Demand-based Scheduling using NoC Probes

Kurt Shuler
Jonah Probell
Monica Tang

Arteris, Inc.
Sunnyvale, CA, USA
www.arteris.com

ABSTRACT

Contention for shared resources, such as memory in a system-on-chip, is inefficient. It limits performance and requires initiator IPs, such as CPUs, to stay powered up and run at clock speeds higher than they would otherwise. When a system runs multiple tasks simultaneously, contention will vary depending on each task's demand for shared resources. Scheduling in a conventional system does not consider how much each task will demand shared resources.

Proposed is a demand-based method of scheduling tasks that have diverse demand for shared resources in a multi- processor system-on-chip. Tasks with extremes of high and low demand are scheduled to run simultaneously. Where tasks in a system are cyclical, knowledge of period, duty cycle, phase, and magnitude are considered in the scheduling algorithm.

Using demand-based scheduling, the time variation of aggregate demand is reduced, as might be its maximum. The average access latency and resulting idle clock cycles are reduced. This allows initiator IPs to run at lower clock frequencies, finish their tasks sooner, and spend more time in powered down modes.

Use of probes within a network-on-chip to gather demand statistics, application-specific algorithm considerations, operating system thread scheduler integration, and heterogeneous system middleware integration are discussed.

Contents

1. Introduction

Our research, described in this paper, is into the relationship between software and systems-on-chip (SoCs). Our investigation is primarily motivated by a need for better algorithms for scheduling tasks that affect the performance and power consumption of power-sensitive machines such as mobile phones. The greatest consumers of power in such devices are typically the display, radios. We do not address those. Next is DRAM accesses, which benefit from our work only indirectly. Our focus is mainly on power consumed by and within SoCs.

Note that our focus is task scheduling in general, not just operating system thread scheduling. A nod is hereby given to the great body of research into the effect of operating system thread scheduling on L2 cache performance in n-way homogenous multicore processors. While demand-based scheduling is applicable to homogeneous systems, this paper is concerned with the relationship of tasks in heterogeneous systems, including tasks performed by CPUs and tasks performed by other initiator IPs such as GPUs, DSPs, codecs, and wireless interfaces that compete for shared resources such as DDR DRAM memory interfaces. Though demand-based scheduling is widely applicable, the types of tasks considered in this paper are operating system threads run on CPUs as well as cyclical tasks such as video frame coding, video line sourcing, audio sampling, modem data framing, and other high-bandwidth functions performed by SoCs.

It is worth noting the research of the MIT CSAIL group in implementing a framework of Application Heartbeats that could be implemented using the hardware probes discussed in this paper. [1]

In section 2 we discuss methods used to reduce power consumption within the SoC. Section 3 addresses the use of probes as a primary means for monitoring activity within SoCs. Section 4 provides the methods of using the awareness of activity to optimize the scheduling of tasks using demand-based scheduling. Section 5 discusses implementations of demand-based scheduling in software. Section 6 presents a discussion of results and comparison of different demand-based scheduling algorithms to conventional scheduling. Section 7 covers the implementation of demand-based scheduling across processors in heterogeneous systems.

2. Power management within SoCs

The performance of many IPs depends more on the response time of the data bus than on clock speed. This is particularly true for CPUs when running programs that have high cache miss rates. The data bus response time depends on the efficiency of the utilization of shared resources. In many SoCs the critical shared resources are memory interfaces.

Demanding IPs such as CPUs, GPUs, codecs, and modems can run at different frequencies and can be powered off when no information needs to be processed. This is the primary means for scaling power usage in advanced SoCs. Demand-based scheduling addresses ways to use shared resources more efficiently so that demanding IPs can take advantage of power scaling.

The tools to control power usage within chips are well known. PLL source clock dividers can be selected in order to control clock frequencies. Power supplies' voltages can be scaled, in conjunction with reduced clock frequencies. Most importantly, different power supplies can be turned off independently.

Most large IPs each have their own power supply and their own power domain. To safely turn off a power domain it is important to ensure that no read or write transactions are pending to IPs in the domain. Otherwise, the IPs in the other domain will be left in an unknown state, and liable to cause a fatal error. This is handled in modern chips by applying a transaction counter within a disconnect unit at each power domain interface. The counter increments upon each transaction request and decrements on each transaction response that passes across the power domain interface. A system level power manager in hardware is accessible to software. When the power-down of a domain is requested, the system level power manager sends a power disconnect request to the power disconnect unit at the interface between domains. The power-disconnect unit blocks further transactions and waits for responses to pending transactions. When the counter reaches zero (no more transactions are pending across the domain boundary) then the power disconnect unit sends an OK signal to the system level power manager, which then disconnects the power supply.

When to turn off power to a domain and when to turn on power again is a matter of software design.

3. Probes in chips

Demand-based scheduling requires an awareness of the usage of shared resources. This is accomplished by the use of probes built in to SoCs. Probes can be capable of gathering statistics, creating alarm interrupts, and outputting trace information from chips. Probes are used primarily for software debug and secondarily for design-time software performance optimization.

Within SoCs, IPs such as CPUs, GPUs, DSPs, codecs, and modems initiate read and write transactions to target memory interfaces and peripherals arranged such as in Figure 1. In modern SoCs, initiators and targets communicate through a network-on-chip (NoC), which transports transactions through the NoC as packets.

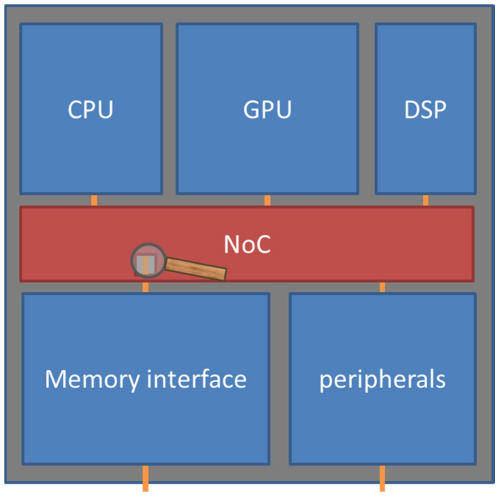


Figure 1: IPs within a SoC

The advent of the prevalence of NoCs within SoCs has made probes available to software in a way that was not possible before. Today, probes are primarily used for system bring-up,

software debug, and software performance optimization. However, running software can make use of probes in real-time just as well.

Probes are located at points within a NoC topology, such as the one shown in Figure 2. In the example, three probes observe traffic near the memory interface target IP. The NoC topology is different for every chip. The NoC topology and the capabilities of each probe within it is documented by chip vendors for programmers to use in order to understand what traffic each probe can observe.

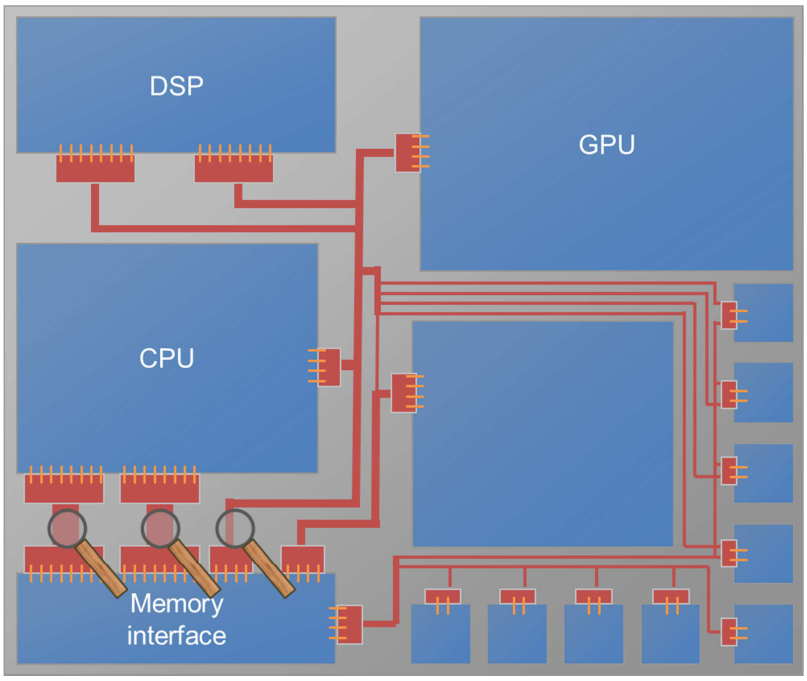


Figure 2: An example SoC topology

Types and uses of probes

There are two kinds of places within a NoC topology that probes can be useful. Packet probes are located within the network. It is common for chips to have a packet probe near each main memory interface. This is useful for performance monitoring since memory interfaces are commonly critical to system performance.

Transaction probes are located at the "edges" of the network where initiators are connected to the NoC. This is useful for measuring the amount of time latency from when an initiator makes a request until it receives a response.

Packet probes

Unlike the performance counters that are a built-in feature of many CPUs, a packet probe within a NoC can measure traffic from all initiators to all targets. Furthermore, if appropriately placed and configured in hardware, packet probes can provide access to the data of different heterogeneous processors using a single software API.

