SOFTWARE ENVIRONMENT FOR A MULTIPROCESSOR DSP

Asaawaree Kalavade
Networked Multimedia Research Dept.
Bell Labs, Lucent Technologies
Murray Hill, NJ 07974
kalavade@research.bell-labs.com

Joe Othmer, Bryan Ackland, K. J. Singh
DSP and VLSI Systems Research, Dept.
Bell Labs, Lucent Technologies
Holmdel, NJ 07733
{othmer,bda,kj}@lucent.com

1. ABSTRACT

In this paper, we describe the software environment for Daytona, a single-chip, bus-based, shared-memory, multiprocessor DSP. The software environment is designed around a layered architecture. Tools at the lower layer are designed to deliver maximum performance and include a compiler, debugger, simulator, and profiler. Tools at the higher layer focus on improving the programmability of the system and include a run-time kernel and parallelizing tools. The run-time kernel includes a low-overhead, preemptive, dynamic scheduler with multiprocessor support that guarantees real-time performance to admitted tasks.

1.1 Keywords

Multiprocessor DSP, media processor, software environment, run-time kernel, RTOS

2. INTRODUCTION

In the past few years we have seen an increasing demand on the performance offered by digital signal processors (DSP's), due to the surge in complex multimedia and communications applications. To support the computation demands posed by these applications, we have designed and implemented a DSP called *Daytona* [1]. Daytona addresses the performance challenge by exploiting parallelism at two levels: processor- and instruction-level. Specifically, Daytona employs a bus-based, shared memory, multiprocessor architecture, where each processor itself is augmented with a SIMD accelerator.

Daytona is a reasonably powerful architecture — a four-processor chip, running at 100 MHz is capable of delivering 9.6 GOPS of peak performance in a 8-bit mode and 4.8 GOPS in a 16-bit mode. Harnessing the large compute power offered by such a chip is a challenge. We believe that the key to exploiting this power lies in the software environment. The software environment should allow applications to be developed at different levels of granularity without compromising performance. It should also isolate the application programmer from the nuts and bolts of the hardware by capturing architecture-specific details within the tools.

We have designed and implemented a software environment for Daytona that attempts to address these challenges. Our approach is to adopt a "layered" software architecture. At the lower layer, it provides tools for developing and debugging application modules. Application *modules* form the core routines, or kernels, of DSP applications and are primarily hand-coded to achieve maximum performance. The tools at the lower layer focus on easing the job of the DSP programmer. These include a simulator, debugger, compiler, and profiler. At the higher layer, the software environment provides tools to put together these modules to generate complete applications. The emphasis here is on improving the programmability of the system. The components of this layer include a run-time kernel as well as parallelizing tools. The run-time kernel is designed to manage multiple concurrent applications and comprises a low-overhead, prioritized, preemptive, run-time scheduler with multiprocessor support. The scheduler does admission control, dynamically maps tasks to processors, and guarantees real-time performance to admitted tasks.

This paper describes the details of the software environment and is organized as follows. The hardware architecture is summarized in Section 2.1. The design of the software environment is discussed in Section 3. The tools at the lower layer are described in Section 4. The run-time kernel is described in some detail in Section 5.

2.1 Daytona architecture

Daytona is a shared-memory, bus-based, multiprocessor architecture. Figure 1 shows a block diagram of the architecture. Multiple processing elements (PE's) are connected via a shared bus. While different flavors of PE's can be supported, we focus our attention on a particular PE, called *Firebird*. Each Firebird PE consists of a SPARC core, a SIMD (single instruction multiple data) vector coprocessor (*VC*), and a reconfigurable local memory. The SPARC core executes SPARC V8 integer operations. The VC has a 64 bit datapath that can perform vector operations on eight 8-bit components and four 16-bit components; a subset of operations is also supported on two 32-bit components. The result can be either a scalar or in one of the vector formats. One SPARC and one VC instruction can be issued simultaneously. Each PE also contains 8 KB of a reconfigurable local memory. The local memory can be used as any combination of instruction cache, data cache, and a user-managed buffer. The actual partitioning into the three types of

