered below from coarse to fine granularity.

*Signal-type:* Signal-type-level task parallelism assigns, for example, tracking for GPS L1 C/A, L2C, L5I+Q, and Galileo E1B+C across four cores. Signal-type parallelism results in a high cache hit rate and low communication and synchronization overhead, but can lead to poor load balancing.

*Channel:* Channel-level task parallelism is a partition at each unique combination of satellite, frequency, and code. Each channel update, which includes correlation operations and updates to the channel's tracking loops, is distributed across cores. With heterogeneous channel types, channel-level parallelism results in a lower cache hit rate than signal-type-level parallelism, but load balancing is typically better than signal-type-level parallelism and communication and synchronization overhead is low.

*Correlation:* Correlation-level task parallelism is a partition at each unique correlation performed, whether in-phase, quadrature, early, prompt, or late. Load balancing is easy with correlation-level parallelism, but communication and synchronization overhead is high.

### C.3 Data Parallelism

Data parallelism refers to "stateless" actors that have no dependency from one execution to the next. For example, a dot product operation between two large vectors can be parsed such that the multiply-and-accumulate operations on separate sections of the vectors are performed in parallel. Similarly, the correlate-and-accumulate operation in a software GNSS receiver can be parsed into separate sections that are treated in parallel. After each section's accumulation is complete, the section-level accumulations are combined into a total. This is an example of fine-grained data parallelism, which suffers from a high communication and synchronization overhead owing to the shortness of the fork/join execution block.

For a typical GNSS receiver implementation, neither channel updates nor correlations can be data-parallelized because the carrier and code tracking loops that are integral to these operations retain state. Hence, current updates and correlations affect subsequent ones. However, a method called *post-hoc* tracking will be introduced later in this paper that substantially data-parallelizes channels at the expense of some loss of precision in the code and carrier observables. The *post-hoc* tracking approach is an example of coarse-grained data parallelism, which benefits from low communication and synchronization overhead.

### C.4 Preferred Partitioning

Preliminary experiments with software GNSS receiver parallelization revealed, not surprisingly, that task parallelism is best for maximizing parallel execution speedup. In particular, Doppler-bin-level task parallelization of the acquisition operation and channel-level task parallelization of the tracking operation were shown to produce the maximum speedup. Accordingly, the remainder of the paper will focus on these parallelization strategies.

## D. Master Thread and Thread Scheduler

In implementation of parallel programs, the multiple parallel tasks that result from a fork are often referred to as worker threads. At a join, a master thread performs serial operations on the products of the previous fork/join block and prepares for the upcoming fork into separate worker threads. For the current software GNSS receiver implementation, each worker thread initially executes a short decision segment that determines which channel the worker thread should process, if any. One can think of these decision segments as a distributed thread scheduler. The excerpt of source code below illustrates the structure of a fork/join block as implemented in the OpenMP framework (to be described subsequently). Each block contains a fork, a decision segment, a process segment, and a join.

```
// The master thread creates parallel worker threads that
// each execute the following code block.
#pragma omp parallel
{ /** FORK **/
 while(true) {
   /** DECISION SEGMENT **/
   #pragma omp critical
   {
     // Only one thread can execute the decision segment
     // at a time, a condition enforced by the "critical"
     // pragma. The decision segment determines which
     // channel each thread should update or whether the
     // thread should exit.
   }
   /** PROCESS SEGMENT **/
 }
} /** JOIN **/
```

### D.1 Objectives

For efficient channel-level parallelism, the thread scheduler attempts to load-balance parallel threads subject to the constraint that each channel's updates must be performed serially (i.e., there must be no simultaneous processing of the same channel).

If the thread scheduler does not balance the load between the threads, some threads may end up idling before the next join operation, leading to inefficient use of CPU resources. In the case of heterogeneous channel types, the thread scheduler's load balancing objective is analogous to playing the popular computer game "Tetris" except that all blocks have identical shape (straight-line) and orientation (upright), though they have variable length, where length represents the time required to perform a channel update. The thread scheduler aims to place the blocks such that the number of complete rows is maximized.

The thread scheduler's constraint can be explained as follows: state retained in each channel's tracking loops implies that a given channel update depends on information resulting from the previous update of that channel. Hence, each channel's updates must proceed serially and thus separate cores cannot be allowed to simultaneously process the same channel. The technique of *post hoc* tracking, introduced later on, improves load balancing by relaxing this serial channel processing constraint.

### D.2 Strategy

The following strategy is employed by the thread scheduler to optimally load balance parallel threads subject to the serial processing constraint. Each channel is assigned a lock mechanism, which remains locked when the channel is being updated, and a counter representing the number of updates remaining in the current fork/join block. When a thread requests a channel from the thread scheduler, the scheduler chooses the unlocked channel with the most updates left.

## E. Simulation

### E.1 Simulator Description

A simulator was developed to test the thread scheduler. The simulator is a C++ application that implements the thread scheduler strategy but uses programmable dummy loads for the process segment instead of GNSS channel processing. The number of dummy loads and the run time of each load can be configured in the simulator to test the thread scheduler in different scenarios. The simulator output contains timing and thread information that can be plotted in MATLAB to visualize how the scheduler arranges the load blocks in time and by threads.