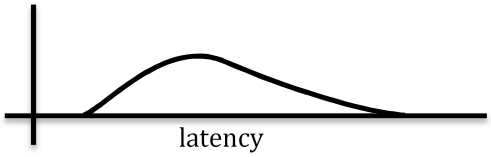
A packet probe observes packets that pass through its location in the NoC topology. Therefore, to implement demand-based scheduling based on data from probes, it is important to know where each probe is in the topology and what type of network packet traffic flows through each

probe. It is common in chips to have one or more probes just before or within each main memory interface IP.

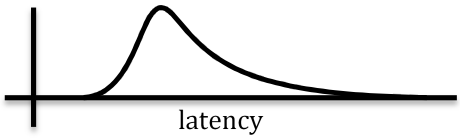
A packet probe contains statistics counters to take measurements of the NoC traffic. One example is to use the packet probe to measure data throughput at the point where it is inserted within the NoC topology. The probe can also filter transactions based on transaction attributes, such that measurements can be made on transactions of interest only.

Arteris Transaction probes™
Transaction probes can measure the latency of transactions (time from request to response). Since the latency of a single transaction alone is not very useful, results are collected as a histogram.

A histogram with a wide spread is an indication of unpredictable response times from a shared target.



A histogram with a tight spread is an indication of a predictable response time. That generally corresponds to a situation in which the target is being used efficiently by the initiators.



Chips with Arteris probes

Chips from the following vendors use an Arteris NoC:

Access Network Tech
Allwiner
Altera
Applied Micro
AXIS Communications
Cavium
CoreLogic
CSR
Freescale
Fujitsu
GCT
Hisilicon
IC-Logic
Ingenic
Leadcore
LG
Megachips
Mobileye
Mtekvision
Nethra
Nokia Siemens Networks
NTT
Nufront
Open Silicon
Pixelworks
Qualcomm
RDA
Renesas
Rockchip
Samsung
SKiPIO
Socle
Spreadtrum
ST
ST Ericsson
TI
Toshiba
Via Telecom

Many have probes, which are typically near memory interfaces. All chips built with Arteris probes are similar in their programming model, but may have different configurations. To use the NoC probes of any particular chip it is important to be familiar with its technical reference manual, such as the one published by Texas Instruments for the OMAP 4430 mobile applications processor. [2] See section 5 regarding portable software implementations.

4. Task scheduling

Operating systems are the best -known task scheduling software. However, tasks other than OS threads are scheduled within a running SoC. Tasks that are cyclical are particularly important. For example, in multimedia applications video and graphics frames are rendered at 60 frames per second intervals. For those and other applications audio samples are generated at 48 kHz.

Types of tasks

Cyclical tasks are schedule-invariant tasks (SITs). They have a heartbeat. Video frame coding, video line processing, audio sampling, and LTE data framing are some examples. Other tasks are schedule-flexible tasks (SFTs). That is, their processing can be delayed somewhat and run as their required resources are available. These include most tasks run on CPUs. Some examples are rendering a GUI, accessing a flash memory, or responding to button presses.

Note that it is important to consider timescale. Processing video macroblocks is schedule-flexible within the limits of a schedule-invariant video frame display period. The amount of lateness of a SFT within a schedule-invariant time budget must be considered in an effective task scheduling algorithm. Delay in processing tasks that are completely schedule-flexible might annoy the user, but will not cause a functional failure.

Transactions with shared resources from SITs are known as real-time traffic. They are sourced by initiator IPs that have inflexible deadlines. Transactions from SFTs are known as best-effort traffic. Best-effort traffic must not delay real-time traffic, due to arbitration, to an extent that would prevent real-time initiators from meeting their deadlines. To this end, packets sent through a NoC are labeled with an urgency level that biases arbitration.

Demand-based scheduling

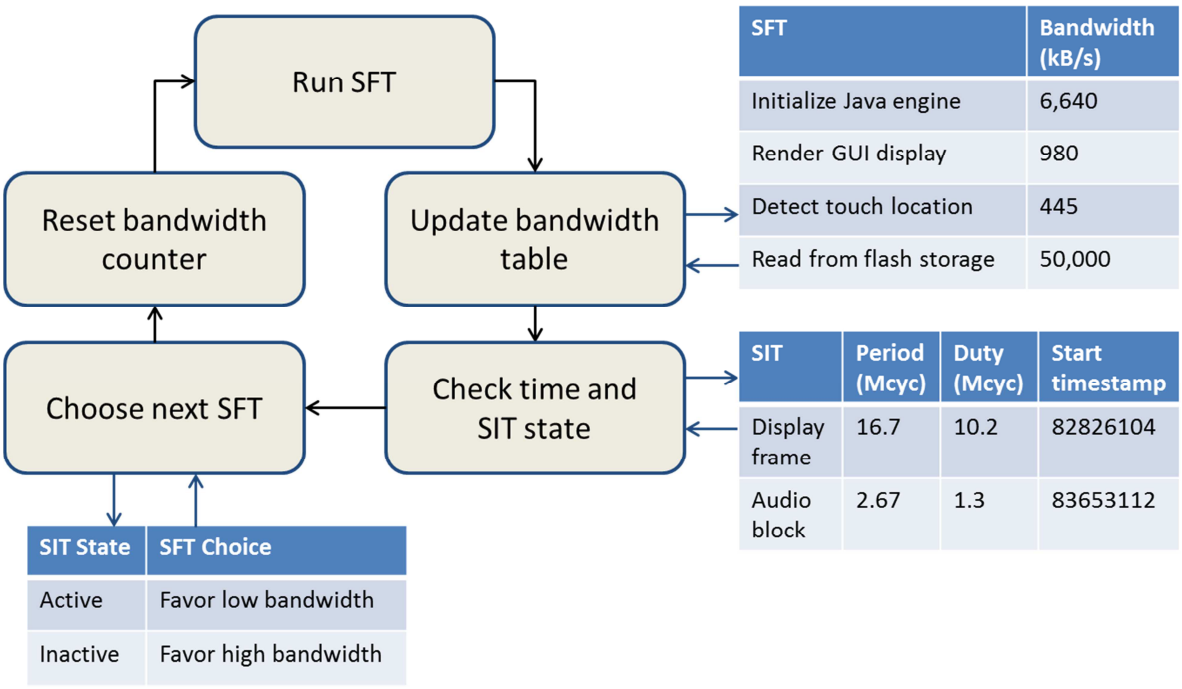
Fairness is a primary concern for operating systems, but less so for demand-based scheduling. Not all tasks are created equal. Demand-based scheduling is useful in systems with some SITs with phases, and some SFTs that compete for a shared resource. An example is a video codec IP core processing frames of video (schedule-invariant) and a CPU generating a user interface overlay (schedule-flexible) that both access a shared memory.

Demand-based scheduling requires doing the following:

- Maintaining a table of SITs, their period, duty cycle, and most recent starting time.
- Maintaining a table of SFTs and the predicted bandwidth consumption during their next time slice.
- Running a SFT while using a probe to measure the bandwidth to the shared resource consumed while it runs.
- Sorting the SFT table by resources consumed.

- At the completion of each time slice of a SFT, checking the phase of the SIT's cycle and choosing the next SFT to run from the low bandwidth end of the sorted table near the beginning of the period and the high bandwidth end of the table nearer the end of the period.

Making that choice is where magical arts are required. A diagram of the basic demand-based scheduling loop with example tables is shown in Figure 3.



SFT	Bandwidth (kB/s)
Initialize Java engine	6,640
Render GUI display	980
Detect touch location	445
Read from flash storage	50,000

SIT	Period (Mcyc)	Duty (Mcyc)	Start timestamp
Display frame	16.7	10.2	82826104
Audio block	2.67	1.3	83653112

SIT State	SFT Choice
Active	Favor low bandwidth
Inactive	Favor high bandwidth

Figure 3: Using bandwidth measurement in demand-based scheduling.

Various variations of the basic algorithm can be used to choose the next-scheduled task, though the choice should be weighted towards tasks in opposite proportion to the likely bandwidth used by the SITs. Highly optimized algorithms are based on the combined utilization due to multiple SITs as well as the utilization of multiple shared resources.

In the degenerate case of a single queued SFT, a choice might be made to postpone the SFT completely until an SIT is inactive.

Per-task and aggregate resource consumption for conventional and demand-based scheduling are shown in section 6 below.

The bandwidth and priority of every SFT for each shared resource and the phase of every SIT's usage of each shared resource are inputs to the demand-based scheduling algorithm. The output of the algorithm is the schedule, which determines which SFT to run on each processing IP, and for what duration of time slice. All of those variables create a multi-dimensional problem with a potentially large matrix. The complexity can be so complex that in large systems it is more efficient to use a dedicated small processor to implement an optimal algorithm.

Determining phase, frequency, and magnitude of SITs

Knowing the frequency, magnitude, and especially the phase of SITs is critical to effective demand-based task scheduling. In some cases it is easy to determine the phase of a SIT, such as for tasks that are initiated by a driver in response to a timer interrupt. In other cases, the phase is not a direct result of an event on the processor that runs the scheduling algorithm. This is true for initiators that run independently.

