RC23488 (W0501-036) January 11, 2005
Computer Science

IBM Research Report

Scheduling for Heterogeneous Processors in Server Systems

Soraya Ghiasi, Tom Keller, Freeman Rawson
IBM Research Division
Austin Research Laboratory
11501 Burnet Road
Austin, TX 78758

Scheduling for Heterogeneous Processors in Server Systems

Soraya Ghiasi, Tom Keller and Freeman Rawson
IBM Austin Research Laboratory
11501 Burnet Road
Austin, Texas 78758
{sghiasi,tkeller,frawson}@us.ibm.com

Abstract

Applications on today's high-end systems typically make varying load demands over time. A single application may have many different phases during its lifetime, and workload mixes show interleaved phases. Memory-intensive work or phases may exhibit performance saturation at frequencies below the maximum possible for the processors due to the disparity between processor and memory speeds. Performance saturation is a sign of over-provisioning and leads to energy-inefficient systems. Computers using heterogeneous processors, with the same ISA, but different implementation details, have been proposed as a way of reducing power while avoiding or limiting performance degradation. However, using heterogeneous processors effectively is complicated and requires intelligent scheduling.

The research reported here explores the use of a heterogeneous system of processors with identical ISAs and implementation details, but with differing voltages and frequencies. The scheduler uses the execution characteristics of each application to predict its future processing needs and then schedule it to a processor which matches those needs if one is available. The predictions are used to minimize the performance loss to the system as a whole rather than that of a single application. The result limits system power while minimizing total performance loss. A prototype implementation on a Power4 four-processor system is presented.

1. Introduction

The primary design concern for processors has, until recently, been performance. As technologies continue to scale to smaller feature sizes and more transistors are placed on a die, power has also become a first order design constraint. Problems that have long fallen in the domain of small handheld devices are now faced by large systems. The details and focus may be different, but the underlying issues remain the same. While the primary problem for embedded and laptop computers is battery life and, thus, total energy consumption over time, the most important problem for servers and server clusters is maximum power [1]. Servers have limitations on their internal power-delivery and cooling systems as well as site limits on the total power and cooling available in the external environment. Processor frequency and voltage scaling has been studied as a way to reduce processor power, which is often the most important contributor to system-level power consumption [1].

However, dynamic voltage scaling comes at a cost – when the voltage is changed, some amount of processing time is lost as the voltage settles. Although settling times are getting smaller, they do place limits on the number of times the voltage may be changed in a given time period. For example, in a 1GHz system using a 10-millisecond scheduling quantum, no interruptions and a settling time of 30 microseconds, the performance loss due only to changing the voltage is negligible (~0.3%). In a system with interrupts, the actual effective time a task runs without interruption may be much shorter leading to larger performance losses (~1%) due to more voltage changes. Actual performance losses may be much higher because some sort of monitoring and prediction is needed to accurately identify the appropriate voltage and frequency for each task in the system.

This paper considers an alternate approach. Rather than identifying the desired frequency and changing the voltage and frequency to match, it investigates a simpler system with fixed frequencies and voltages. The processors in the system run at different, but fixed frequencies, and tasks are assigned to appropriate processors based on their frequency demands. This approach can be used to control maximum power dissipation in SMP servers and offers the possibility of a further extension to server clusters.

Earlier work by Kotla, et al., [2] demonstrates that workloads vary in their level of memory intensity, both between different workloads and over their execution lifetimes. Given that secondary cache and memory performance are unaffected by processor frequency scaling, memory-intensive workloads exhibit performance saturation at a characteristic frequency related to their level of memory intensity. Raising the frequency above the saturation point yields no further benefit in performance. Figure 1 demonstrates this property for a simple benchmark with the ability to set the memory intensity via a command line argument to the benchmark.

Kotla's work considered only a single application at a time and used post-processing to determine the most appropriate frequency for each task based on aggregate measures of the task over its lifetime. This research instead extends the Linux operating system scheduler to dynamically monitor and place tasks in a heterogeneous system. The new scheduler is called the task-to-frequency scheduler (**TFS**).

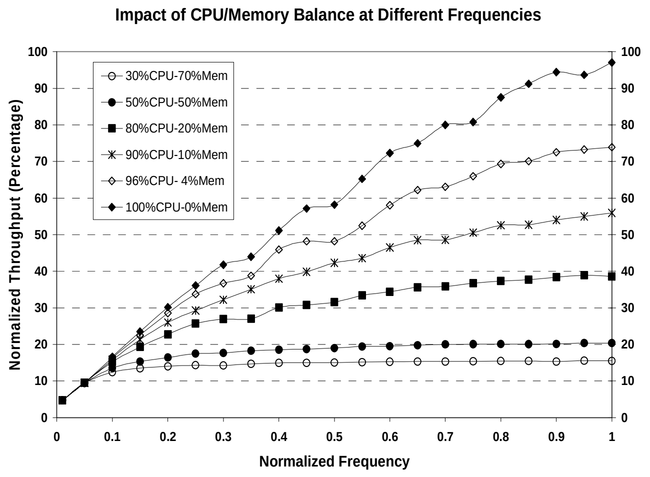


Figure 1: Performance saturation with a synethic benchmark. Taken from Kotla, et. al [2].