### E.2 Homogeneous Signal Type

The thread scheduler's expected arrangement of homogeneous channels for two scenarios on a four-core platform is illustrate schematically in Fig. 4. Each filled block represents the processing for a single 10-ms L1 C/A accumulation (a single channel update). In each scenario, five channel updates are shown, representing one fork/join block. The scenarios are "worst case" in the sense that the number of vacant processing blocks is maximized. If the number of channels $n$ is greater than the number of cores $N$, then the maximum possible number of vacant blocks is $N-1$ (left panel of Fig. 4). For long fork/join blocks, the impact of these vacant blocks is negligible. If $n < N$, then $N-n$ cores cannot be used at all due to the sequential processing constraint (right panel of Fig. 4). In this case, the sequential processing constraint prevents proper load balancing. Actual simulation results for this scenario are shown in Fig. 5

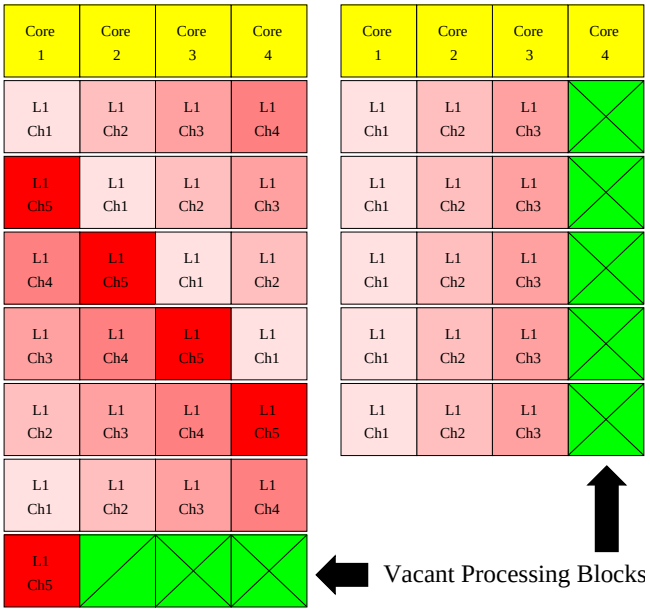


Fig. 4. Expected load balancing results for two different scenarios with homogeneous signal type. Each shade of red corresponds to a different L1 C/A channel. Vacant processing blocks are colored green.

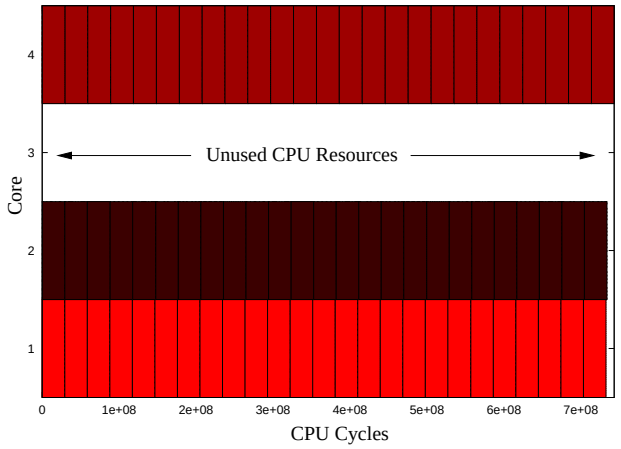


Fig. 5. Execution graph showing poor load balancing when the number of channels is less than the number of cores. Each shade of red corresponds to a different L1 C/A channel.

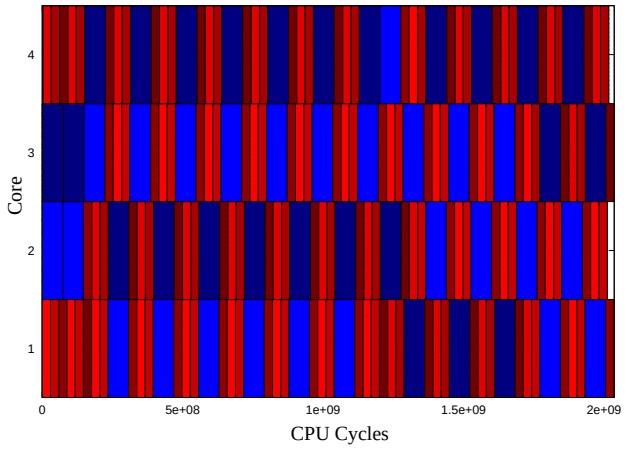


Fig. 7. Execution graph showing good L1 C/A and L2C load balancing. Each shade of red (blue) corresponds to a different L1 C/A (L2C) channel.