One approach to determining phase, frequency, and magnitude is to regularly sample the bandwidth used, as measured by a probe, and perform a Fourier transform on the bandwidth data. For this, it is especially beneficial to take advantage of the filtering capability of a packet probe in order to detect just the bandwidth due to a particular initiator of interest. This function could be performed by a custodial microcontroller as described in section 7 below.

Bandwidth prediction

In a basic implementation of demand-based scheduling, the bandwidth consumed during the last time slice of a SFT is used as the prediction of the bandwidth consumption during its next time slice. In fact, the processing that a SFT performs may be affected by the activity of SITs or other SFTs. A more sophisticated prediction of SFT time slice bandwidth consumption can be used that takes into consideration the state of other tasks. Optimally, an adaptive algorithm can be used that measures and attempts to detect bandwidth usage patterns that would indicate task dependencies and thereby enable a more accurate prediction of bandwidth consumption during the next time slice of a SFT.

Latency as feedback on effectiveness

Whereas packet probes are used to gather the statistics needed for scheduling, transaction probes can be used to measure the effectiveness of the scheduling. When competition for the shared resource is high initiators experience more arbitration delay. When competition is low arbitration delay is low. When the scheduling of SFTs is done without an awareness of the phase of SITs, competition will be sometimes high, sometimes low, and generally unpredictable. With demand-based scheduling, since competition is minimized, latencies will have less variation and be lower on average. The amount of variation is indicated by the spread of the latency histogram, with a tight spread being best.

Latency histograms for conventional and demand-based scheduling are shown in section 6 below.

Adaptive algorithms

Using the feedback of histograms from transaction probes it is possible to measure the effectiveness of a scheduling algorithm. For some types of processing, different algorithms might be useful. For example, in a system running an I-frame-only video codec, frame processing duration is pretty similar from one frame to another whereas an I/P/B-frame video codec will have significant variations between frames in both frame processing duration and bandwidth used. A scheduling algorithm that is optimal for the system running an I-frame-only codec might not be optimal for the system running the I/P/B-frame codec.

An intelligent scheduling system might, from time to time, try switching between scheduling algorithms and checking the resulting histogram tightness. An even more intelligent scheduling system will continuously change values, such as variable weights, and track the effect on latency

histogram tightness. By walking through different parameters and different adjustments, the algorithm will gradually move to a maximum optimality.

An ideal adaptive algorithm solves the equation to generate a multi-dimensional field of possible solutions. The field might have multiple maxima. The ideal adaptive algorithm chooses the highest global maximum. As the system runs, and perturbations in the type of processing load occur, the ideal adaptive algorithm will continually update the multi-dimensional field and repeat the detection of the highest maximum. These functions could be performed by a custodial microcontroller as described in section 7 below.

5. The software stack

Though many chips have Arteris probes and run the same operating systems, the number and types of IPs in chips vary as do their topology of their interconnectivity. It is important for industry that operating systems, and the software that runs on them, be portable between chips and that programmers be able to design software without knowing the particular details of specific chips. To be portable, the operating system kernel, which includes the thread scheduler, must be abstracted from the hardware. [3] If conventional thread schedulers are to be adapted for demand-based scheduling, which uses data from probes that are configured differently from chip to chip, a new probe-aware hardware abstraction layer (HAL) and probe drivers are needed. Note that whereas a HAL is a specific part of Microsoft Windows and Oracle Solaris, within the Linux ecosystem `udev` serves the functions of a HAL. [4]

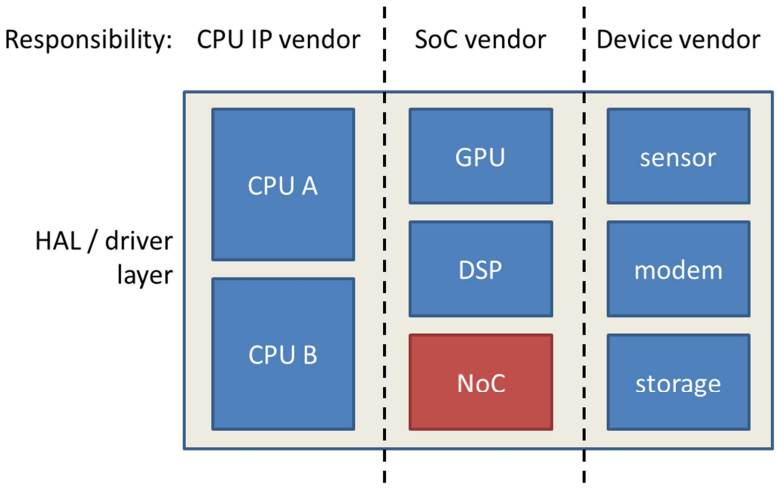


Figure 4: Vendor responsibility for HAL and driver layer software

Figure 4 shows the division of responsibility between hardware IP vendors for the HAL and driver software layer in an operating system. The CPU IP vendor creates a HAL for the OS and must provide it to SoC vendors. The HAL for other IPs, such as GPUs, DSPs, NoCs, and peripheral connectivity IPs must be assembled or created by the SoC vendor. IPs are often configurable, and therefore their HAL software is SoC-specific. SoC vendors must provide the HAL to end device vendors who must integrate drivers for the specific peripheral chips in their systems. Peripherals are ones such as sensors, modems, and storage.

The HAL

Different chips that support demand-based scheduling have different configurations of Arteris probes. Of particular importance are differences in the number of statistics counters and encodings of initiator IDs used to filter traffic of particular initiators. It is ideal to have at least one counter for each processor core hardware thread. In a configuration with fewer counters, they must be shared, and the HAL must allocate counters to SFTs and update SFT bandwidth statistics less often than every time an SFT is allocated a time slice. In either case, one more counter should be present in the system to count clock cycles.

To implement demand-based scheduling in a system with the benefit of the software knowing the period, duty cycle, and starting timestamp, only a single counter per SFT-running processor core hardware thread, configured to filter for its traffic, is necessary. In a system where the operating system is not in control of the period, duty cycle, or starting time of SITs, those attributes must be detected. This requires allocating bandwidth counters for SITs that can be used to sample SIT bandwidth in order to detect its attributes.

A HAL should provide functions to the operating system for the following:

- resetting a specified counter
- reading a specified counter value

The scheduler

The probe aware HAL provides a data structure that can be queried by the OS CPU thread scheduler. A baseline requirement is that the OS functions correctly even if no probe is present. Furthermore, an OS thread scheduler must support priority mechanisms and a guarantee of no starvation. A way to implement this is for the scheduler to use probe information to bias an otherwise default scheduling decision.

Linux scheduling

The Linux operating system, for example, associates one of five policies with each thread (SCHED_OTHER, SCHED_BATCH, SCHED_IDLE, SCHED_FIFO, and SCHED_RR). [5] In particular, SCHED_FIFO and SCHED_RR policies support real-time threads that must preempt threads of the other policies, and should be unaffected by resource demand.

Threads with a SCHED_OTHER, SCHED_BATCH, or SCHED_IDLE scheduling policy are chosen round-robin among all threads in a RUN state that have the same dynamic priority. The dynamic priority is increased by an amount inversely related to its `nice` value for every time slice that the thread is not chosen.

API for demand-based scheduling

The following is a proposed API for an OS kernel scheduler to call for execution by a HAL. Figure 5 shows a diagram of the relationship between the OS kernel scheduler layer and HAL layer.

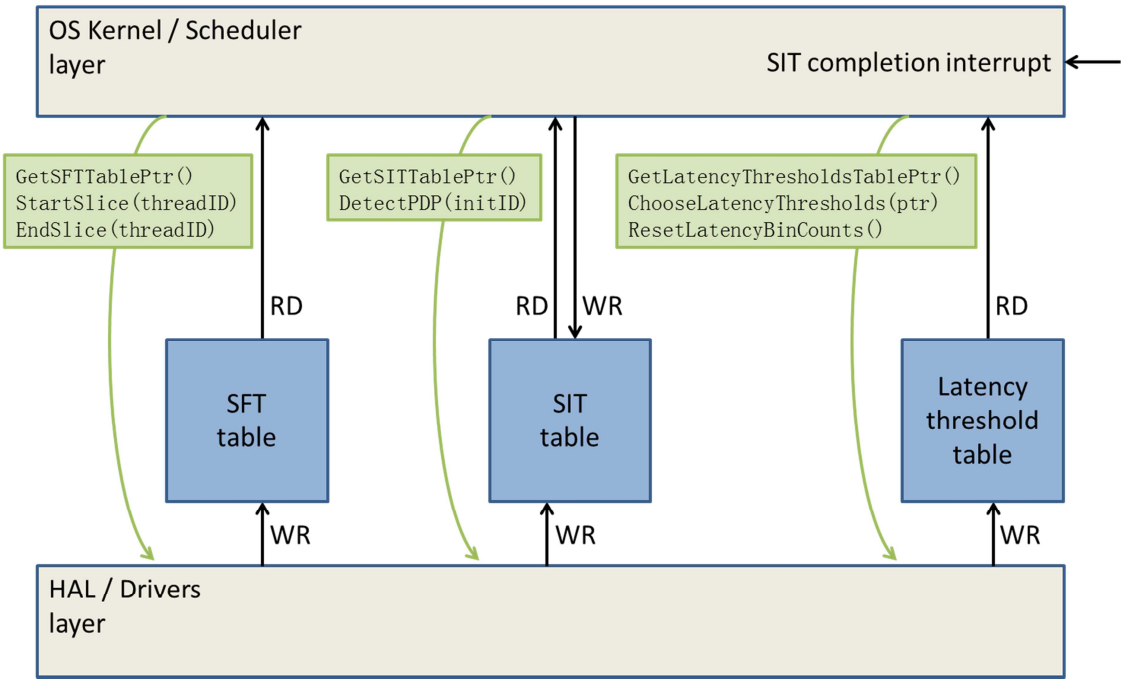


Figure 5: Scheduler/HAL API

The SFT, SIT, and latency threshold tables are as shown in Figure 6.