The **TFS** may be considered an extension of traditional complexity-effective designs which respond to varying demands for the core and memory subsystems by reconfiguring components to meet current demand. The reconfiguration seen by a task is the selection of a frequency and subsequent scheduling to that frequency rather than a direct change in the features of the core. This approach to reconfiguration makes this work more similar to the recent work on general purpose heterogeneous processors than to traditional complexity effective designs. Like some recent work on heterogeneous processors ([4], [5], [6], [9], [10], [11]), this effort uses processors that have the same instruction architecture but offer different implementation characteristics including different performance and power levels by using different microarchitectures or different but fixed frequency and voltage settings. The research presented in this paper makes a number of contributions beyond the prior work in the area.

- It uses the known phenomena of workload diversity and performance saturation along with the predictive performance model of Kotla, et. al[2] to determine the appropriate frequency setting for each task based on its observed behavior.
- It introduces task-to-frequency scheduling.
- It applies task-to-frequency scheduling rather than frequency and voltage scaling, to SMP servers.
- It minimizes the overall system performance lost.
- It reduces total power consumption of the system by using lower voltages and frequencies for some processors.
-

The results reported here represent a prototype implementation and its initial evaluation rather than a definitive study of the underlying ideas and techniques.

2. Related Work

This research extends the investigation that started in [2], but it draws upon other, related work in frequency and voltage scaling and designs with heterogeneous processors. The goal of the current paper is to offer a scheme that controls maximum power and minimizes average power by scheduling tasks to the appropriate frequency available in a system with processors using a small set of fixed frequencies and voltages.

2.1. Dynamic Frequency and Voltage Scaling

Transmeta's LongRun [7] and Intel's Demand Based Switching [8] respond to changes in demand, but do so on an application-unaware basis. In both schemes, an increase in CPU utilization leads to an increase in frequency and voltage while a decrease in utilization leads to a corresponding decrease. Neither one makes any use of information about how efficiently the workload uses the processor or about its memory behavior. Instead, they rely on simple metrics like the number of non-halted cycles in an interval of time.

Flautner and Mudge [3] explored the use of dynamic frequency and voltage scaling in the Linux operating system with a focus on average power and total energy consumption. They examined laptop applications and the interaction between the system and the user to determine the slack due to processor over-provisioning. They used frequency and voltage scaling to reduce power while consuming the slack by running the computation slower. Their Vertigo system dynamically uses multiple performance-setting algorithms to reduce energy.

Elnozahy, et al. [15] extended the ideas found in Flautner and Mudge[3] to the domain of web server farms. They explore the use of DVS to respond to changes in server demands. They also examine the use of request batching to gain larger reductions in power during periods of low demand. The two techniques compliment each other, but neither provides a means to address peak power

This work differs by responding to easily observed changes in memory subsystem demands rather than to changes in the CPU utilization metric. The latter approach takes advantage of system idle, while the former approach takes advantage of pipeline idle. Tasks are assigned to frequencies based on the characteristics of the memory subsystem accesses.

2.2. Heterogeneous Processors

The scheduling scheme described in this paper uses an environment in which an SMP server has heterogeneous processors that differ in frequency and voltage. Prior work on single-ISA, heterogeneous processors falls into two distinct categories. The first uses a processor family which may be run at the same frequency, while the second category uses a processor family which cannot be run at the same frequency.

Single frequency heterogeneous processors have been studied by Kumar, et al. ([4], [5], [6]). Their work uses different generations of the Alpha processor family scaled into the same technology generation and run at the same frequency. The goal of the work is to minimize energy consumption while maintaining performance. The authors use a variety of metrics to identify which tasks should be assigned to which core, with all cores running simultaneously. Trial-and-error testing is used to identify the best-suited core. In contrast, this paper predicts performance to find the appropriate core.

Ghiasi and Grunwald ([9], [10], [11]) explored single-ISA, heterogeneous cores of different frequencies for controlling the thermal characteristics of a system. Applications are run simultaneously on multiple cores and an operating system component is introduced to monitor and direct applications to the appropriate task queues.

This work uses a single generation in IBM's PowerPC processor family, but the cores are run at different frequencies. It also differs from prior work by using a commercial product and direct evaluation of the proposed techniques on real hardware, rather than relying on simulation.

2.3. Hardware Approaches

There is some work other than that reported in [2] that attempts to manage power while minimizing performance loss by predicting the impact of the power-management actions on the performance. In particular, Stanley-Marbell, et al, [12], propose a hardware mechanism that does frequency selection based on predicted performance loss. Like this work, it makes use of the memory behavior of the workload to determine when the processor can run more slowly due to a heavy use of the cache and memory subsystem. It differs by virtue of its focus on microprocessor changes for the uniprocessing environment. The details of the performance model are also quite different since [12] works from processor/memory overlap values, which are generally unavailable on standard hardware, while the scheme presented here uses access counts.

