Towards a power-aware application level scheduler for a multithreaded runtime environment

Alan S. de Araujo, Cícero Augusto de S. Camargo, Gerson Geraldo H. Cavalcheiro, Maurício L. Pilla
Federal University of Pelotas, Brazil
{asaraujo, cascamargo, gerson.cavalheiro, pilla}@inf.ufpel.edu.br

Abstract

At the same time that the modern society becomes more dependent on computing power, people become more concerned about the environment and, in consequence, about energy consumption. In the high performance computing field, most works only take into account performance aspects such as throughputs to measure schedulers. In this paper, we introduce and evaluate an energy-aware list scheduler that uses heuristics based on the critical path to determine processor affinity and the clock rate of each core. We have observed that it is possible to implement an execution support able to offer acceptable performance at same time that provides a strategy to save energy. Two case studies discussed in the paper support this conclusion.

1 Introduction

The advent of multicore architectures in the mainstream market made parallel processing widely available, even becoming a necessity rather than an option. The increasingly costs of developing faster processors that could exploit parallelism in the instruction level made the industry change its focus to replicating the same design many times in a chip. Together with this increase in the opportunities for exploring thread-level and process-level parallelism, power dissipation and energy consumption issues required new techniques to be developed for these processors. It is now possible to change clock rates to match energy restraints and processing necessities.

Development environments for parallel processing are still an open research topic. Currently, there are no efficient tools to develop general-purpose programs that allow programmers to easily exploit thread-level parallelism. Libraries and tools can be too low-level (such as Pthreads [7]) or be high-level but not map well all the kinds of programs to parallel implementations (such as OpenMP [3]). The manycore systems, a tendency in the processor industry, make it harder to deal with the work division and synchronization required for such systems as the number of cores increases.

One of the main hindrances to program for these systems is that many applications cannot be divided into homogeneous loads. A popular model of heterogeneous tasks is to represent the tasks and their dependencies as a directed acyclic graph (DAG). This model is assumed throughout the current paper. Cilk [1] and Anahy [2] are parallel programming tools from research groups that exploit this program representation model associated with a scheduling runtime that allows for efficient exploitation of hardware parallelism.

Currently, parallel programming tools are focused on extracting the maximum performance possible, and do not take advantage of the available hardware mechanisms to reduce the expended energy. In this paper, we study ways to improve energy-awareness on the Anahy runtime environment. An application-level strategy that reduces required energy and power dissipation without significative performance loss was developed.

This paper is divided in the following sections. In Section 2, techniques such as processor affinity and frequency management are presented. Section 3 presents the Anahy multithreaded programming environment. Then, Section 4 discusses how Anahy was modified in order to become energy-aware. Section 5 presents two case studies. Finally, Section 6 presents final remarks and future work.


Table 1. Execution times for Producer/Consumer test.

<table>
<thead>
<tr>
<th></th>
<th>5 threads</th>
<th>15 threads</th>
<th>25 threads</th>
<th>35 threads</th>
<th>45 threads</th>
<th>55 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular execution</td>
<td>0.5684s</td>
<td>1.5930s</td>
<td>2.7463s</td>
<td>3.8659s</td>
<td>4.9766s</td>
<td>5.8741s</td>
</tr>
<tr>
<td>Standard deviation</td>
<td>0.0284s</td>
<td>0.0881s</td>
<td>0.1740s</td>
<td>0.1963s</td>
<td>0.0133s</td>
<td>0.3167s</td>
</tr>
<tr>
<td>Execution exploiting affinity</td>
<td>0.4702s</td>
<td>1.3718s</td>
<td>2.2753s</td>
<td>3.1799s</td>
<td>4.0793s</td>
<td>4.9664s</td>
</tr>
<tr>
<td>Standard deviation</td>
<td>0.0051s</td>
<td>0.0055s</td>
<td>0.0075s</td>
<td>0.0133s</td>
<td>0.0160s</td>
<td>0.0091s</td>
</tr>
<tr>
<td>Performance</td>
<td>17.27%</td>
<td>16.12%</td>
<td>8.83%</td>
<td>21.57%</td>
<td>22.00%</td>
<td>18.27%</td>
</tr>
</tbody>
</table>

2 Related Techniques

This work is developed in the application-level scheduling [5]. In such level, information related to a program in execution are made available to the scheduler, and thus they may be used to decide which threads should be scheduled next. In order to reduce the required energy to run an application, we have chosen to explore processor affinity and frequency management, which are described in the next subsections.

2.1 Processor Affinity

Processor Affinity is the capability of associating a given thread to a specific processor. Instead of running in the first processor available, threads that have set affinity with a processor will be scheduled only on that specific resource. Processor affinity allows for more efficient use of cache, as less misses will occur when the thread runs in the same processor. As the memory hierarchy may be comprised of more than two levels of cache, setting threads that exchange much data in processors that share the highest level of cache possible may make them more efficient.