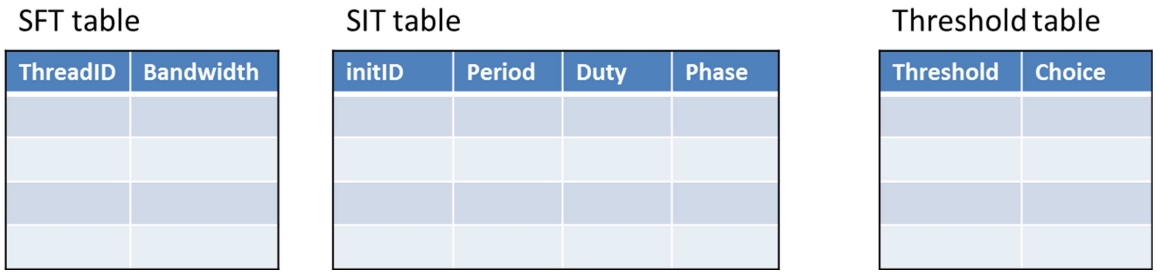


Figure 6: Software tables

SFT functions

`GetSFTTablePtr()` returns a pointer to a variable-length SFT table maintained by the HAL for thread entries allocated by the scheduler based on thread ID values.

`StartSlice(threadID)` indicates that the HAL should choose an available counter, reset it, and set it to filter on the initiator running the thread specified by `threadID`. If no counter is available then the function returns an error.

`EndSlice(threadID)` indicates that the HAL should read the value of the counter allocated to `threadID`, divide by the length of time that the thread ran, store the resulting calculated bandwidth in the SFT table, and sort the table in order of bandwidth usage.

SIT functions

`GetSITTablePtr()` returns a pointer to a SIT table allocated by a HAL.

`DetectPDP(initID)` indicates that the HAL should perform (or invoke on another processor) a period/duty cycle/phase detection procedure for one or more SITs running on initiator `initID` and update the SIT table.

Latency functions

`GetLatencyThresholdsTablePtr()` returns a pointer to a latency thresholds table allocated by a HAL.

`ChooseLatencyThresholds(ptr)` indicate to the HAL a set of threshold values to choose from a set of chip-specific choices.

`ResetLatencyBinCounts(initID)` reset the counts of read/write transactions within each latency bin.

Updating the SIT table

Any software may update the period, duty cycle, and latest start timestamp data in the SIT table for a SIT that invokes each period. If software does not invoke each SIT period, software can write the period or the period and the duty cycle information. A detection mechanism in the chip can detect the start of a period and generate an interrupt that writes the start timestamp. A more advanced detection mechanism can detect the completion of the SIT duty and generate an interrupt that writes the duty cycle value.

If the SIT is not controlled by software, then the table then a period/duty cycle/phase detection DSP algorithm can be run to update the table. This might be the responsibility of a microcontroller IP within the SoC.

Measuring latency

A latency table is needed if measuring the effectiveness of scheduling is to be implemented. It is impossible to distinguish latency for transactions that are due to different SITs, but the overall effectiveness of the scheduling can be determined.

A latency table stores thresholds that define the bins of a histogram of read/write transactions per bin latency range. The thresholds are selected from a chip-specific hardwired list. The OS scheduler is responsible for choosing the hardwired choices of threshold values in order to determine the latency histogram spread. An initial choice is likely to result in almost all transactions falling into either the highest or lowest bin. From there the scheduler must adjust the thresholds, reset the latency bin counters, and run again for another statistically significant number of SIT cycles.

Though the scheduler does not know the actual latency thresholds, since they are vary from one chip to another, it can detect the relative histogram spread across thresholds of using different scheduling parameters. This can be used for algorithm development or even for a real-time adaptive algorithm.

A basic implementation

One possible implementation of demand-based scheduling would be, for time slices that a thread is not chosen, increment its dynamic priority by an amount inversely related to its `nice` value and also by another amount related to the absolute value of the difference between its bandwidth and the sum of the bandwidths of all currently running SITs.

Bandwidths are measured from a probe just before the main memory interface, presented to the scheduler by the HAL. In chips with multiple interleaved memories, the HAL chooses just one, assuming statistically even distribution of transactions between memories. Alternatively, the HAL computes sums and averages of statistics from all memory interfaces for better correctness. The HAL presents to the OS scheduler the set of functions defined by the API above.

At boot time the OS runs the `GetSFTTablePtr()` and `GetSITTablePtr()` functions to find the tables allocated by the HAL. At the start of a time slice the OS scheduler calls `StartSlice(threadID)`, which causes the HAL to reset its access counter for the selected thread. At the end of a time slice the OS scheduler calls `EndSlice(threadID)`, which returns the average bandwidth used during the time slice that the thread ran. The OS scheduler writes that to the SFT table.

To choose the next SFT, the scheduler determines whether each SIT is active.

$$(Current\ time - Start\ timestamp)\ mod\ Period\ <?\ Duty$$

The result for each SIT, multiplied by its demand, is added together. That determines the aggregate demand on the shared resource. The next SFT is chosen based on the ratio of aggregate demand to total throughput capacity of the shared resource. A high ratio suggests a low demand SFT is preferable and a low ratio suggests that a high demand SFT is preferable.

When invoking a cycle of a SIT, the OS thread scheduler writes the start timestamp of the SIT. Routinely, all start timestamps must be updated so that time counter wrap-around doesn't cause a mistaken phase. This is done by overwriting the Start timestamp field with

$$Current\ time - \big((Current\ time - Start\ timestamp)\ mod\ Period\big).$$

This corresponds to the scheduling algorithm described in section 4 above.

An advanced implementation

A more advanced implementation of an OS CPU thread scheduler measures the magnitude, period, phase, and duty cycle of bandwidth used by SITs. The HAL presents to the OS a list of the devices in the system that can perform SITs and their device types. Based on the types of device, the scheduler associates them with OS functions. For example, a GPU device type would be associated with rendering the display and a DSP device would be associated with coding audio sample blocks.

The scheduler invokes HAL functions to set up the chip-specific probe and its filters to measure the bandwidth of transactions from the initiator associated with the device ID. By sampling bandwidths periodically, and performing an FFT, the scheduler determines the magnitude, period, and phase of the bandwidth demand of the SIT. That is used to improve the biasing used in the scheduling decision.

In chips that support a device-management microcontroller, it can be programmed to do the measurements. Ideally, future chips will be designed with such microcontrollers to manage power control and scheduling.

Aside from the above-mentioned functions that a HAL might provide to the scheduler within an OS, OSes should also be designed to provide scheduling-related services to higher level software.

6. Results

The benefit to system performance of demand-based scheduling depends on various attributes of the system. The scheduler can do best if it knows the start and end time of the phase of high demand of SITs. Failing that, the scheduler can do better if the SIT task demand can be measured. The performance will be even better if the period and phase of SITs can be detected. The scheduler also has an advantage if it can give variable duration time slices to SFTs. The scheduling will also be better if the scheduler has a measurement of demand for each SIT and each SFT.

Conventional scheduling

Conventional schedulers consider neither SITs nor the demand requirements of the SFTs to be scheduled. As a result, high and low demand threads are interleaved. The aggregate demand for the shared resource is bursty. As shown in Figure 7, aggregate demand goes through peaks and troughs.