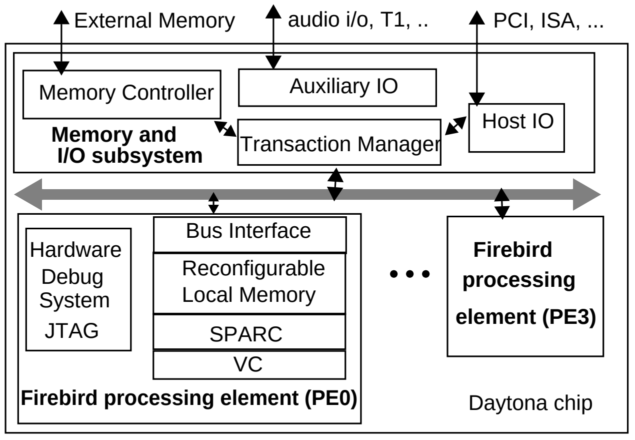

Figure 1. Daytona architecture

memory can be done dynamically through software control. The architecture incorporates a 128-bit wide on-chip split transaction bus (ST bus). The ST bus implements cache coherency through snooping, is pipelined, and supports split transactions to maximize throughput. The PE's can also transfer data to/from the shared memory via DMA (direct memory access). A transaction manager handles the I/O and memory requests. A JTAG-based hardware debug logic has been added to each processor for debugging. The rest of this document focuses on the software environment for Daytona.

3. SOFTWARE ENVIRONMENT

There are several challenges in developing the software environment for Daytona. These are specifically attributed to the need to run multiple, real-time, high-performance applications on one or more PE's. Let us consider the challenges posed by these factors in more detail.

Performance vs. Programmability: Programming a DSP typically involves trade-off between performance and user programmability. To achieve high performance, the trend in the DSP community is to hand-craft applications. However, dynamically changing application sets, the need for flexibility and upgradeability, and rapidly shrinking time-to-market intervals call for the use of sophisticated high-level tools. To achieve an efficient trade-off, our approach is to allow application programs to be developed at two layers. We call this a "layered" software architecture (Figure 2)

Slim real-time support for multiple dynamic applications: Daytona is expected to support multiple simultaneous applications. Additionally, these applications may have to be dynamically invoked. For instance, consider a modem pool where multiple modem applications are dynamically run and stopped at user request. Another example is a settop box application where audio, video, and graphics applications run simultaneously. Different combinations of these applications need to be run, depending on the user activity. In both these cases, mechanisms are needed that enable these applications to efficiently share resources without affecting their performance. Further, applications often tend to have real-time constraints. To support these requirements, we have designed a dynamic scheduling environment.

The factors listed above have been the driving force behind the architecture of the software environment. The components of the software environment and the application design methodology are summarized in Figure 3. The software design methodology begins with the application writer developing the algorithm with the aid of high-level software design environments (1). Once the algorithm is finalized and the modules of the application are identified and designed, the next step is to develop the code for the modules (2). Module development tools such as a compiler and debugger are used to implement the modules. Once modules are available in the form of a module library, applications are put together. Depending on the class of applications, either a static or a dynamic scheduling environment is used to put together applications. In the dynamic scheduling environment (4), applications are specified as a set of modules and are compiled with the run-time kernel. The kernel sequences these applications and also manages external requests to