2.4. Performance Counter-based Scheduling

Phase detection is important to any real world system which is designed to take advantage of the variability of applications. Sherwood, et al.[16] provides the most detailed analysis of phase detection, but their work uses offline phase analysis. They use different performance-related metrics and correlate them to different phases of a program. Dhodapkar and Smith [17] compared the use of working set signatures, basic block vectors and conditional branch counters and illustrated the tradeoffs between identifying stable phases and phase length. Dynamic phase detection is more prone to error and is not as well studied at the operating system level. The simple performance model developed here can be used for dynamic phase detection.

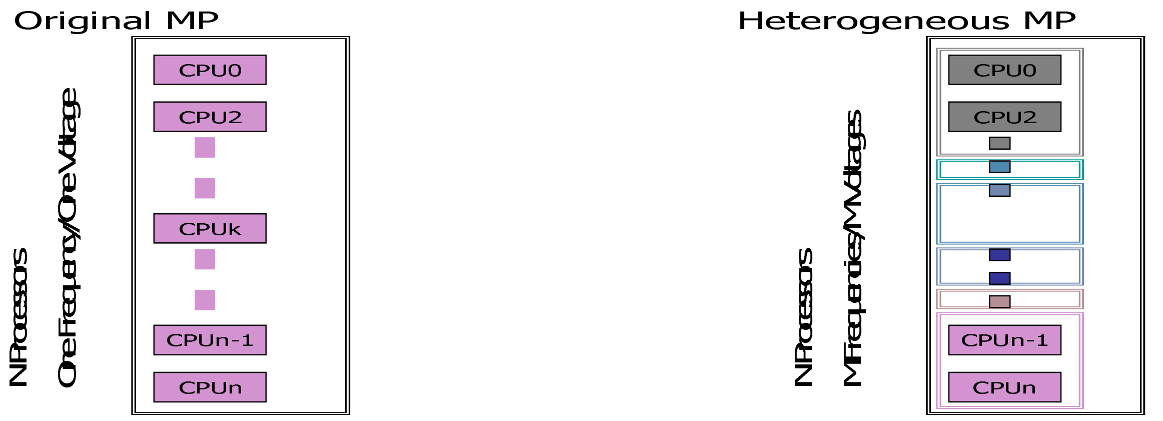
Most recent work in the area has been on identifying threads to run simultaneously on multi-threaded systems via the use of core metrics accessible through performance counters. Snavely and Carter [18] used trial and error on a subset of possible application combinations to determine which applications can be run together with minimal impact on the Tera MTA. Snavely differs from most work in this area by performing experiments on commercial hardware. Kumar, et al, [5] also detects and responds to changes in behavior through the use of trial-and-error. Ghiasi [11] used temperature-defined phases, relying on simulated temperatures rather than performance counters to detect phases.

3. System Design

This work considers a multiprocessor system in which the different processors run at different, but fixed frequencies. The different processor frequencies are accompanied by different voltages since lower frequencies allow processors to run at lower voltages. The frequency set used by the processors may be changed by some external mechanism, and the **TFS** responds to the new frequency set.

Although a server cluster is a form of multiprocessor, this paper concentrates on studying SMP systems, leaving a detailed investigation of clusters to future work. A simple example of the type of heterogeneous system studied here is shown in Figure 2. The original system is composed of N processors, all of which use the same voltage and frequency. The heterogeneous system is composed of the same N processors, but each processor may have a different frequency and voltage. At each frequency, the minimal voltage necessary to reliably drive that frequency is chosen. For this paper, the frequencies and voltages are fixed. In a more general system planned for the future, the frequencies and voltages are not permanently fixed. Instead, the frequency and voltage set might occasionally change in response to large-scale changes in system demands as well as changes in power constraints which force changes in frequencies, or even changes in environmental conditions such as temperature.

On a system with processors at different frequencies, a mechanism is needed to determine which frequency to assign for each task. This section presents a methodology for predicting the performance impact of the different possible frequency settings, given counts of the cache and memory accesses, and then using the predictions to guide the assignment of tasks to frequencies in order to meet power constraints and reduce performance loss.



$$P_{os} = n * P(V, F) >= P_{hs} = \Sigma n_i * P_i(V_i, F_i)$$

Figure 2: Heterogeneous Multiprocessor System

3.1. Performance Saturation

The reason that it is often possible to use a lower processor frequency is that some workloads cannot make use of all of the available frequency due to the latencies associated with cache and memory accesses. This phenomenon is referred to here as *performance saturation.* Even when the maximum power constraint is severe enough to require some performance penalty for all tasks, it is generally possible to take advantage of performance saturation to minimize the overall performance penalty of the power-management action. The intuition is that many programs obtain a limited benefit from increasing processor frequency due to the slow speed of memory relative to the processor. Thus, at some point the speed of a program making memory references is limited by the speed of the memory. The ratio of memory-intensive to CPU-intensive work in a workload determines the saturation point as illustrated for a simple synthetic program by Figure 1. Each task in the system has different characteristics and a different saturation point. The saturation point may change during the lifetime of a task as the task itself changes phases.