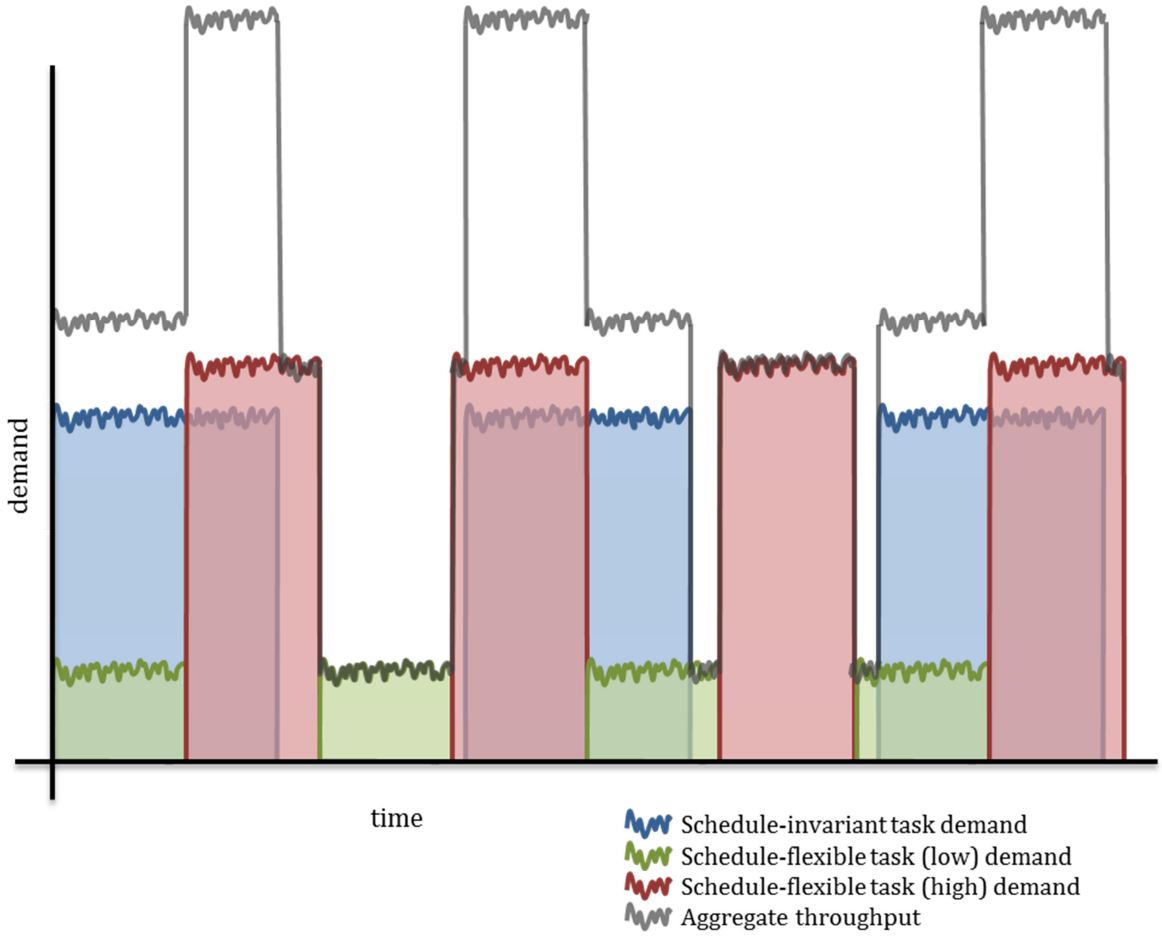


Figure 7: Conventional scheduling

When a SIT and a high demand SFT are simultaneously active, aggregate demand is high. This can occur for extended periods of time, and requires the shared resource to be designed to handle the peaks, though otherwise lightly used.

The following are case studies showing the results of demand-based scheduling algorithms within different system constraints.

Ideal scheduling

Ideally, the scheduler knows exactly when SITs will begin and end their periods of high demand and can give flexible time slices to SFTs. In this case, as shown in Figure 8, aggregate demand will be quite steady.

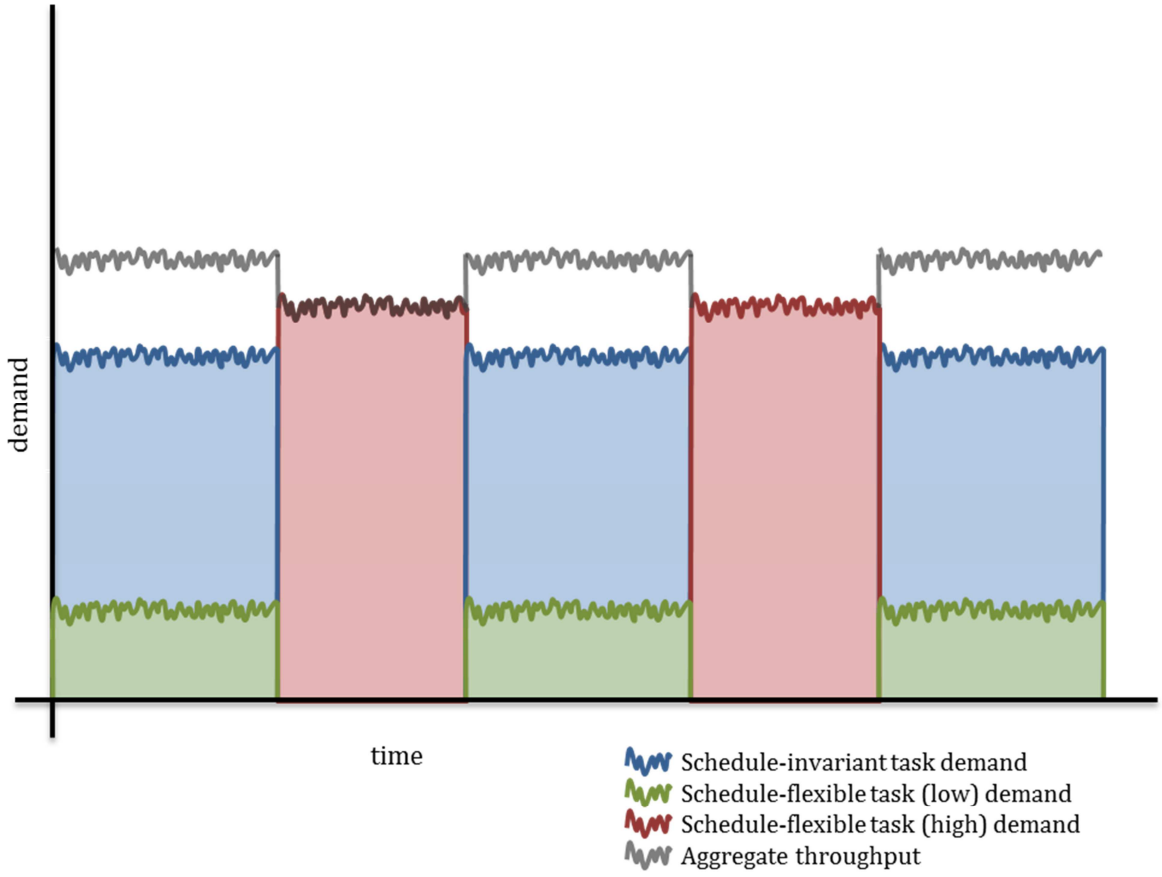


Figure 8: Ideal scheduling

The shared resource can be designed less expensively since it need not handle such demanding peaks of activity.

Probabilistic scheduling

In some systems the scheduler knows when a SIT begins its high demand phase, but does not know when the SIT becomes idle. This is the case, for example, in video codecs. I frames have very different resource demands than P and B frames. The demand of such as system is shown in Figure 9.