### E.3 Heterogeneous Signal Types

The thread scheduler's expected arrangement of heterogeneous channels for seven scenarios with a decreasing number of L1 channels and a fixed number of L2 channels is illustrated schematically in Fig. 6. For this figure, L2 channel updates were assumed to take 2.5 times as long as L1 channel updates. The scenarios are designed to illustrate the loss of throughput efficiency when the thread scheduler must schedule two L2 channels and less than five L1 channels. This loss of efficiency is an extension of the second homogeneous worst-case scenario to heterogeneous signal types. It can be shown that when the number of L1 channels is less than the 2.5 times the number of L2 channels, there is no way to arrange all the channels for maximum efficiency without breaking the sequential processing constraint. The next section will show that a simple formula can express the general conditions required for maximum efficiency. Actual simulation results for best- and worst-case scenarios are shown in Figs. 7 and 8.

## F. Optimum Load Balancing for Channel-Level Task Parallelism

The load-balancing trend evident in the foregoing plots as the number of channels decreases can be generalized. Let the following definitions hold for a software-defined GNSS receiver implemented via channel-level task parallelism on a multicore processor:

- $N$ number of cores
- $m$ number of signal types
- $n_i$ number of signals of type $i$
- $\tau_i$ update run time for signals of type $i$

where the update run times are ordered such that $\tau_1 \leq \tau_2 \leq \cdots \leq \tau_m$. Then for optimum load balancing the following condition must hold

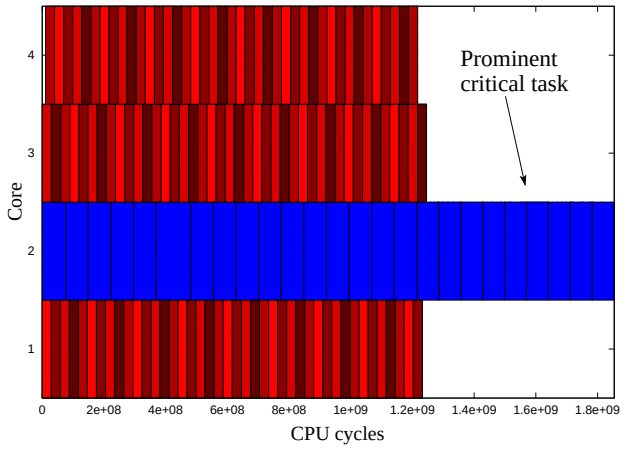


Fig. 8. Execution graph showing poor L1 C/A and L2C load balancing due to a scarcity of channels. Each shade of red corresponds to a different L1 C/A channel. Only one L2C channel, marked in blue, is assumed to be present.

$$\sum_{i=1}^{m} n_i \tau_i \geq N \tau_m \tag{1}$$

This condition can be interpreted as follows. Under the constraint of sequential channel processing, the signal type with the longest update run time $\tau_m$ immediately sets the critical task length to $n_u \tau_m$ for the steady-state tracking fork/join block, where $n_u$, which ranges from 10 to 100, is the number of updates for each channel within the fork/join block (i.e., between calculations of the navigation solution). With the critical task length set, optimum load balancing can only be achieved if there are sufficient channels to occupy all $N$ cores during the fork/join block. This condition is achieved if the total average run time per update $\sum_{i=1}^{m} n_i \tau_i$ meets or exceeds $N \tau_m$.

Fig. 6. Expected load balancing results with heterogeneous signal types, from well-balanced cases (toward left) to poorly-balanced cases (toward right) as a reduction in total channel load leaves a prominent L2C critical task. Each shade of red (blue) corresponds to a different L1 C/A (L2C) channel. Vacant processing blocks are colored green.

## IV. EXPERIMENTAL TESTBED

An experimental testbed was developed to test the predictions of the foregoing simulations on an actual parallel implementation of a software-defined GNSS receiver. Parallelization in the experimental testbed is based on the OpenMP framework [12, 13].

### A. Post-Processing GRID Receiver

The software-defined GNSS receiver that was parallelized is a post-processing version of the dual-frequency (L1 C/A, L2C) GPS receiver described in [14]. This receiver, known as the GRID receiver, is a dual-frequency extension of the receiver described in [15]. The post-processing version shares the code base of the embedded real-time implementation of the GRID, which targets the TI TMS320C6455 DSP, but the post-processing version implements a desktop interface. The software is written entirely in natural language C++, which facilitates code development and maintenance. Most of the parallelization modifications (OpenMP directives) were surgically inserted into the post-processing-specific code, requiring minimal modifications to the GRID code base.

### B. The OpenMP Framework

The OpenMP framework provides C, C++, and Fortran language extensions for shared memory concurrency and is based on the fork/join execution model [13]. The languages extensions consist of compiler directives that control the distribution of tasks over the processor cores and the necessary synchronization of these tasks. In the case of C++, these extensions are `#pragma` statements, which allows portability with compilers that do not support OpenMP, since they will simply ignore the statements. Most modern C++ compilers, such as the Intel C++ Compiler and GCC 4.2, now support OpenMP directives [12].

GCC's implementation of the OpenMP framework consists of a series of code transformations applied on the original source code. The final expansion replaces the directives with function calls to a runtime library libgomp. The runtime library is a wrapper around POSIX threads with various system-specific performance enhancements [13]. Examples of system-specific performance enhancements include using a Linux extension to the POSIX API, `pthread_setaffinity_np`, to enforce POSIX thread affinity to specific cores and the reimplementation of mutex synchronization using atomic CPU instructions and the Linux futex system call rather than using pthread mutex primitives (see the libgomp directory at http://gcc.gnu.org/viewcvs/trunk/libgomp/config/linux).