3.2. Predicting Performance at Frequency *f*

The system under consideration has a small number of fixed frequencies. To predict the performance impact of running task A at frequency B, some method of projecting future processing needs must be used. Kotla, et al.[2], proposed a simple mechanism for predicting the observed instructions per cycle (IPC) at different frequencies. Existing processor hardware, such as that found in the IBM Power4, has performance counters which may be utilized by a task to frequency prediction mechanism. The counters are used to track the number of accesses made to each level of the memory hierarchy in during a full scheduling quantum. A more direct method is to count the number of cycles stalled due to outstanding memory references, but many processors, including those of the experimental platform used in this study, do not have such a counter.

To do the IPC projection, the performance model breaks the IPC into frequency-dependent and frequency-independent components. In the following, α is the IPC of a perfect machine with infinite L1 caches and no stalls. α is a task-phase specific constant that takes into account both the instruction-level parallelism of the workload and the processor resources available to extract it.

$$IPC \equiv \frac{Instructions}{Cycles} = \frac{Instr}{C_{stall} + C_{inst}}$$

$$= \frac{1}{\frac{1}{\alpha} + \frac{C_{other\_stalls}}{Instr} + \frac{1}{Instr}(N_{L2}T_{L2\_stalls} + N_{L3}T_{L3\_stalls} + N_{mem}T_{mem\_stalls}) * f}$$

Each N_x is a count of the number of occurrences of a particular type of cache or memory reference, as provided by the performance counters, each C_x is the number of processor cycles per reference and each T_x is the time consumed by each reference.

T_x is pre-determined for the particular processor. The equation assumes that the T_x values are all truly constant. In reality, this is not true and is a source of error, but in practice it does yield a reasonable approximation for the purpose of frequency and voltage scheduling. The reason that this assumption is not valid is due to differences in accessing different ranks of memory as well as the latency masking effects of pre-fetching and the effects of overlapping memory accesses. Two possible techniques have been studied to compensate for this shortcoming. The first technique, proposed in Kotla, et al.[2], requires running each task at two frequencies and then finding the solution to a linear equation. The second uses the minimum and maximum observed latencies to provide fuzzy boundaries for each prediction. Performance is predicted using both latencies. If the required frequency is different using the two latencies, the faster frequency is chosen as the desired frequency. The second technique is used in this work. Comparing the two methods is beyond the scope of this paper.

At any given frequency, the previous equation can be used to predict the IPC at another frequency given the number of misses at the various levels in the memory hierarchy as well as the actual time it takes to service a miss. This provides a mechanism for identifying the optimal frequency at which to run a given phase with minimal performance loss. As expected, the more memory-intensive a workload is, as indicated by the memory subsystem performance counters, the more feasible it is to schedule the work to lower frequency and voltage processors to save power without impacting overall performance.

$$Perf(t, f) = IPC(t, f) * f$$

$$PerfDelta(t, f, g) = \frac{Perf(t, f) - Perf(t, g)}{Perf(t, f)}$$

Rather than calculating the IPC at a given frequency ***f*****,** the *PerfDelta(f,g)* equation can be rewritten to instead solve for the ideal frequency, given the predicted performance at the maximum frequency, the performance counter data at the current frequency and a measure of the maximum performance loss the system will tolerate. The calculation of ideal frequency, f_{ideal}, is used as part of the **TFS**. It is possible to calculate *Perf(f)* at each available frequency in the system, but this alternative approach is limited to systems with fixed, known frequencies and a relatively small set of such frequencies. In the following, f_{max} is the nominal maximum frequency for the processor family and ε is a small constant indicating the amount of performance loss the system will tolerate.

$$f_{ideal} = f_{\max} \text{if } IPC >= 1$$

$$f_{ideal} = \frac{1}{\alpha}\left(\frac{instr * Perf(t, f_{\max}) * (1-\varepsilon)}{inst - T_{mem\_all} * Perf(t, f_{\max}) * (1-\varepsilon)}\right) \text{otherwise}$$

$$\text{where } T_{mem\_all} = N_{L2}T_{L2\_stalls} + N_{L3}T_{L3\_stalls} + N_{mem}T_{mem\_stalls}$$

3.3. Calculating Performance Loss

The implementation or the capacity of the system may be such that it is not possible to schedule all tasks at their desired f_{ideal}. In such situations, it is necessary to make reasonable scheduling decisions based on additional criteria. The criteria used here are the performance loss and priority of the tasks under consideration.

The predicted IPC can be translated into a more meaningful metric for calculating the performance loss of task **t** at frequency *f*. Rather than working directly with the predicted IPC, the predicted throughput at frequency *f* and f_{max} are used. Throughput is used as the metric of performance when attempting to minimize performance loss, as defined by the next two equations. Here throughput performance is the produce of IPC and frequency while performance loss the fraction of the performance lost by running at the same work at a different frequency. The idea behind the use of throughput can be seen in Figure 1. Performance saturated tasks gain nothing from the increase in frequency which will be reflected by a constant throughput value.