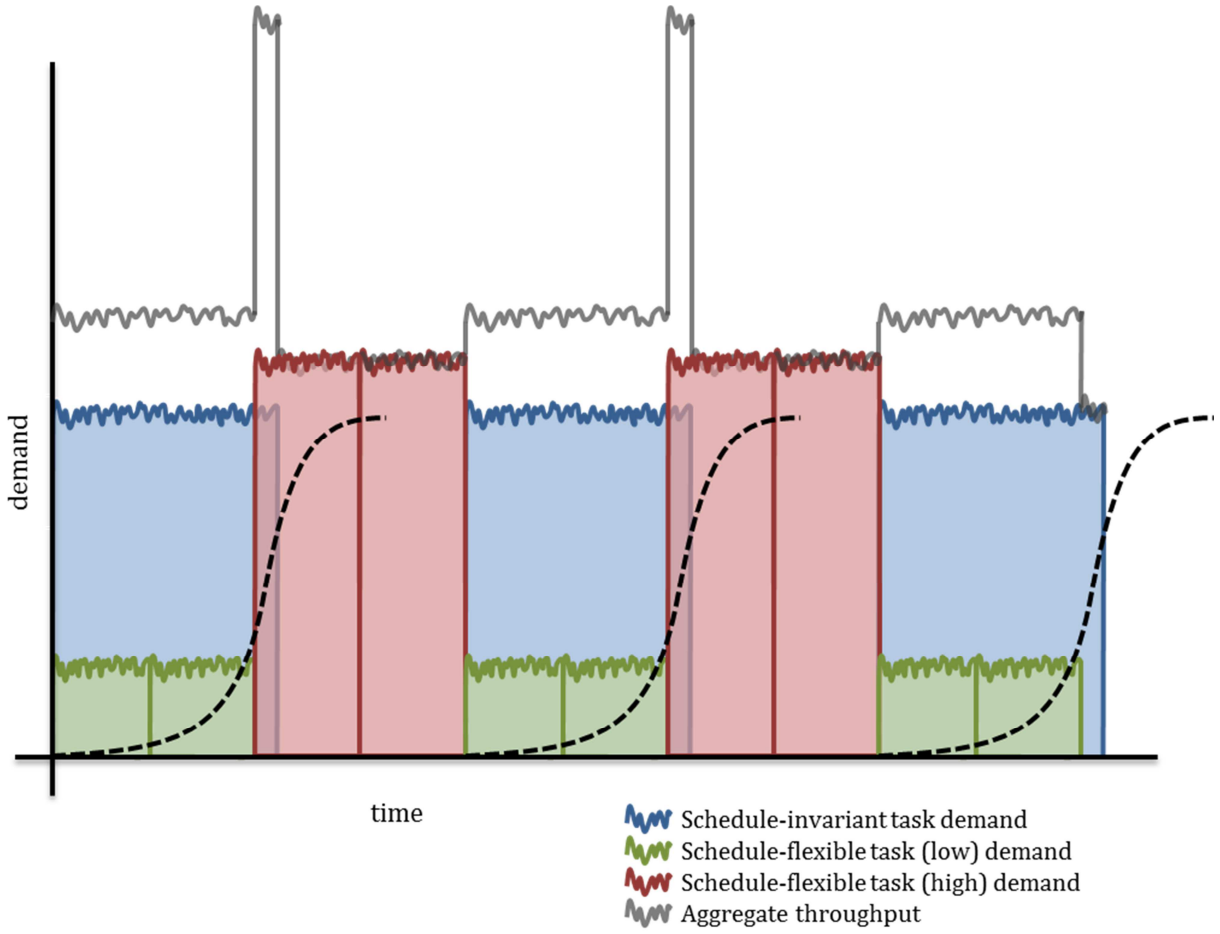


Figure 9: Scheduling with probabilistic SIT completion time

The scheduler does not know the completion time of the SIT, but can use a model of the probability model of its completion. A probability curve is shown as a dashed line in the figure. Note that a video codec with lower frame processing time deviation will have a sharper probability curve. [6]

Tasks to be scheduled are presented in a table and sorted. Based on the probability at the time of choosing a new task, the scheduler gives preference to tasks based on their location within the table. Low demand tasks are preferred when the probability is low and high demand tasks are preferred when the probability is high.

Though durations cannot be ideally scheduled, in this case, performance is improved if time slices are short.

Scheduling with detected SIT completion time

If it is possible to detect the end of the high demand phase of a SIT then the scheduler can know, for sure, whether to favor high or low demand SFTs. Knowing the start time of the next SIT active period, the scheduler can also adjust the time slice duration to last just until that time. The demand graph for such scheduling is shown in Figure 10.

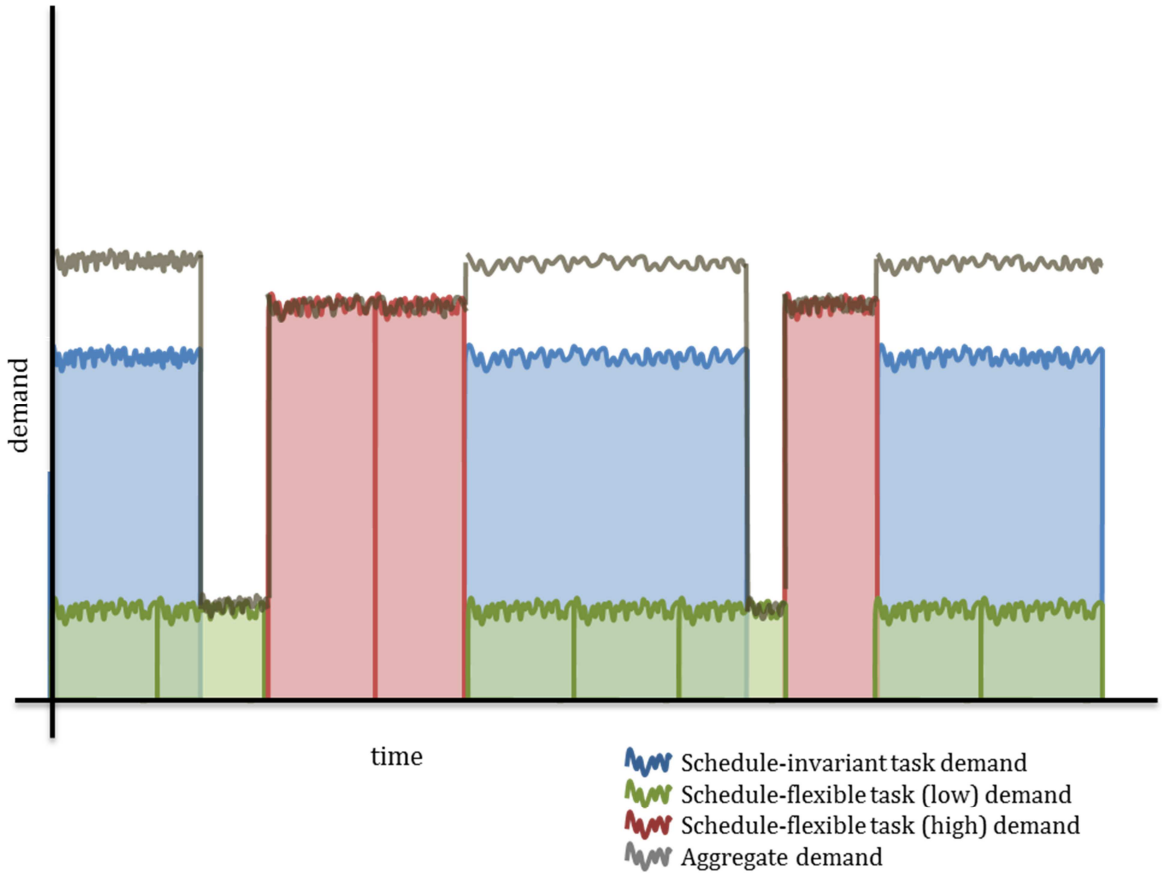


Figure 10: Scheduling with detected SIT completion time

At the time of scheduling a new task, whatever is the status of the SIT demand determines whether a low demand or high demand SFT is favored. As in case 2, performance is improved if time slices are short.

One way that detection can be done is by frequent checking of a transaction counter within a probe. Rather than doing this within the operating system running on the main host processor, a simple peripheral state machine or microcontroller can be used to monitor demand and interrupt the operating system as needed.

Scheduling with FFT

As in case 3, scheduling is based on detection of the completion of the SIT high demand phase. However, the SIT is completely independent of the scheduler. To use demand-based scheduling, the scheduler, or a supporting microcontroller, samples probes at frequent regular intervals. The samples are processed with an FFT or similar transform in order to determine the period, phase, and magnitude of SIT demand.

Latency histograms

A comparison of conventional and ideal demand-based scheduling aggregate demand is shown in Figure 11.

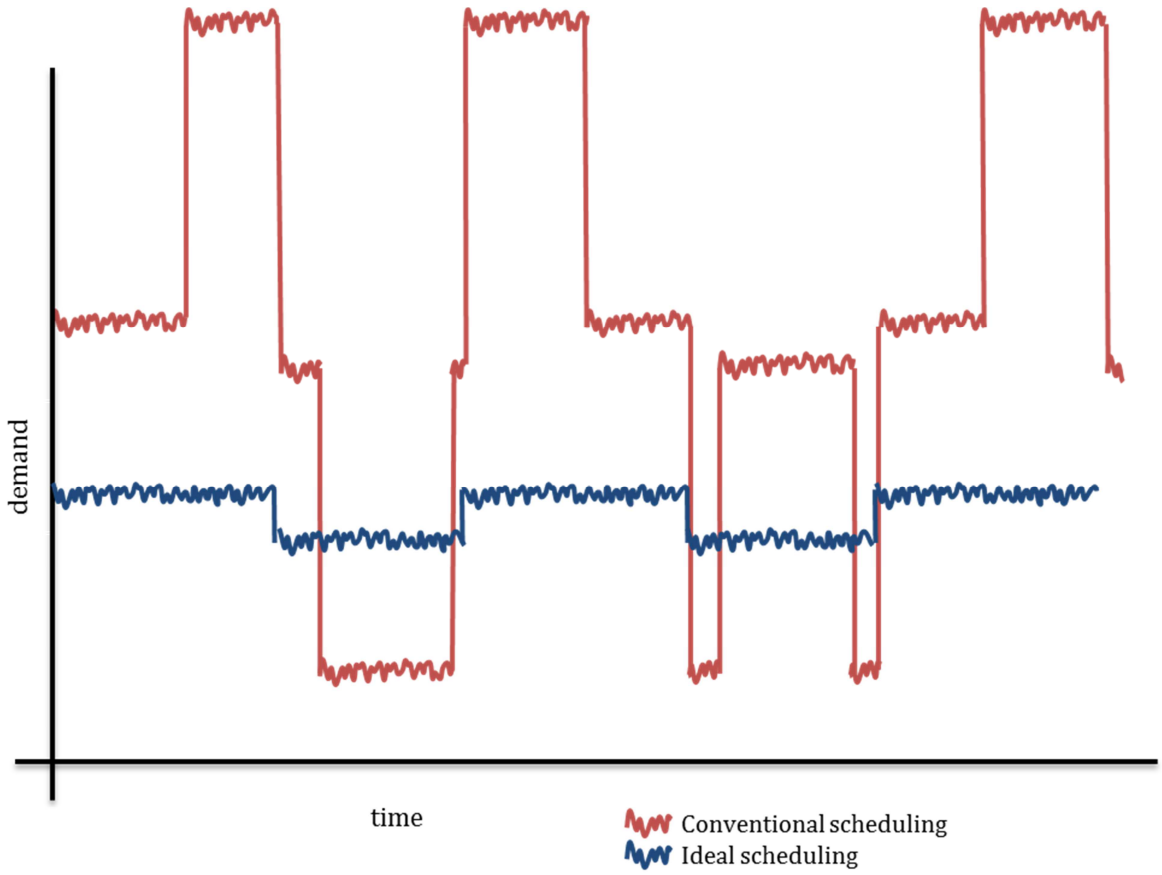


Figure 11: Aggregate demand of different scheduling algorithms

Read and write transactions are numerous during each time slice. On average transactions receive fast responses when resource demand is low and slow response when resource demand is high. A histogram plot of the number of transactions over latency can be plotted. Figure 12 below shows two histograms, one for conventional scheduling, and one for an ideal implementation of demand-based scheduling.