OMPi and OdinMP/CCp are programs that parse C code with OpenMP directives and generate multithreaded C code using POSIX threads. With this program, compiler support for OpenMP directives is not necessary; all that is needed is system support for POSIX threads. For example, a group of developers testing parallelization of digital signal processing algorithms on the ARM MPCore platform used OMPi with GCC 3.2 to compile their OpenMP code [12]. Hence, the OpenMP directives can be used to parallelize code intended for embedded targets. Figure 9 illustrates the use of OpenMP directives for a basic fork/join block.

### C. Hardware Platform

The experimental testbed hardware platform is specified in Table II.

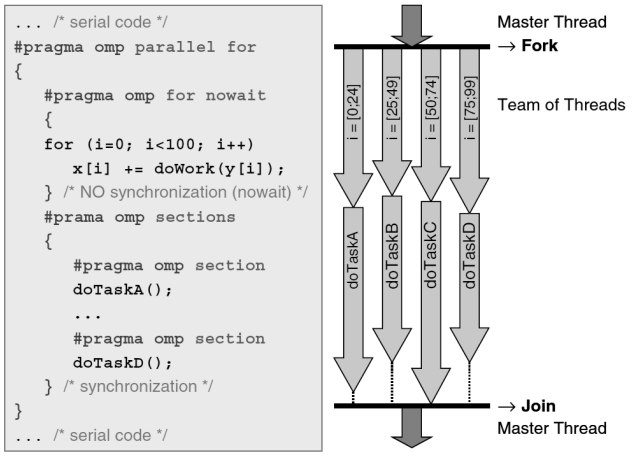


Fig. 9. An example of OpenMP-based execution of a basic fork/join block. (From [12].)

TABLE II
Experimental Testbed Hardware Platform Specification

| | |
|---|---|
| Processor | Intel Xeon E5530 |
| Number of Cores | 4 |
| Clock Frequency | 2399 MHz |
| L3 Cache Size | 8 MB |
| Operating System | openSUSE Linux 11.1 64bit |

## V. TESTBED RESULTS

Testbed results were generated by accumulating the CPU cycles elapsed during the parallelized update segments of the post-processing GRID receiver. The CPU cycles required to load the data from file into memory and compute the navigation solution were not considered in the speedup factors. This makes the speedup factors applicable to the real-time embedded GRID receiver, where the CPU cycles required for data loading and for computation of the navigation solution are minimal.

As shown in Tables III–V, minimally-invasive parallelization via OpenMP delivers near-optimal speedup ($\geq$ 3.6x on 4 cores) for acquisition and track for typical signal ensembles. For atypical signal ensembles, having few L1 or L2 channels, the speedup factor for tracking drops due to poor load balancing across the cores.

TABLE III
Speedup Factor for Heterogeneous Signal Types: Acquisition

| 2 Cores | 3 Cores | 4 Cores |
|---|---|---|
| 1.933 | 2.743 | 3.594 |

TABLE IV
Speedup Factor for Homogeneous Signal Types: Tracking

| Channels | 2 Cores | 3 Cores | 4 Cores |
|---|---|---|---|
| 2L1 | 1.907 | 1.727 | 1.690 |
| 3L1 | 1.932 | 2.715 | 2.549 |
| 4L1 | 1.954 | 2.734 | 3.611 |
| 5L1 | 1.936 | 2.745 | 3.586 |
| 6L1 | 1.956 | 2.773 | 3.614 |
| 7L1 | 1.941 | 2.774 | 3.649 |

TABLE V
Speedup Factor for Heterogeneous Signal Types: Tracking

| Channels | 2 Cores | 3 Cores | 4 Cores |
|---|---|---|---|
| 2L1, 1L2 | 1.934 | 1.917 | 1.887 |
| 3L1, 1L2 | 1.938 | 2.470 | 2.415 |
| 4L1, 1L2 | 1.952 | 2.740 | 2.899 |
| 5L1, 1L2 | 1.955 | 2.759 | 3.639 |
| 6L1, 1L2 | 1.946 | 2.767 | 3.618 |
| 7L1, 1L2 | 1.951 | 2.780 | 3.638 |

## VI. POST-HOC TRACKING TO RELAX THE SEQUENTIAL PROCESSING CONSTRAINT

As described earlier, Eq. (1) must be satisfied for optimum load balancing under the constraint that channels be processed sequentially. Unfortunately, satisfying Eq. (1) is likely to be difficult over the next five years or so. Consider the following factors:

- The number of cores in coarse-grained multicore architectures will increase over the next few years (e.g., from 4 to 8);
- The disparity in update run time $\tau_i$ between signal types will increase over the next few years. For example, the update run time for L5I+Q signals on the real-time embedded GRID receiver will be approximately 30 times that of the L1 C/A channels. This is due to the wider bandwidth of the L5 signal and the need to generate the oversampled ranging codes in real time, as opposed to drawing them from a table [14].
- Until a significant fraction of the GPS constellation is modernized, there will likely be fewer computationally expensive (e.g., L5) channels than cores on a multicore software receiver.