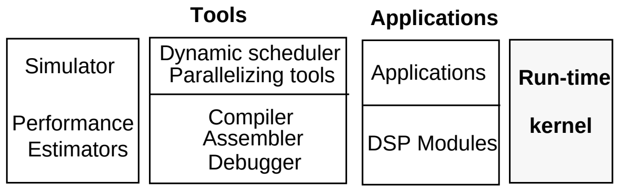

Figure 2. Layered Architecture

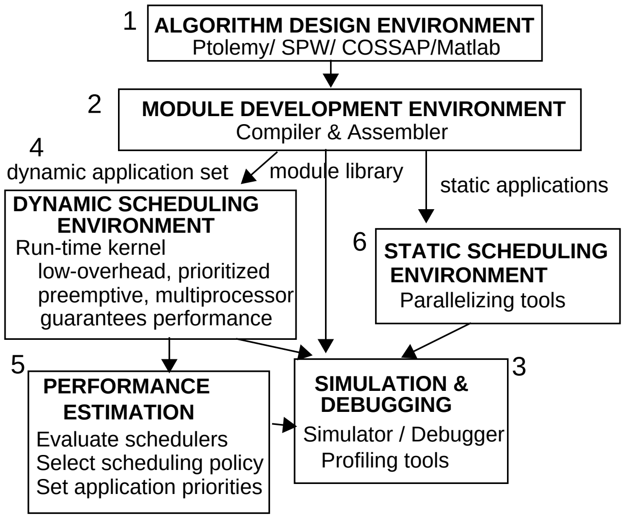

Figure 3. Software Design Methodology and Tools

start and end applications. Performance estimation tools can be used to select the appropriate scheduling policy within the kernel (5). For static application sets, static scheduling tools such as Ptolemy can be used (6).

4. MODULE DEVELOPMENT TOOLS

4.1 PE Compiler

Recall that the PE consists of the SPARC core and the VC, where the VC is a SIMD vector array that operates in parallel with the SPARC. The VC supports several data formatting and alignment modes. The formatting modes, such as rounding, scaling, and saturation are handled through a format register. The alignment modes, also controlled by an alignment register, allow data to be aligned at different boundaries.

While it is desirable to have a complete parallelizing compiler that maps instructions onto the SPARC and VC and extracts parallel instructions for the SIMD coprocessor, designing such an efficient compiler is non-trivial. We have taken an intermediate approach to the compiler. The programmer writes code in a C-like language. The language has been expanded to support VC-specific data types, including 8, 16, and 32 bit vectors and a 64 bit scalar. The compiler parses the source code to identify SPARC and VC statements. It then does a statement-wise translation of the code into assembly code. The compiler analyzes the VC data types and appropriately sets the format and alignment register attributes. Instruction scheduling and code generation are also handled by the compiler. The PE compiler uses the superscalar mode of *gcc* to schedule instructions in parallel to the VC and the SPARC. This required specifying VC-specific dependencies and operation latencies. The PE compiler is fairly efficient. Table 1 compares hand-crafted assembly code to compiled code for a 64-tap convolution routine on 64 data samples. The compiled code is 23%

	code size	**execution time**
hand-crafted assembly for PE	176 Bytes	1113 cycles
compiled code	216 Bytes	1271 cycles

Table 1: Hand-crafted code vs. compiled code

bigger and 14% slower than hand-crafted code. The limitation in this approach is the statement-wise translation. However, this is a reasonable compromise between performance and programmability. While the programmer identifies SIMD parallelism in the application, the tedious tasks of managing the format and alignment registers, instruction scheduling, and code generation are managed by the compiler.

4.2 Simulator and Debugger

Two simulators have been developed for Daytona: a cycle-accurate VHDL simulator and an instruction-level C++ simulator. For a 10PE simulation on a Sparc10, the speeds of the VHDL and C++ simulators are 10 Hz and 10,000 Hz respectively. The C++ simulator is functionally accurate and has a cycle-accuracy within 10% of that of the VHDL simulator. (It does not capture some of the details of the memory latencies.) The C++ simulator is typically used for all application development; the VHDL simulation is used for final performance analysis.

A Tcl/Tk-based GUI has been developed for the C++ simulator. This provides basic debugging support for the simulator. Features supported include: disassemble code, set multiple breakpoints, view/edit SPARC and VC registers on any PE, view/dump memory, view Icache performance, single step, dump simulation trace, etc. Figure 4 shows a screen dump of the current debugging environment.