$$PerfLoss(t, f) = \frac{Perf(t, f_{\max}) - Perf(t, f)}{Perf(t, f_{\max})}$$

3.4. Minimizing Performance Loss

Performance loss estimates for individual tasks can be used to minimize the performance loss of the system as a whole. Any particular task may suffer a performance loss, but as long as the system incurs the least possible performance loss under the current frequency constraints the situation is acceptable. The total performance loss is the sum of the performance loss of each task scheduled at each frequency. If the possible frequency settings are **F** = $f_0, \ldots, f_m$, where $f_m <= f_{max}$, then

$$TotalPerfLoss(T,F) = \sum_{\forall t @ f_0} PerfLoss(t, f_o) + ... + \sum_{\forall t @ f_m} PerfLoss(t, f_m).$$

The minimum performance loss can be found by considering all possible combinations of tasks and frequencies. Each task is considered at each available frequency. However, this approach has a number of shortcomings. The first is that the algorithm necessary to study all possible combinations is computationally prohibitive for use in a kernel-based scheduler. This issue is address in Section 4. Another problem is that the *TotalPerfLoss* metric described above does not take into account the different priorities of tasks in the system. A high priority task may itself suffer a small performance loss, but the impact of that lost performance may be felt elsewhere. To alleviate this problem, *TotalPerfLoss* can be modified to take into account the priorities of the tasks involved:

$$TotalPerfLoss(T,F) = \sum_{\forall t @ f_o} p(t, f_0) * PerfLoss(t, f_o) + ... + \sum_{\forall t @ f_m} p(t, f_m) * PerfLoss(t, f_m).$$

4. Scheduling Algorithm

The scheduler in Section 3.4 cannot be implemented in a fixed resource, time-critical kernel scheduler. It would require the entire system partitioning to be recalculated every time some characteristic of the system changes. This would include new phase behavior of a task, new tasks entering the system and old tasks leaving the system. The original minimum-combination algorithm requires

$$O(\frac{T!}{M!(T-M)!})$$

where T is the number of tasks in the system and M is the number of possible frequencies at which those tasks may be placed.

It is possible to take into account features of the system to find less computationally intensive algorithm that will produce the same results. The discussion below uses the following terminology:

$f_{desired}$ = lowest available frequency $\geq f_{ideal}$

f_{down} = frequency one step slower than f

f_{up} = frequency one step faster than f

f_{actual} = frequency assigned by TFS

$PerfLossDown = PerfLoss(f_{down})$

4.1. Initialization Phase Algorithm

The current implementation of **TFS** is broken down into two main phases. The initialization phase uses the algorithm in Figure 3. The initialization phase requires $O(MT^2)$ time to identify the optimal task placement. The algorithm uses insertion sort to place tasks and the worst case occurs when all tasks are originally allocated to the fastest frequency. Less computationally intensive algorithms exist, but were not used here due to the generally small number of tasks in the system.

The process begins by tracking the performance counters for all the n. The initialization algorithm is invoked either during initialization or when the balancing algorithm used in the second phase fails to find a reasonable task schedule.

The first step involves calculating the ideal frequency for each task in the system as well as the maximum possible performance. Since f_{ideal} is unlikely to be available, $f_{desired}$ is found and used throughout the rest of the calculations. The performance at the $f_{desired}$ as well as the performance and performance loss at f_{down} are also calculated. As these calculations are performed, each task is inserted into a frequency bin sorted by the performance lost at f_{down}.

After $f_{desired}$ and the associated performance metrics have been calculated, it is possible to identify an optimal task schedule for the system. The task schedule is built using the same fairness principles used by the standard Linux scheduler. This leads to the requirement that each frequency bin have the same number of tasks assigned to it. Each frequency bin, beginning with the fastest frequency, is considered in turn. Each bin may be in one of three possible states: (1) the number of tasks assigned to the bin equals the number of available slots in the bin; (2) the number of tasks assigned to the bin is less than the number of available slots in the bin; (3) the number of tasks assigned to the bin is greater than the number of available slots, In case (2), tasks are stolen from f_{down} until the bin is at capacity. In this case, the tasks at f_{down} which would suffer the largest performance loss at the next lower are the tasks chosen to fill the remaining available slots at the current frequency. In case (3) **TFS** has to give away tasks until the number of tasks in the frequency bin is reduced to the capacity of the bin. Tasks which would suffer the least performance loss at f_{down} are migrated to the f_{down} bin.

A number of additional details must be filled in for all the tasks, whether or not they were moved. To improve the performance of the balancing phase, the performance and performance loss at the $f_{desired}$, f_{up}, f_{actual} and f_{down} are monitored.

```
Start with some set of tasks assignments and a running system with an
ordered set of known frequencies, $f$ in $\mathbf{F}$

for all tasks
   calculate $PerfMax$, $f_{ideal}$
      find $f_{desired}$, $f$ closest to $f_{ideal}$ from set $\mathbf{F}$
      calculate $Perf$ at $f_{down}$
      calculate $PerfLossDown$
      place into $f_{desired}$ bins (sorted by $PerfLossDown$ )

//load balance
for each $f$ bin{
   start at $f = f_m$
   for all tasks in $f$ bin {
      if tasks in $f$ bin = number of available slots (T/M)
         allocate tasks to queues
      if tasks in $f$ bin < number of slots (T/M)
         borrow tasks from $f_{down}$ until bin is full
            give priority to tasks which suffer the most performance loss
            at $f_{down}$
      if tasks in $f$ bin > number of slots (T/M)
         move tasks to $f_{down}$ until  $f$ bin is at full
           move tasks which suffer the least  performance loss at $f_{down}$
      if a task is kept in the current bin
            update $TotalPerfLoss$ for the current bin
   }
   Move to the $f_{down}$ bin
}
```

Figure 3: Initial Task Placement Algorithm.