Based on these factors, load balancing will be a challenge for multicore multi-frequency software-defined GNSS receivers over the next few years. This prospect motivates a second look at the sequential processing constraint. The following procedure relaxes the sequential processing constraint, thereby allowing concurrent processing of the same channel, at the expense of a slight increase in the noise of the channel's code and carrier tracking observables.

1. Over a suitable sub-interval of $n_p$ accumulations within a fork/join block, break the carrier- and code-tracking

feedback loops so that code and carrier tracking for each channel are performed open loop based on the last estimated Doppler shift and a model of satellite motion. Call this open-loop Doppler time history the *nominal Doppler trajectory.*

2. Calculate the in-phase $I$ and quadrature $Q$ accumulations for the $n_p$ intervals assuming the nominal Doppler trajectory, yielding $\{I_p(i), Q_p(i), I_{eml}(i), Q_{eml}(i)\}$ for $i = 1, 2, ..., n_p$, where the $p$ and $eml$ subscripts on the $I$s and $Q$s respectively denote prompt and early-minus-late.

3. Perform *post hoc* closed-loop carrier tracking on the prompt accumulations $\{I_p(i), Q_p(i)\}$ for $i = 1, 2, ..., n_p$ by predicting the Doppler shift and carrier phase at the beginning of the $i$th interval based on the carrier tracking loop state at the end of the $(i-1)$th interval. Before measuring the carrier phase at the midpoint of the $i$th interval, rotate the $[I_p(i), Q_p(i)]$ vector by an angle $\bar{\varphi}$ equal to the sum of (1) the average phase error resulting from the difference $\Delta f_i$ between the predicted and nominal carrier phase and Doppler trajectory over the $i$th accumulation and (2) the phase difference at the beginning of the $i$th accumulation interval between the phase implied by the nominal Doppler trajectory and the phase implied by the closed-loop-predicted Doppler time history over the interval $1, 2, ..., i-1$.

4. Apply the closed-loop-estimated Doppler time history $f_i$, $i = 1, 2, ..., n_p$ to estimate the code phase offset $\delta\tau_i$ at the beginning of each accumulation interval.

5. From the differences between each $\delta\tau_i$ and the code phase offset predicted by the measurements $\{I_p(i), Q_p(i), I_{eml}(i), Q_{eml}(i)\}$, estimate a constant code chipping rate offset over the $n_p$ updates due to code-carrier divergence.

6. Accounting for this constant code chipping rate offset, estimate the code phase at the beginning of each of the $n_p$ accumulation intervals.

The *post hoc* tracking algorithm produces Doppler, carrier phase, and code phase estimates for each of the $n_p$ accumulation intervals within the *post hoc* tracking window. The precision of these is slightly degraded compared to those produced by normal closed-loop tracking because (1) the power in the accumulations $\{I_p(i), Q_p(i), I_{eml}(i), Q_{eml}(i)\}$ for $i = 1, 2, ..., n_p$ decreases as the nominal and actual Doppler time histories diverge, and (2) the power in the prompt accumulations $\{I_p(i), Q_p(i)\}$ decreases as the prompt correlator slips off the peak of the autocorrelation function. The number $n_p$ of accumulation intervals in the *post hoc* tracking window must be small enough that the increased noise in the observables remains within acceptable limits. The value of $n_p$ will depend on these limits, on the stability of the receiver clock, and on unmodeled satellite and receiver dynamics. To enable good load balancing, in no case does $n_p$ have to exceed the number of cores $N$.

Performing the correlations and accumulations necessary to calculate $\{I_p(i), Q_p(i), I_{eml}(i), Q_{eml}(i)\}$ for $i = 1, 2, ..., n_p$ in an open-loop manner makes the correlation/accumulation operations independent from one iteration to the next, which is the defining characteristic of data parallelism. Hence, the $n_p$ correlation/accumulation operations for any given channel can be scattered across all available cores. This is an example of how GNSS signal processing can be specially adapted to multicore platforms.

## VII. APPLICATIONS OF MULTICORE SOFTWARE-DEFINED RADIOS

The emergence of multicore technology marks an inflection point for software-defined GNSS radios, allowing them to transition from quaint research platforms to practical science-grade GNSS platforms. The following subsections illustrate the kinds of GNSS applications multicore platforms enable.

### A. The Multicore NavX-NSR 2.0 Receiver

The NavX-NSR v2.0 is a dual-frequency PC-based software-defined GNSS receiver that makes use of an L1/L5 front end connected to the PC via the USB port. The NavPort2 front end also receives IMU input and optionally provides a stable OCXO. The receiver can acquire and track multiple GNSS services and runs on any Intel x86 platform including Intel Atom CPUs. It provides an application programming interface and targets R&D applications. A screenshot of the user interface is shown in Fig. 10.