4.3 Profiler

Several profiling aids are included within the software environment. A call-graph profiling tool *dprof* (similar to gnu gprof) gathers statistics on the number of calls and share of CPU time for each external symbol for each processor. Several *perl* scripts have also been written to analyze the trace output for instrumenting memory access behavior. These profiling tools have been very useful in detecting architecture as well as algorithm bottlenecks.

The Daytona application development environment also contains the standard gnu software development utilities. They are: addr2line, ar, as, c++filt, gasp, ld, nm, objcopy, objdump, ranlib, size, strings, strip, ddis.

The tools provided in the module development environment are functionally comparable to those provided by today's single processor DSP vendors, while also supporting multiprocessor software development.

5. RUN-TIME KERNEL

The run-time kernel is the key part of the dynamic scheduling environment. It is responsible for managing the operation of multiple tasks that share the processors. The requirements of the run-time kernel are summarized as follows: (1) dynamically create/ delete/reactivate tasks (2) map a new task to a processor that can sustain its performance requirements (3) sequence the tasks on each processor such that real-time constraints are met (4) prioritize tasks (5) interrupt (preempt) a low priority task. An important constraint is that the kernel should be compact and should offer minimal overhead. The advantage of such a generic kernel over a customized application-specific kernel (which is frequently used in high-performance DSP applications) is that application writers need not be aware of the interaction of their applications with other applications that may be concurrently running. Once the scheduler is provided with information about the applications (execution times and timing constraints), it provides real-time guarantees to all admitted applications.

We have designed a run-time kernel with multiprocessor support for Daytona that satisfies these requirements. The kernel comprises the scheduler, interrupt handlers, and routines to manage context switches. Before we go into the details, we digress briefly to discuss the task, which is the basic schedulable entity.

Tasks : We are concerned with applications that perform repetitive computations and a deadline constraint is associated with each iteration (e.g. modem transmitter, speech encoder). We define a task as one iteration of the computation associated with an application. A task is characterized by its execution time and a deadline. For example, an audio encoding task involves processing 160 samples per iteration and this has to be done within a deadline of 20ms. The execution time of a task is the total time required for completing the execution of the task. Our current approach is to assume worst-case execution time when deadlines are to be guaranteed. Estimation of execution time of a task can be obtained by profiling each task independently. The deadline (D) of a task is the interval since the task becomes ready, before which the task has to finish execution of the current iteration.

System architecture of the run-time kernel: The system architecture is shown in Figure 5. External interrupts from a "host" are assumed to provide requests to create/delete/re-activate tasks. A *create(delete)* request corresponds to the request to add(remove) a task to the system. A *re-activate* request indicates that data for the next iteration of a task is available. Multiprocessor support is achieved in the scheduler through a two-level scheduling paradigm. Admission control and processor assignment for new tasks is handled through a centralized control scheduler that resides on a control processor (PE0, without loss of generality). Task scheduling on each processor is managed through a separate prioritized task scheduler that runs on each processor. This scheduler is responsible for ensuring that all tasks meet their deadlines.

The operation of the kernel on PE0 is summarized in Figure 6-a. Figure 6-b summarizes the operation of the kernel on other processors. Note that the task scheduler is the same on all PE's.

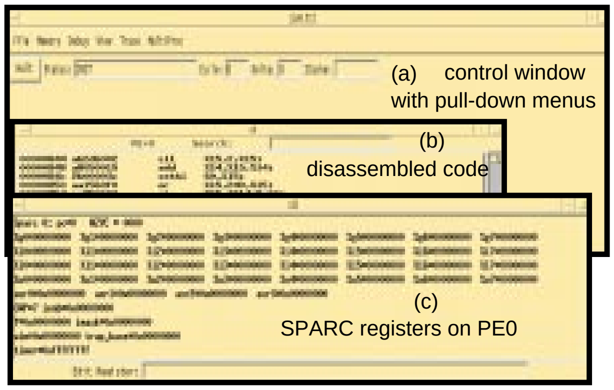
(a) control window with pull-down menus
(b) disassembled code
(c) SPARC registers on PE0

Figure 4. Screen dump of the debugger and simulator.