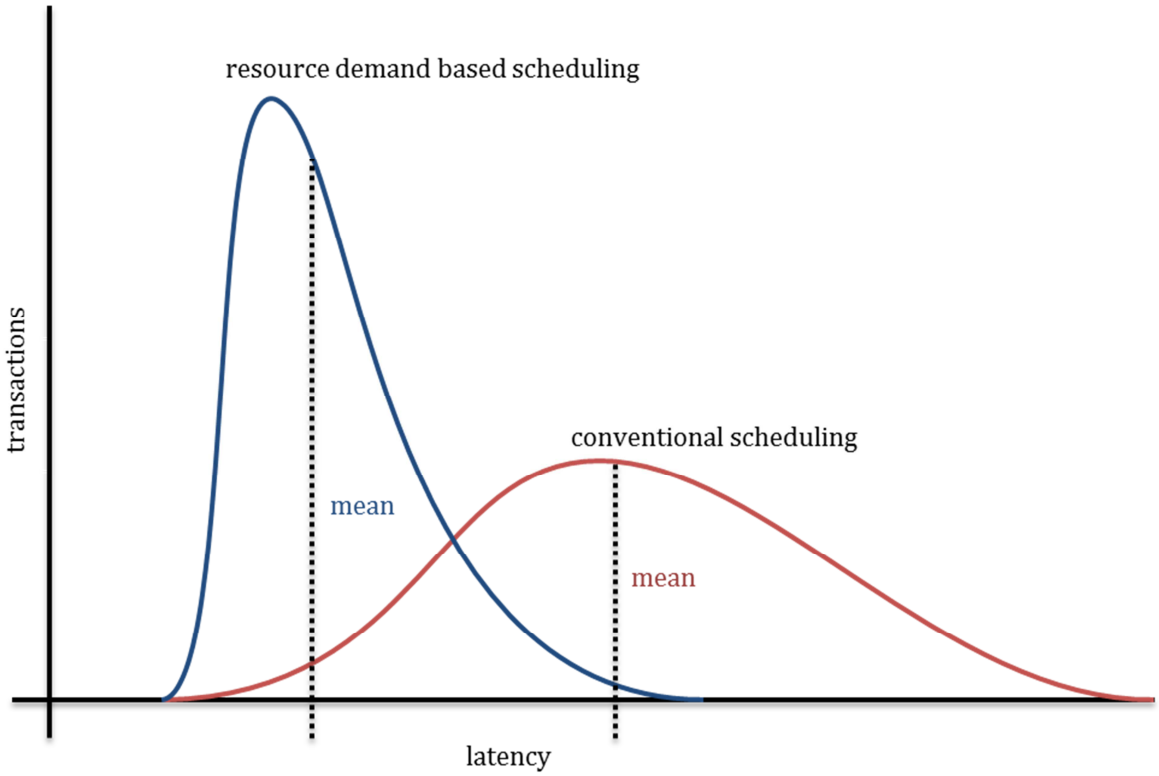


Figure 12: Latency histograms of different scheduling algorithms

There is a certain minimum latency for all transactions, which is apparent in the initial offset of the curves from the zero latency point. With demand-based scheduling, the spread of the latency curve is less, and the worst occurring latency is shorter than for conventional scheduling. In this example, the mean latency for demand-based scheduling is roughly half of the mean latency for conventional scheduling.

There are several significantly beneficial results from a tighter latency distribution:

- The average access latency to the shared resource is lower. This is true for all tasks. Lower average access latency improves performance of most initiator IPs. This is especially beneficial to CPUs that are performance-bound by main memory read accesses for cache misses. Improved performance allows initiator IPs to complete their processing jobs while running at a slower clock speed. Logic is able to run tolerate lower skew rates and thereby run at lower voltage or be designed with lower threshold voltage (V_T) transistors. Alternatively, improved performance allows initiator IPs that run fast to complete their processing jobs sooner and thereby spend more time in power-saving clock-off or complete power-down states.
- A decreased in available peak bandwidth pushes the latency histogram out. The decreased average access latency due to demand-based scheduling allows a system to meet real-time schedule deadlines with a lower available peak bandwidth to the shared resource. This means that a lower speed or narrow interface can be used to the shared resource. Alternatively, for a given available peak bandwidth, more processing can be accomplished.

- To tolerate unpredictable latencies, some initiator IPs have internal data storage buffers. The larger the access latency variation (histogram spread) the larger the storage buffer capacity must be to ensure that over-runs and under-runs don't occur. With perfectly predictable latency an initiator IP would have no need for any buffering. Therefore, deterministic latency reduces the buffering requirements of initiator IPs. That reduces sil icon area requirements in SoCs.
- Many SoCs have their performance limited by access to shared memory interfaces. An appropriately-designed memory interface subsystem gives the best performance when the patterns of accesses are sequential. Demand-based scheduling, by separating the periods during which different high-demand tasks run, decreases the average diversity of requests made to the memory. To the extent that accesses for any particular task are sequential, this ensures that the DRAM memory scheduler will have more choices of sequential accesses to issue to the DRAM chip. This allows more efficient scheduling, and thereby higher throughput from a DDR interface.

In the SoC design process, many interdependent aspects affect each other. The dependencies, and particularly the chain of causation of system design benefits by which demand-based scheduling provides lower power is shown in Figure 13.

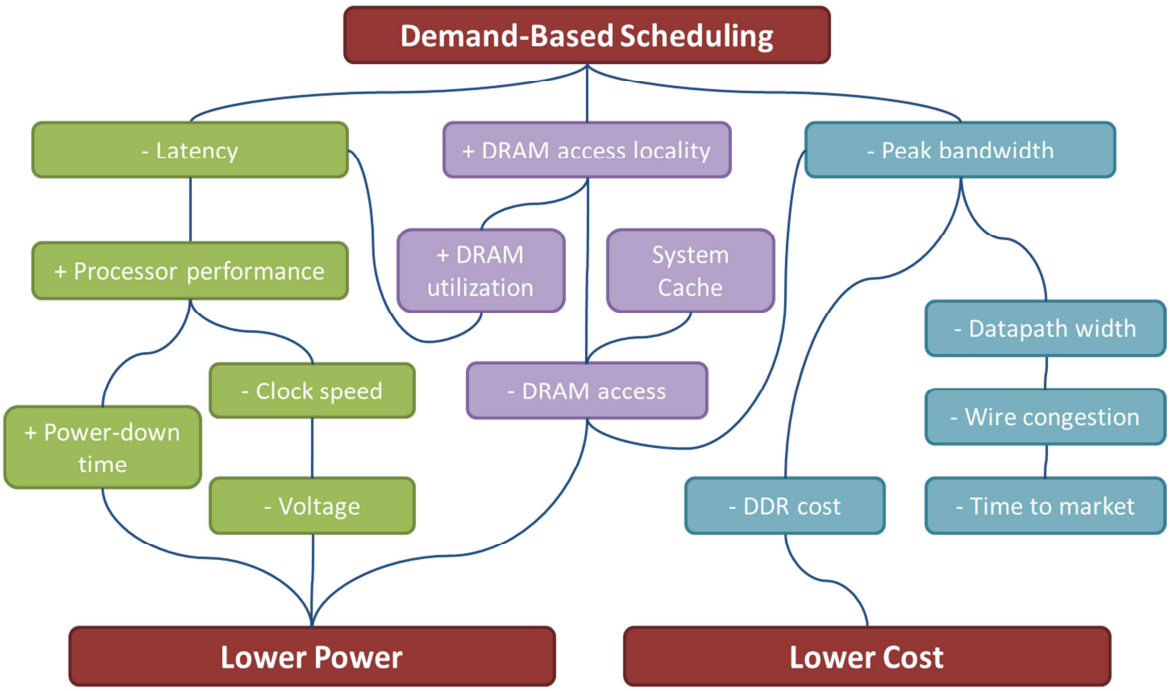


Figure 13: System design benefits causation network.

Practical considerations

In practice, many systems have multiple SITs with differing periods, phases, and magnitudes. For example, a video recorder captures frames of video, pre-processes them with the same frame rate period but a different phase, captures audio at a different rate of sample blocks, and writes a muxed bitstream at a different rate of network allocation unit bandwidth. An appropriate scheduling algorithms should consider as many SITs as it can. The consideration of

each should be weighted by the magnitude of its demand. For example, a compressed bitstream has much less resource demand than uncompressed video frames.