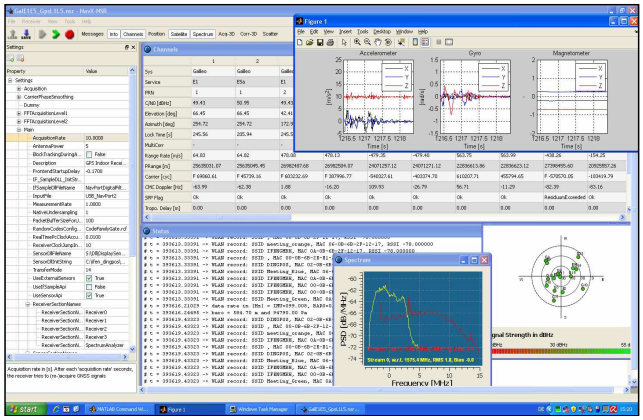

Fig. 10. Graphical user interface of the NavX-NSR 2.0 receiver.

The NSR v2.0 is a successor of the V1.2 version [16, 17] and shares common ideas with the ipexSR of the University FAF Munich [18], but uses different core algorithms and front ends. It performs signal correlation at baseband by representing IF samples as 8-bit integers and exploits Intel's SSSE3 command PMADDUBSW to perform 16 8-bit MACs per clock cycle per core, which makes software

correlation possible even for the wideband GPS L5 and Galileo E5a signals.

The receiver employs two cores for FFT-based acquisition operations and employs all available cores for parallel tracking of GNSS signals. It parallelizes the signal tracking at signal-type-level, which, as discussed in Section III-C.2, is not as efficient as channel-level parallelism, but it nonetheless achieves spectacular throughput: On the hardware platform specified in Table VI, and when connected to a GPS/Galileo simulator, the NSR v2.0.10 tracks 10 GPS and 8 Galileo satellites with an average processor load of around 55%. In recognition of its potential as a science-grade reference receiver, the NSR v2.0 was recently incorporated for test within the International GNSS Service network [see IGS Mail message number 5899].

TABLE VI
NavX-NSR v2.0 Hardware Specification

| | |
|---|---|
| Processor | Intel QX9300 |
| Number of Cores | 4 |
| Clock Frequency | 2526 MHz |
| L2 Cache Size (shared) | 12 MB |
| Operating System | Windows XP |
| Front end | L1@16 MHz, L5@33 MHz |
| GPS channels | 12 × L1 C/A, 12 × L5I+Q |
| Galileo channels | 12 × E1B+C, 12 × E5aI+Q |

### B. Parallel Acquisition of TDMA/FDMA Signals

Aside from its application to code-division-multiple-access-type GNSS signals, parallel processing can be applied to speed acquisition of time-division-multiple-access/frequency-division-multiple-access (TDMA/FDMA) signals. Signals of this type are designed to support thousands of simultaneous communication channels—many more than the number of unique GNSS signals. For example, the GSM (Global System for Mobile communications) standard supports thousands of independent communication channels for each cellular base station. Each channel is allocated a unique frequency access and time-slot pair. It has been shown that TDMA/FDMA signals can be exploited for navigation and timing [U.S. Patent Applications 20080001819 and 20080062039].

Acquisition of TDMA/FDMA-type signals is typically aided by one or more pilot signals and by handshaking between the basestation and the mobile user that enables the mobile user to determine the correct frequency access and time slot. In contrast, when using TDMA/FDMA-type signals as navigation signals, a brute-force search of all frequency accesses and time slots may be required. Parallel execution naturally benefits such a brute-force acquisition.

Figure 11 shows acquisition performance results for simulated TDMA/FDMA acquisition on a 4-core desktop processor. Parallelization was effected in this case by direct handling of POSIX threads. In this example each thread is assigned to perform accumulation calculations for one frequency access over a sequence of frames. Two parameters were varied in the experiment: (1) the number of separate threads (from 1 to 16), and (2) the duty cycle of the TDMA/FDMA signal along the time axis (i.e., the duration of a time slot within a frame).

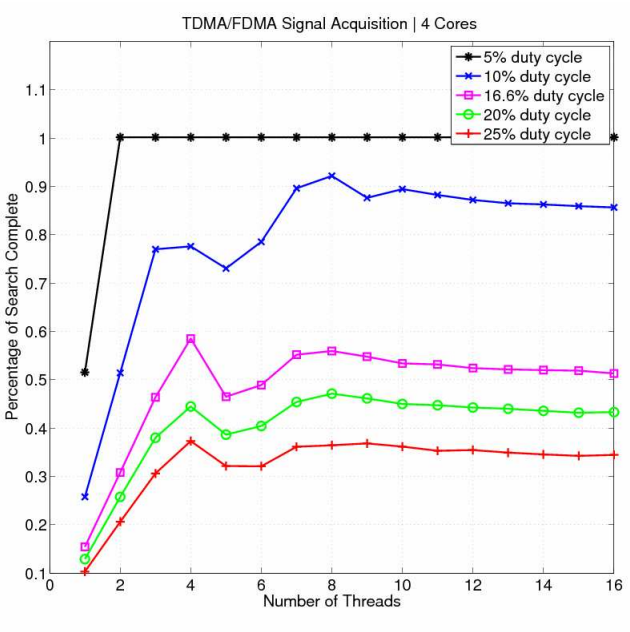


Fig. 11. Simulation results for TDMA/FDMA signal acquisition. The y-axis represents the percentage of the search that could be completed in real time.