In this work, the classic producer/consumer model was implemented in order to study the effects of processor affinity in the scheduling strategy. A group of threads named producers generate items that are consumed by another group of threads, the consumers. The communication among producers and consumers is implemented through a shared buffer. In a system composed of m cores, where RP is the set of available cores such as $RP = \{P_0, P_1, P_2, \ldots, P_{m-1}\}$. The scheduling applied to the program determines that all producers are scheduled in $P_{m-1}$, while the other cores execute the consumer threads. As the process of producing an item takes more time than consuming it, cores running consumer threads may have their clock rates slowed to reduce energy consumption.

The importance of using processor affinity can be illustrated by Table 1. It presents the average execution times and standard deviation for the producer/consumer test program, with in a regular execution and also in an execution considering that all producer threads are executing in a dedicated processor and the consumers in the remaining. The dedicated to the producers runs at 100% of its capacity while the others have the speed reduced to 66% of the maximum. For each configuration, the test was run 30 times. The computer used in the experiments featured an Intel®Core™2 Quad, 2.66GHz, with 64 KB first-level caches, 4 MB second level caches, and 4 GB of main memory. Results with processor affinity are 22% better than those without affinity, in the best case. Besides, standard deviations are also smaller, thus producing a more predictable performance.

2.2 Frequency Management

The relationship between energy consumption and clock rate is determined by Equation 1 [8], where $C_{DD}$ represented the commutated capacitance, i.e., the amount of electric energy than can be stored in an equivalent capacitor, $V_{DD}$ is the operation voltage, and $f$ is the clock rate. $C_{DD}$ is a fixed value for each circuit, but $V_{DD}$ and $f$ can be modified to certain limits. Increasing $V_{DD}$ is required to increase $f$ or the system becomes unstable. Frequency management techniques work in these last two variables to achieve power savings.

$$Power = C_{DD} \times V_{DD}^2 \times f$$  \hspace{1cm} (1)$$

The control of clock rate for each core present in modern processors allows the efficient management of workload for multithreaded programs. Threads that require more processing power may be executed in a core with the maximum clock rate available, while threads that execute less demanding tasks may run on cores with reduced clock rates. The larger the clock rate, the more energy will be consumed. Hence, proper frequency management allows to save energy and reduce heat to be dissipated.
3 The Anahy Environment

Anahy is a programming environment for multi-threaded applications that provides a dynamic scheduler. Its programming interface is simple, implementing functions for the creation and synchronization of threads with a syntax similar to POSIX threads [7]. The main primitives to manipulate threads are `create` and `join`. Both primitives can be executed over more than one thread. This interface allows the description of programs as Directed Cyclic Graphs (DCGs). Each node is a task, while the edges represent dependencies. Further details may be found in [2].

The runtime execution core of Anahy is composed by virtual processors (VPs), implemented using system threads. Therefore, the scheduling is done in two levels: in the first level, VPs are scheduled to real processors; in the second level, the runtime core schedules ready tasks to VPs. In the later, application-level scheduling, the runtime may use information about the program to heuristically choose from the set of ready tasks.

3.1 Execution architecture

Figure 1 presents the architecture model that supports the execution of Anahy programs. In this model, $N$ user-level threads are mapped to $M$ system-level threads, in a many-to-many model. The execution support is provided by a shared multiprocessor architecture (SMP), where a common addressing space is shared among all processors.

3.2 Design of the Executive Core

Anahy’s runtime core implements a scheduling strategy that keeps the description of relationships among threads as a DCG. This graph abstracts the DAG that describes the data flow among tasks defined by the application. A task is comprised of a code segment inside a thread between the `create` and `join` primitives.

Each thread $\Gamma_i$ represents a sequency of tasks $\Gamma_i = \{\tau_{i,1}, \tau_{i,2}, \ldots, \tau_{i,n}\}$. Hence, a task is a unity of work described by an instruction sequence that can be executed in a finite time. The end of a task $\tau_{i,k}$ is determined by the end of the thread $\Gamma_i$ or the invocation of a primitive that changes the DCG, i.e., in the creation of a new thread or the synchronization with the end of another thread. In the last cases, a task $\tau_{i,k+1}$ will be created.

The runtime presents five lists of threads: `ready`, with threads that are waiting for execution; `executing`, with threads that were scheduled to VPs; `finished`, `suspended`, and `blocked`. The first three lists are shared among all VPs, while the last two are local.

For example, consider a uniprocessor architecture and an initial state where the DCG contains a single node representing $\Gamma_1$, the list of ready threads containing only $\Gamma_1$. Then, the scheduler removes $\Gamma_1$ from the ready list and inserts it in the executing list. The task $\tau_{1,1}$ is mapped to the $VP_1$. When a `create` is executed, a new thread $\Gamma_2$ is generated and inserted in the DCG. The corresponding thread is also stored in the list of ready threads. $VP_1$ keeps executing $\Gamma_1$, now with task $\tau_{1,2}$. In the sequence, a synchronization operation of $\Gamma_1$ to $\Gamma_2$ implies that $\tau_{1,2}$ ends its execution and a new task $\tau_{1,3}$ is generated. As $\Gamma_1$ has called a `join` primitive, it becomes part of the suspended list, but without leaving the executing list. After this, the thread $\Gamma_2$ can be transferred from the ready list to the executing list, being assigned to $VP_1$. Then, the task $\tau_{2,1}$ starts execution. This procedure is repeated in a recursive way and, in the end of the execution of $\tau_{2,n}$, the processor returns to execution of $\Gamma_1$.