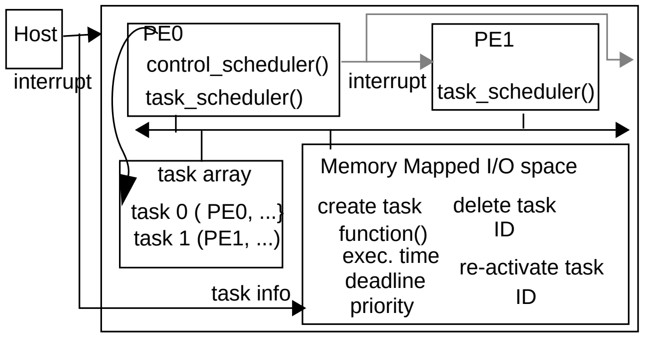

Figure 5. Run-time Kernel: System Architecture

This scheduler is responsible for sequencing the tasks on the corresponding processor such that each task meets its deadline. Recall that other PE's get interrupted only by PE0. On an interrupt from PE0, control is transferred to the ISR. The ISR just reads the create/delete/reactivate information provided by PE0 and returns control to the task scheduler.

The decision to use a centralized scheme for admission control and processor selection was made to simplify handling of the external interrupts. It also makes the task scheduler associated with the other PE's simpler and smaller. The disadvantage of such a two-tiered approach is the increased latency.

Scheduler: The scheduler is the central element of the dynamic scheduling environment. The scheduler is prioritized, preemptive, multitasking, and supports multiprocessor operation. In order to guarantee deadlines, we have implemented an earliest deadline first (EDF) scheduler. In the EDF scheduling algorithm, the scheduler dynamically assigns priorities to tasks according to the deadlines of their *current* requests. A task is assigned the highest priority if the deadline of its current request is the nearest, and is assigned the lowest priority if the deadline of its current request is the farthest. At any instant, the ready task with the highest priority is set to run. The priority of a task changes dynamically, depending on its deadline with respect to deadlines of other tasks in the system. Under the EDF scheduling policy, admitted tasks are guaranteed their deadlines. Admission check is done by solving the inequality $\Sigma(e_i/D_i) \leq 1$, over all tasks i currently in the system plus the new task requested. Here, e_i is the execution time of task i and D_i is the deadline of task i [2]. In other words, a new task is admitted into the system only if its load can be sustained, which is equivalent to checking if the above inequality holds. If tasks are admitted according to this admission criterion, they are guaranteed to satisfy their deadline constraints as long as they operate in a preemptive mode.

Performance: The kernel is reasonably compact: the size of the kernel resident on PE0 is 3144 Bytes while that on other PE's is. 2676 Bytes. The data size required to store the state and status bits for each task is 608 Bytes/task.