Several observations can be made from the results. First, for each of the five accumulation duty cycles, the percentage of the search completed in real time scales linearly when going from one to four threads. This is because each of the four cores can be used independently to search over more of the frequency accesses and time slots. Second, as the accumulation duty cycle increases the percentage of the search completed in real time decreases. This is because a higher accumulation duty cycle corresponds to larger computational load per frame, and thus an overall larger computational load for the entire search. Third, there is, in general, a slight benefit to allocating more than four threads to the search. This is because for lower accumulation duty cycles multi-threading on the same processor is possible even in real time. Fourth, as the number of threads allocated to the search grows beyond eight, the percentage of search completed in real time is reduced due to competition between threads for the hardware cores.

### C. The GPS Assimilator

The final multicore-enabled application considered in this paper is a method for upgrading existing GPS user equipment, without requiring hardware or software modifications to the equipment, to improve the equipment's position, velocity, and time (PVT) accuracy, to increase its PVT robustness in weak-signal or jammed environments, and to protect the equipment from counterfeit GPS signals (GPS spoofing). The method is embodied in a device called the GPS Assimilator that couples to the radio frequency (RF) input of an existing GPS receiver (Fig. 12). The Assimilator extracts navigation and timing information from RF signals in its environment—including non-GNSS signals—and from direct baseband aiding provided, for example, by an inertial navigation system, a frequency reference, or the GNSS user. The Assimilator optimally fuses the collective navigation and timing information to produce a PVT solution which, by virtue of the diverse navigation and timing sources on which it is based, is highly accurate and inherently robust to GNSS signal obstruction and jamming. The Assimilator embeds the PVT solution in a synthesized set of GNSS signals and injects these into the RF input of a GPS receiver for which an accurate and robust PVT solution is desired. The code and carrier phases of the synthesized GNSS signals can be aligned with those of the actual GNSS signals at the input to the target receiver. Such phase alignment implies that the synthesized signals appear exactly as the authentic signals to the protected receiver, which enables a user to "hot plug" the Assimilator into the protected receiver with no interruption in PVT. Besides improving the PVT accuracy and robustness of the attached receiver, the Assimilator also protects the receiver from GPS spoofing by continuously scanning incoming GNSS signals for signs of spoofing, and, to the extent possible, eliminating spoofing effects from the GPS signals it synthesizes.

A prototype version of the Assimilator has been implemented on a single-core TI TMS320C6455 DSP. Although it is the flagship of TI's single-core high-performance line, the 'C6455 can only support assimilation of GPS L1 C/A and L2C signals. To extend the prototype Assimilator to the other input signals shown in Fig. 12, it is currently being ported to the 3-core TI TMS320C6474. Pipeline parallelism will be employed, with correlation, navigation fusion, and RF signal synthesis each performed on a separate core.

## VIII. CONCLUSIONS

This paper's first goal was to investigate how to efficiently map GNSS signal processing techniques to the multicore architecture. Conclusions relevant to this goal are presented below in question-and-answer format with the questions as originally posed in the introduction.

*Q:* How invasive will be the changes required to map existing serial software GNSS receiver algorithms to multiple cores?

*A:* It has been shown that the OpenMP framework can be exploited to convert the post-processing version of a software-defined GNSS receiver, originally written for serial execution, to an efficient parallel implementation by minimally-invasive insertion of compiler directives into the C++ source code.

*Q:* Where should the GNSS signal processing algorithms be partitioned for maximum efficiency?

*A:* If the level-2 cache memory is shared among the multiple cores, with access speed invariant across the cores, then Doppler-bin-level task parallelization of the acquisition operation and channel-level task parallelization of the tracking operation produce the maximum speedup. If the level-2 cache is distributed among the cores so that a memory accesses to another core's level-2 cache is much slower than to a core's own level-2 cache, then signal-type-level task parallelization of the tracking operation may on balance be better due to a high cache hit rate.

*Q:* What new GNSS processing techniques will be suggested by multicore platforms?

*A:* The challenge of load balancing across multiple cores subject to the constraint that channel updates be processed serially has motivated a technique called *post hoc* tracking, which relaxes the serial processing constraint by performing open-loop correlation and accumulation and after-the-fact (*post hoc*) tracking.

This paper's second goal was to explore software GNSS applications that are enabled by multicore processors. Three applications have been showcased:

1. The NavX-NSR v2.0 dual-frequency PC-based software-defined GNSS receiver, which is capable of simultaneously tracking 12 each of GPS L1 C/A, GPS L5I+Q, Galileo E1B+C, and Galileo E5aI+Q on a 4-core Intel processor;
2. A brute-force technique for acquiring TDMA/FDMA signals such as those used in GSM telephony systems;
3. The GPS Assimilator: a novel device targeted for implementation on a 3-core DSP platform that can be used to upgrade existing GPS equipment, without requiring hardware or software changes to the equipment, to improve the accuracy and robustness of the equipment's position, velocity, and time solution.

### References


[1] Cringley, R. X., "Parallel Universe," Technology Review, Jan. 2009, http://www.technologyreview.com/computing/21806.