4.2. Balancing-Phase Algorithm

The Linux 2.6.7 scheduler upon which **TFS** is based performs load balancing every 100 milliseconds. The load balancer in Linux essentially balances the number of tasks assigned to each processor. **TFS** replaces the load balancer with a more complicated algorithm that takes into account the performance loss of scheduling tasks at a frequency less than $f_{desired}$ as well as the number of tasks assigned to each processor.

The initialization phase performs many calculations that may not be necessary each time the some characteristic of the system changes. The balancing-phase algorithm is takes advantage of this behavior to minimize the number of calculations and comparisons which must be performed. In the worst case, it defaults to the initialization phase, but its average case behavior is much better. It too has $O(MT^2)$ time but, on average, many fewer tasks are moved and fewer operations are performed. The balancing-phase algorithm is based heavily on the notion of performing small local minimizations in order to keep the overall minimum small.

Once the system has had its initial task placement schedule identified, the **TFS** continues to monitor the performance counters for each task in the system. At the end of every scheduling quantum, $f_{desired}$ is recalculated. If it has not changed and the other performance characteristics such as the IPC, and number of memory references remain approximately the same, then nothing is done with the current task. If **TFS** detects a change in the task, the task is treated more carefully. The new characteristics, particularly $f_{desired}$ and the new performance loss metrics, are updated and the task is re-inserted into its new position in its current bin. Load balancing occurs more infrequently than scheduling quanta and tasks may be moved only during load balancing.

The balance-phase algorithm is presented in

Figure 4. Processing begins at the lowest frequency bin in order to percolate resource-needy tasks to faster frequencies. In each frequency bin, all needy tasks are considered. The **TFS** first checks the $f_{desired}$ bin to see if there is a task which would suffer less performance loss at the current frequency than the needy task. If such a task exists, the swap is performed, and the performance metrics are updated.

If no such task exists at $f_{desired}$, the **TFS** instead searches the f_{up} bin to identify which task would be best suited for swapping positions with the current task. In this case, the task at f_{up} which would suffer the largest performance loss that is less than the performance loss of the current task at the current frequency is chosen if there is more than one task which would like more resources. If this is not the case, then the smallest performance loss is chosen. The reason for choosing the largest performance loss that is less than the current performance loss when additional tasks need to move is to make sure that additional tasks may be able to move. If the smallest performance loss is taken, subsequent tasks may not be able to move even though they would have been able to if a different original task was selected.

If the total performance lost becomes too large, the initialization algorithm is invoked again and the system is recalibrated to the new total performance loss bounds set by the new partitioning.

```
Start with set of task assignments and running system with an
ordered set of known frequencies, $f$ in $\mathbf{F}$

//load balance
for each $f$ bin{
  start at $f = f_m$
  for all tasks in $f$ bin {
    if tasks in $f$ bin = number of available slots (T/M)
      allocate tasks to queues
    if tasks in $f$ bin < number of slots (T/M)
      borrow tasks from $f_{down}$ until bin is full
        give priority to tasks which suffer the most performance
        loss at $f_{down}$
    if tasks in $f$ bin > number of slots (T/M)
      move tasks to $f_{down}$ until $f$ bin is at full
       move tasks which suffer the least  performance loss at $f_{down}$
    if a task is kept in the current bin
        update TotalPerfLoss for the current bin
  }
  Move to the $f_{down}$ bin
}

start at the slowest bin
for all frequency bins
  for all tasks at current frequency that benefit from higher
frequency
    for all tasks at $f_{desired}${
        if (task at $f_{desired}$ wants current frequency)
        swap tasks, placing in correct order
        continue with the next task in the current bin
    }
    for all tasks at frequency one faster than current{
      if (task would suffer less performance loss at the
        current frequency than current task)
        swap tasks, placing in correct order
        continue with the next task in the current bin
    }
   move to next bin
```

Figure 4: Balancing Phase Algorithm

5. Prototype Implementation

To evaluate the **TFS** proposed in Section 4, the authors implemented a prototype version on an IBM pSeries system running Linux. The prototype runs on a single SMP. The details of the experimental environment are given in the next section.

The prototype relies on an approximation of frequency scaling and cannot actually scale voltages. The underlying hardware provides mechanisms for throttling the pipeline using by interspersing the dispatch, fetch or commit cycles with dead cycles. Since the hardware does not currently support kernel-directed frequency scaling, fetch throttling is used to mimic the effects of frequency scaling. Experimental data indicate that it provides a good approximation to frequency scaling, even though the remainder of the pipeline continues processing during fetch-throttled cycles.