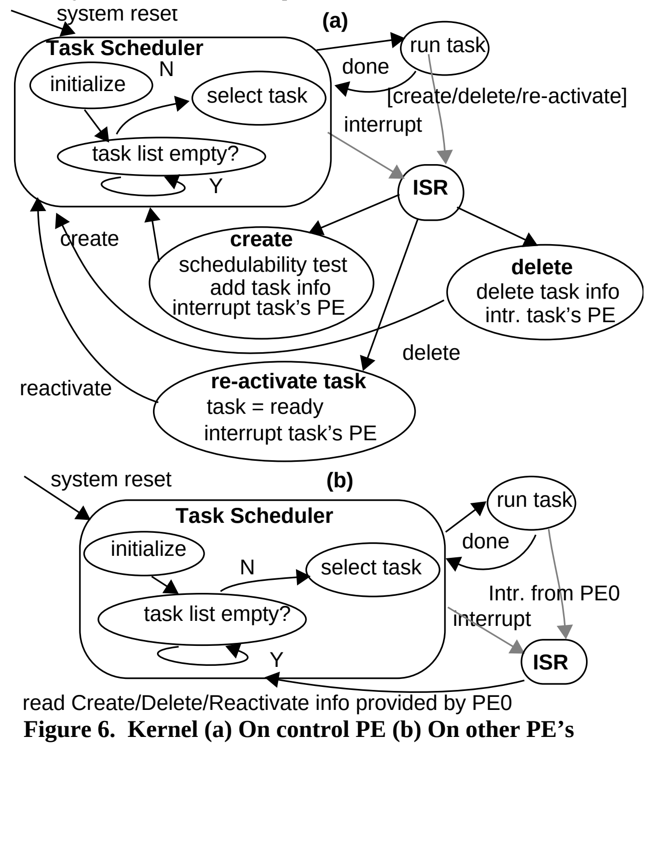

Figure 6. Kernel (a) On control PE (b) On other PE's

The *task switch time* includes the overhead of the scheduler and the context switch. The scheduling overhead depends on the number of tasks in the system as well as the number of tasks on the particular processor. The context switch overhead depends on the number of windows to be saved. For a representative modem pool application with 5 tasks/processor and 2 register windows per modem application, the scheduler overhead is 800 cycles and the interrupt handler takes about 200 cycles. The typical overhead is in the order of 1200 cycles (12 μ s at 100 MHz, 0.12% overhead for a typical 10ms task).

The *interrupt latency*, which is the maximum time interrupts are disabled and represents the maximum time an interrupt may have to wait before it is serviced by the PE, is typically 800 cycles. Finally, the typical latency of passing an interrupt request arriving at PE0 to other PE's is 160 cycles.

Related Work: The features of the run-time scheduler for Daytona are a superset of the features supported by other commercial DSP operating systems [3] such as SPOX (Spectron Microsystems) [4] and Virtuoso (Eonic Systems) [5]. The Daytona kernel also provides multiprocessor scheduling support, does admission control, and provides real-time performance guarantees.

Implementation issues: We have designed techniques into the kernel to minimize context switch overhead by identifying conditions that do not need save/restore.

Limitations: The kernel is not completely preemptable since nested interrupts are not supported. However, since the interrupt latency is reasonably low, this may be acceptable. Finally, due to hardware limitations, the kernel does not support dynamic linking of code, memory management, and security.

6. CONCLUSIONS

There are several challenges in designing the software environment for a multiprocessor DSP. In this paper we described a candidate environment. The key challenges arise due to the need to design high-performance applications rapidly. Our experiences with the tools for developing real applications indicate the following: (a) Module designers spend most of their time minimizing code and data usage and execution time. To simplify this process, good profiling and simulation tools are needed. (b) Dynamic scheduling tools are far more important than static schedulers. A low-overhead run-time kernel that gives real-time guarantees is required to reduce the time to market for sophisticated dynamic application mixes. The software environment described in the paper is currently being used by application designers.

7. ACKNOWLEDGEMENTS

The authors gratefully acknowledge the entire Daytona team at Bell Labs. Specifically, we acknowledge: S. Chandra, B. Denny (TLW Inc.), A. Kulkarni (summer intern 1997), M. Moturi, C. Nicol, J. O'Neill, E. Sackinger, A. Sharma, P. A. Subrahmanyam, J. Sweet, and J. Williams. The authors also acknowledge P. Moghé for his constructive feedback on the paper.

8. REFERENCES

[1] B. Ackland et al. "A Single-Chip 1.6 Billion 16-b MAC/s Multiprocessor DSP", *Proc. CICC'99*, May 1999.

[2] C.L. Liu, J. W. Layland, "Scheduling Algorithms for Multi-programming in a Hard-Real-Time Environment", *Journal of the ACM*, vol. 20, no. 1, Jan, 1993, pp. 46-61.

[3] DSP FAQ: What DSP operating systems are available? *http://www.bdti.com/faq/7.htm*

[4] Spectron Mircrosystems. *http://www.spectron.com*

[5] Eonic Systems. *http://www.eonic.com*