[2] Tyma, P., "Multicore Programming," Technology Review, Jan. 2009, http://www.technologyreview.com/computing/21822.

[3] Amarasinghe, S., "Multicore Programming Primer," 2007, http://groups.csail.mit.edu/cag/ps3.

[4] Thies, W., *Language and Compiler Support for Stream Programs*, Ph.D. thesis, Massachusetts Institute of Technology, 2009.

[5] Heikki, H., Jussi, R., Tapani, A., and Jari, N., "Multicore Software-Defined Radio Architecture for GNSS Receiver Signal Processing," *EURASIP Journal on Embedded Systems*, Vol. 2009, 2009.

[6] Mumford, P., Parkinson, K., and Dempster, A., "The Namuru

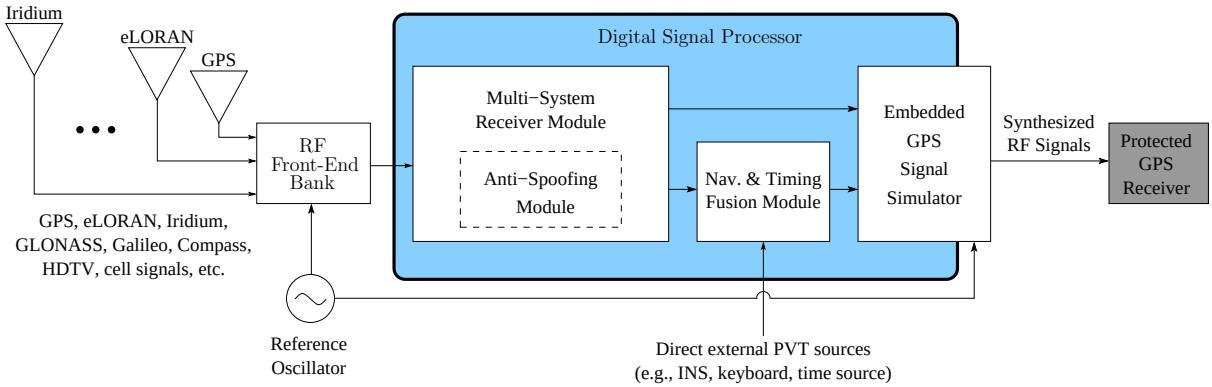


Fig. 12. Functional block diagram of the GPS Assimilator.


Open GNSS Research Receiver," *Proc. ION GNSS 2006*, Institute of Navigation, Fort Worth,Texas, Sept. 2006.

[7] Cobb, S., "FPGA-optimized GNSS receiver implementation techniques," *Proc. ION GNSS 2008*, Institute of Navigation, Savannah, Georgia, Sept. 2008.

[8] anon., "BDTI Releases Benchmark Results for Massively Parallel picoChip PC102," Sept. 2007, www.insidedsp.com.

[9] anon., "PC102 Product Brief," 2004, www.picochip.com.

[10] Williston, K., "FPGAs look for a lingua franca," Sept. 2007, www.dspdesignline.com.

[11] anon., "FPGAs vs. DSPs: A look at the unanswered questions," Jan. 2007, www.dspdesignline.com.

[12] Blume, H., Livonius, J., Rotenberg, L., Noll, T., Bothe, H., and Brakensiek, J., "OpenMP-based parallelization on an MPCore multiprocessor platform–A performance and power analysis," *Journal of Systems Architecture*, Vol. 54, No. 11, 2008, pp. 1019–1029.

[13] Novillo, D., "OpenMP implementation in GCC," GCC Developers Summit, June 2006, http://www.airs.com/dnovillo/Papers/gcc2006-slides.pdf.

[14] O'Hanlon, B. W., Psiaki, M. L., Kintner, Jr., P. M., and Humphreys, T. E., "Development and Field Testing of a DSP-Based Dual-Frequency Software GPS Receiver," *Proceedings of ION GNSS 2009*, Institute of Navigation, Savannah, GA, 2009.

[15] Humphreys, T. E., Ledvina, B. M., Psiaki, M. L., and Kintner, Jr., P. M., "GNSS Receiver Implementation on a DSP: Status, Challenges, and Prospects," *Proceedings of ION GNSS 2006*, Institute of Navigation, Fort Worth, TX, 2006.

[16] anon., "NavX-NSR GPS/Galileo navigation software receiver brochure," 2007, www.ifen.com.

[17] Heinrichs, G., Restle, M., Dreischer, C., and Pany, T., "NavX-NSR–A Novel Galileo/GPS Navigation Software Receiver," *Proc. ION GNSS 2007*, Institute of Navigation, Fort Worth,Texas, Sept. 2007.

[18] Anghileri, M., Pany, T., Guixens, D. S., Won, J.-H., Ayaz, A. S., Stober, C., Kramer, I., Dotterbock, D., Hein, G. W., and Eissfeller, B., "Performance Evaluation of a Multi-frequency GPS/Galileo/SBAS Software Receiver," *Proceedings of ION GNSS 2009*, Institute of Navigation, Savannah, GA, 2007.