Furthermore, many systems have multiple resources. The main memory subsystem is the primary critical resource in many systems, but other storage such as Flash, or disks can be critical in some cases. Scheduling based on demand for a shared on-chip system cache can significantly improve its miss-rate, thereby reducing the number of main memory accesses. In other cases, network bandwidth such as on Wifi or LTE networks can be critical.

A scheduling algorithm must prevent starvation and system unbalance. Therefore, the choice of task from demand-based scheduling should be a weighted contributor to other scheduling considerations such as fairness.

7. Heterogeneous systems

A heterogeneous processing system is one in which multiple types of processing units run software concurrently. An example is a system on chip with a CPU cluster, a GPU cluster and a DSP. The complexity of thread scheduling increases in the case of general purpose processing, where a software application or thread can be run on either the CPU, GPU or DSP. In this case, middleware must be run on each processing unit to enable information sharing for assigning a thread to a processing unit and for scheduling its execution on the processing unit.

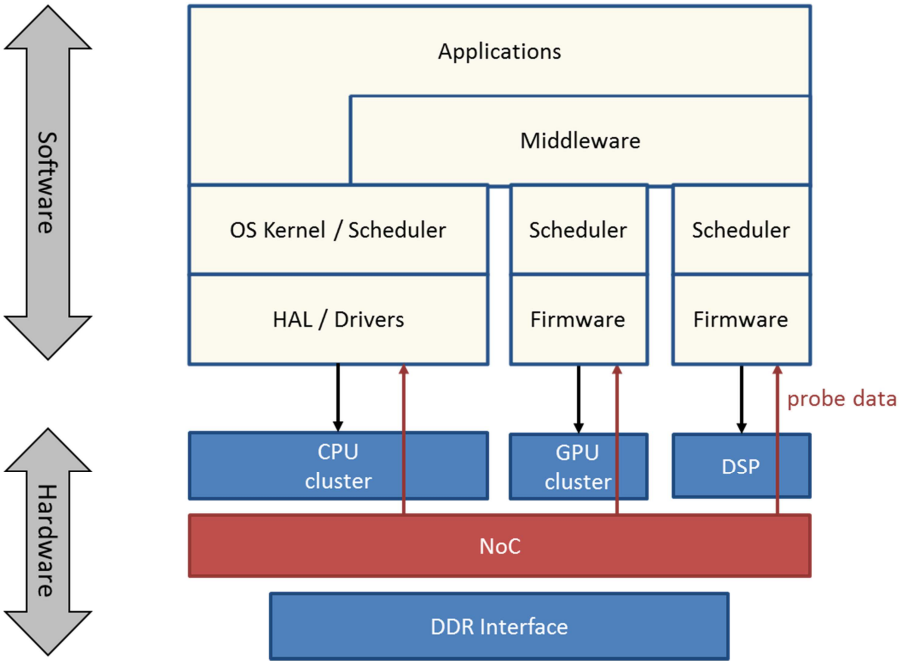


Figure 14: A software stack for heterogeneous processing

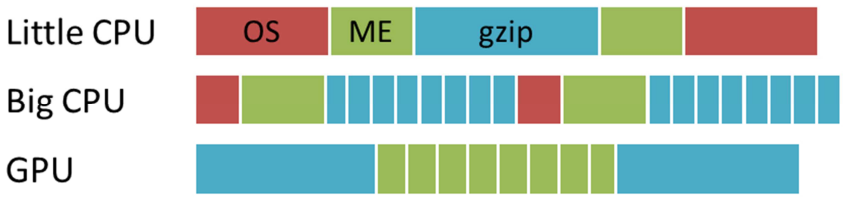
Figure 14 shows the processor, memory, and NoC hardware. It also shows the software stack built on them. In modern SoCs, CPUs are generally implemented in clusters of homogeneous processors. CPU clusters run an OS, which interacts with hardware through the HAL and driver software. The OS kernel schedules threads, and may do so using data from NoC probes. Similarly, task scheduling software runs on other processing IPs such as GPU clusters and DSPs. The scheduling software is abstracted from the hardware through firmware. Some application

software only makes sense to run on a CPU. For application software that is built from system call tasks that can be run on any of a number of different processors, a middleware layer assigns tasks to different processors.

Performance affinity

An ideal heterogeneous scheduler considers each task's dynamic priority, static priority, and performance affinity. Performance affinity is a measure of the capability of a processing IP to yield good performance for the task. Performance affinity clearly depends on efficiency of execution. Performance affinity can be dynamic. In a heterogeneous processing system with demand-based scheduling, performance affinity is also biased by the demand of a processor on a shared resource, as determined by sampling probe data, while running the task.

Consider a task that cycles through a data set that is slightly larger than the cache of a processor. The resulting compulsory cache misses will cause a lot of demand for a shared DRAM. On another processor with cache capacity to fit the data set, no DRAM demand will be required. The resulting effect on system performance is best considered in the computation of the performance affinity table.



Performance affinity table

tasks	Little	Big	GPU
OS service	.25	1	0
Motion estimation	.4	.4	1
gzip	.1	1	.1

Figure 15: Heterogeneous processor affinity

An example of three processes (an OS service, motion estimation, and gzip) run on three different processors (a little CPU, and big CPU, and a GPU) is shown in Figure 15. OS services must be run on a CPU, and have a mix of branching and memory accesses, therefore running better on a big CPU than a little CPU. It's performance affinity is higher for a big than a little CPU, and zero for a GPU, indicating that it should never run on a GPU. Gzip has a lot of branching for relatively little memory access, and therefore performs much better on a big than on a little CPU. Gzip can be implemented on a GPU, but the parallelism of a GPU is of no benefit, and the overhead causes gzip to have poor performance and therefore a low affinity for the GPU. Motion estimation requires a lot of data access and parallel computations, which run very well on a GPU, but not well on CPUs, which would have I/O limited performance.

HSA

The primary role of the middleware is to identify available computing resources and to choose an optimal processing unit on which to run each time slice of a task, thread or process. Middleware uses shared data constructs that are read and written by operating software on each involved processor. Each different processor may run different operating systems or the same operating systems with different HALs, but middleware must be standardized across processor architectures.

The Heterogeneous System Architecture (HSA) is such a standard [7]. It defines a method by which application software can be automatically translated to run on any of a number of different processors, their workloads can be automatically distributed, and the processing of each unit allocated a timeslice on a chosen processor. The decision is made by something known as a finalizer, which makes its determination while the application software is compiled, when it is installed, when it is loaded, or in real-time during execution. Furthermore, the choice of processor can be made using demand-based scheduling based on on the bandwidth per processor per task.

A custodial microcontroller

A custodial microcontroller is a programmable processor of modest performance – apart from the main host or application processor – that can be used to monitor the system and manage its controllable parameters. Few SoCs today are arranged to have a custodial microcontroller that can participate in task scheduling. Having such a custodial microcontroller may provide scheduling benefits in future chips.

In advanced implementations of demand-based scheduling, a custodial microcontroller:

- samples DRAM statistics, per initiator, at regular intervals
- processes the samples to determine phase, period, duty cycle, and magnitude of each SIT
- monitors OS or middleware status variables to determine which SFTs are running
- samples initiator latency histograms to determine scheduling effectiveness
- signals to the OS or middleware the demand-smoothing preference for each SFT

Offloading global thread scheduling to hardware may provide latency and power consumption efficiency benefits.

8. Conclusion

Demand-based scheduling is the scheduling of schedule-flexible tasks, based on the state of schedule-inflexible tasks with regard to demand for a shared resource. Doing so minimizes contention, which yields shorter and more predictable response latency. Demand-based scheduling also improves the locality of sequential requests for a resource, which in the case of DRAM improves its throughput. These factors improve the performance of processors and the system as a whole. Improved system performance allows for relaxed design constraints, enabling optimizations for lower power consumption and cost.

9. Bibliography

[1] H. Hoffman, J. Eastep, M. D. Santambrogio, J. E. Miller and A. Agarwal, "Application Heartbeats," in *ICAC'10*, Washington, DC, USA, 2010.

[2] Texas Instruments, OMAP4430 Multimedia Device, 2011.

[3] L. F. Bic and A. C. Shaw, Operating Systems Principles, Prentice-Hall, 2003.

[4] "udev," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Udev. [Accessed 20 09 2013].

[5] M. Kerrisk, "pthread_setschedparam(3) - Linux manual page," 17 09 2013. [Online]. Available: http://man7.org/linux/man-pages/man2/sched_setscheduler.2.html. [Accessed 26 09 2013].

[6] J. Probell, "http://probell.com/pubs/video_frame_processing.pdf," 05 2008. [Online]. Available: http://probell.com/pubs/video_frame_processing.pdf. [Accessed 24 09 2013].

[7] "HSA Foundation ARM, AMD, Imagination, MediaTek, Qualcomm, Samsung, TI," 2012. [Online]. Available: http://hsafoundation.com/. [Accessed 25 10 2013].