Throttling can be used to cover the entire range from 0% frequency to 100% frequency. This work assumes throttling yields the same power and performance results that using different frequencies for the processors would but ignores the settling time. In other words, if f_{eff} is the effective frequency, $f_{nominal}$ is the nominal frequency, and *throttle* is the throttling percentage, expressed as a decimal, then $f_{eff} = throttle \times f_{nominal}$. Although not completely accurate, microbenchmarks indicate that this is a reasonable first approximation on the hardware used in the experimental studies.

The Power4+ processor used in the current generation of pSeries machines provides the required performance counters for cache and memory accesses. These counters are accessed and stored as part of the context switch. The **TFS** collects the performance-counter data on each context switch and updates performance metrics at the completion of each scheduling quantum. The **TFS** has associated kernel support that generates both scheduling and counter logs that provide performance and frequency information to allow for post-processing analysis. Due to the limitations of the hardware, the program does not do any voltage calculations or detailed power computations. However, the data collected is sufficient to allow one to do post-processing to determine the amount of power that would have been saved as well as the maximum power and the overall impact on system performance.

6. Evaluation Methodology

To determine the practical value of the **TFS** mechanism proposed in the previous sections, the authors ran a number of experiments. This section describes the experimental platform, the metrics used to evaluate the benchmark results and the benchmarks themselves.

6.1. Experimental Platform

The experiments described in this paper were performed on an IBM PowerPC-based pSeries P630 [14] system consisting of 4 1GHz Power4+ cores operating at a core voltage of 1.3 volts. Each core has a private 32 KB L1 instruction cache and a 64 KB L1 data cache. Two adjacent cores share a unified 1.44 MB data cache as well as a 32 MB L3. The machine has 4 GB of main memory. Using experimentation, it was determined that the nominal latency to the L1 cache is 4-5 processor cycles, the latency to the L2 cache is 12-20 cycles, the latency to the L3 is 100-150 cycles and that to memory is 350-400 cycles, and these are the values used by the scheduling implementation. Although they agree reasonably well with other reported values for the same hardware, they are, in fact, dependent on the way in which the measurement program accesses the caches and memory. The experimental platform runs Gentoo Linux with a 2.6.7 kernel with modifications to support CPU throttling. The scheduler and task structures have been modified to support **TFS**. It is very important to note that the prototype runs on a real system in which no effort has been made to reduce the number of tasks present. TFS was responsible for tracking and scheduling approximately 80 tasks when the pSeries P630 was idle. The number of tasks in the system can increase dramatically when under heavy load.

Table 1 shows the frequency set used in the prototype system. Reduced frequencies and voltages for some processors in the system lead to a lower maximum total power. The power numbers are derived from IBM's Lava power-estimation tool.

	Original	Heterogeneous			
	P0-P3	P0	P1	P2	P3
Frequency (MHz)	1000	1000	950	900	800
Power (Watts)	130	130	110	95	75
Maximum Power (Watts)	520	420			

Table 1: Frequencies and estimated power values for the heterogeneous prototype system.

ε was selected to be 0.01 for all experiments. Smaller values of ε indicate a system less tolerant of performance loss, while larger values indicate more performance loss is acceptable. An ε of 0.01 indicates that a task may lose no more than 1% of its performance when identifying the ideal frequency.

6.2. Benchmarks

This work is a preliminary study of using prediction to guide tasks to fixed frequencies while minimizing performance loss. To allow for a controlled study of the scheduling scheme, the first evaluation reported here uses with the synthetic benchmark from [2], which allows one to measure the performance variability of a program with an adjustable ratio of CPU-intensive to memory-intensive operations. The synthetic benchmark is a single-threaded program that accepts parameters that determine the ratio of memory-intensive to CPU-intensive as well as the length of phases. It currently supports two (2) phases, but the phases may be of different lengths and different memory-to-CPU intensity. It is constructed so that a miss in the L1 is highly likely to result in a memory access due to the large memory footprint. The program reports its performance in terms of throughput from its phases.

Although this synthetic benchmark provides a basic evaluation of **TFS**, it does not give a sense of the scheduler's behavior on more realistic workloads. Thus, this paper also includes results on a selected set of additional, standard benchmarks. The benchmark set includes three CPU-intensive benchmarks from SPECCPU2000, gap, gzip, and parser, as well as three memory-intensive benchmarks, equake and mcf from SPECCPU2000 and health from the Olden benchmark suite.

7. Experimental Results

The synthetic benchmark was studied initially because it provides a controllable benchmark for experimentation. The benchmark was configured in a number of different ways, each of which exposes different characteristics of the underlying **TFS**.

7.1. Stability and initialization

The first experiment performed with the synthetic benchmark consisted of running the benchmark at different *fideal*. Target *fideal* were identified offline by using the data from previous characterization experiments. The benchmark was then run in the same configuration that generated the target *fideal*. The experiment exposes stability and initialization characteristics of **TFS**. XXX indicates the stability of the TFS scheduler and highlights the problems with initialization. The frequency settings of the processors were conservatively chosen to minimize the potential for performance loss. This is reflected in the fact that in the situations where there are additional nearby frequencies, the task is sometimes assigned to those frequencies. A larger delta between available frequencies would reduce this problem, but other methods are also available, including a variant of processor affinity that is managed by **TFS**. This suggestion has not been implemented for this work, but it would enhance task-to-frequency assignment stability. The low frequency *fideal* cases illustrated a different, but more tractable problem. In these cases, any time the task spent assigned to a processor other than 800 MHz occurred during the initialization phase. Because initialization behavior can be highly variable

8. Conclusions and Future Work

TFS is a task-to-frequency scheduler for a computing environment that uses processors with the same ISA but operating at different frequencies and voltages. It can be used to keep the total power of the processors below a defined limit and minimizes average power with minimum loss in total system performance. The authors evaluated a prototype implementation of **TFS** on commercial hardware using both a synthetic, controllable benchmark as well as a set of standard benchmark programs.

Although this is a preliminary study, **TFS** shows promise as a practical way of using heterogeneous processors without incurring severe performance penalties. However, there are several important questions that remain to be explored. First, **TFS** needs to be evaluated using a wider set of benchmarks. Second, the current prototype does not react to changes in the frequency and voltage set that may occur due to changes in the available power and cooling for the system. Third, the ideas behind **TFS** have application to clusters where the different machines have processors running at different speeds, but there is currently no implementation on clusters and no evaluation of how **TFS** would behave in such an environment. Fourth, the problem of non-constant memory latencies deserves more study. With better hardware instrumentation or more careful software data collection, one can potentially extend **TFS** to adapt to changing memory latencies.

9. Acknowledgment

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCH3039004.

10. References

[1] Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Mike Kistler and Tom W. Keller, "Energy Management for Commercial Servers", Computer, volume 36, number 12, December, 2003, pages 39-48.
[2] R. Kotla, A. Devgan, S. Ghiasi, T. Keller and F. Rawson, "Characterizing the Impact of Different Memory-Intensity Levels", IEEE 7th Annual Workshop on Workload Characterization (WWC-7), October, 2004.
[3] K. Flautner and T. Mudge, "Vertigo: Automatic performance-setting for Linux", Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02), December, 2002, pages 105-116.
[4] Rakesh Kumar, Keith Farkas, Norman P. Jouppi, Partha Ranganathan, and Dean M. Tullsen, "A Multi-Core Approach to Addressing the Energy-Complexity Problem in Microprocessors", Workshop on Complexity-Effective Design, 2003.
[5] Rakesh Kumar, Keith Farkas, Norman P. Jouppi, Partha Ranganathan, and Dean M. Tullsen, "Single-IA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction", Proceedings of the 36th International Symposium on Microarchitecture, December, 2003.
[6] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, Keith I. Farkas, "Single –ISA Heterogeneneous Multi-Core Architectures for Multithreaded Workload Performance", Proceedings of the 31st International Symposium on Computer Architecture, June, 2004.
[7] Transmeta Corporation, "Transmeta LongRun Dynamic Power/Thermal Management", http://www.transmeta.com/crusoe/longrun.html.
[8] Deva Bodas, "New Server Power-Management Technologies Address Power and Cooling Challenges", Technology@Intel, http://www.intel.com/update/contents/sv09031.htm.
[9] S. Ghiasi and D. Grunwald, "Aide de Camp: Asymmetric Dual Core Design for Power and Energy Reduction", Technical Report CU-CS-964-03, Department of Computer Science, University of Colorado, Boulder, May, 2003.
[10] S. Ghiasi and D. Grunwald, "Thermal Management with Asymmetric Dual Core Designs", Technical Report CU-CS-965-03, Department of Computer Science, University of Colorado, Boulder, May, September, 2003.
[11] S. Ghiasi, "Aide de Camp: Asymmetric Multi-Core Design for Dynamic Thermal Management", Ph. D. thesis, Department of Computer Science, University of Colorado, Boulder, July, 2004.
[12] P. Stanley-Marbell, M. Hsiao and U.Kremer, "A Hardware Architecture for Dynamic Performance and Energy Adaptation", *Power-Aware Computer Systems*, Lecture Notes in Computer Science 2325, Springer Verlag, 2002.
[13] Intel Corporation, "Intel Pentium M: Enhanced SpeedStep Technology", http://developer.intel.com.
[14] Guilian Anselmi, Derrick Daines, Stephen Lutz, Marcelo Okano, Wolfgang Seiwald, Dave Williams and Scott Vetter, pSeries 630 Models 6C4 and 6E4 Technical Overview and Introduction, IBM Corporation, December, 2003.
[15] E.N. Elnozahy, M. Kistler, and R. Rajamony, "Energy Conservation Policies for Web Servers", Proceedings of the 4th Annual Usenix Symposium on Internet Technologies and Systems, Usenix Association, 2003.
[16] Timothy Sherwood, Erez Perelman, Greg Hamerly and Brad Calder, "Automatically Characterizing Large Scale Program Behavior", Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X) , October, 2002,pages 45-57.
[17] Ashutosh S. Dhodapkar and James E. Smith, "Comparing Program Phase Detection Techniques", 36th Annual International Symposium on Microarchitecture (Micro-36),December, 2003.
[18] Allan Snavely and Larry Carter, ``Symbiotic Taskscheduling on the Tera MTA", Workshop on
Multi-Threaded Execution, Architecture and Compilation (MTEAC'00), January, 2000.