In the case of multiprocessor architectures, two or more VPs execute the same algorithm described previously described for a uniprocessor architecture. Thus, when a thread $\Gamma_i$ executes a task $\tau_{i,k}$ that requires synchronization with a thread $\Gamma_j$, two new situations may occur: (i) $\Gamma_j$ is already in the executing list; or (ii) $\Gamma_j$ is in the finished list. In the first case, $\Gamma_j$ cannot continue execution as dependencies of $\tau_{i,k+1}$ are not full-

![Figure 1. Layered model of the Athreads environment.](attachment:figure1.png)
filed, and then it is suspended. When \( \tau_{j,n} \) is finished, then \( \Gamma_j \) is sent to the finished list, and \( \Gamma_i \) may then execute. In the second case, data produced by \( \Gamma_j \) is recovered and \( \Gamma_i \) may immediately continue execution.

4 Implementation of the Heuristic

The proposed scheduling strategy is implemented in the application level to take advantage of the program structure to improve system-level scheduling. It is applied during runtime, monitoring the evolution of the DCG that describes the program in execution. Load is distributed among processors guided by the computational costs, and at the same time the clock rate of each core is set according to the computational cost of the threads that are running on it. In this Section, performance results of test cases are presented for the Anahy's energy-aware scheduler. In the first case, the execution cost of each thread is provided by the programmer. In the second case, the execution cost is inferred from the generated DCG, taking the critical path as the sequence of threads with larger computational cost.

4.1 Scheduling mechanism

The scheduling considers the existence of a set \( P = \{P_1, P_2, \ldots, P_p\} \) composed by \( p \) virtual processors (VPs). Processors in \( P \) are scheduled over an architecture \( M = \{M_1, M_2, \ldots, M_m\} \) composed of \( m \) real processors \((p \leq m)\). The scheduling unit is the thread, and \( n \) threads are created by the program during execution. These threads are described by a DCG \( C \), dynamically created and manipulated.

The scheduling operations are as follows:

- \( \text{void setCost(Thread thid, int cost)} \): declares the cost of execution of thread \( \text{thid} \).
- \( \text{void bind(Thread thid, ProcVirt p)} \): defines that thread \( \text{thid} \) will be executed only by processor \( \text{p} \).
- \( \text{Thread getThread(ProcVirt p)} \): this function returns a thread in \( C \) that is ready to execute, following the algorithm of Section 3.2. Threads bound to VP \( \text{p} \) are given higher priority.
- \( \text{void setAffinity(ProcVirt p, ProcReal \*m, int size)} \): defines that threads in VP \( \text{p} \) can only be executed by the \( \text{size} \) real processors in the list \( \text{m} \).
- \( \text{void setFrequency(ProcReal m, int freq)} \): defines the clock rate for processor \( \text{m} \).

Only the \( \text{setCost} \) is made available for the programmers. The remaining services are used internally by the scheduler. The scheduling of threads in the VP to real processors is done by the operating system.

As described before, Anahy’s scheduler may consider the estimated computational cost as given by the programmer or following the critical path of the DCG \( C \) that describes the application. The scheduling solution for the first case is trivial to implement, but requires precise knowledge of the costs of each task. On the other hand, predicting the largest computational load on a DCG requires some heuristic to determine the critical path of the program. This critical path limits the performance [6] and is used by many list scheduling algorithms. Supposing that the threads in the critical path will be executed in a single processor, it is possible to say that, with enough processors, the computational cost attributed to any processor will not exceed the load of the processor responsible for the execution path.

From these premisses, the scheduling strategy uses processor affinity to bind virtual processors to real processors and determines the clock rate for each processor based on its attributed load. These operations incur in costs that may negatively affect performance of the system. First, voltage must be set accordingly to the desired clock rate. Then, clock rate is adjusted. In the meanwhile, the processor is halted and waits for the changes to take place [9]. Equation 1 simplifies the expression of energy consumption to reduce the costs associated with decisions.

\[
\text{Consumption} = \sum_{i=1}^{m} (f_i \times \alpha_i) \times \Delta t \quad (2)
\]

This equation uses information obtained from the operating systems: \( f_i \) is the clock rate of each core, and \( \alpha_i \) is the usage percent of each core. The sum of these variables multiplied by time \((\Delta t)\) defines an estimation of energy consumption. This method is not precise, but it has the advantage of using the available information without requiring additional hardware.

5 Results

The computer used in the experiments featured an Intel® Core™ 2 Quad, 2.66GHz, with 64 KB first-level caches, 4 MB second level caches, and 4 GB of main memory, the same one used for the experiments in Table 1. Two case studies were developed, one using annotated loads to decide the critical path, the other one dynamically finding the critical path in the DCG describing the application.
5.1 Scheduling Tasks with Annotated Loads

For the first case study, the Josephus problem [4] is considered. This problem consists in generating a circular queue of tasks with random loads. For the experiment, 60% of tasks were created such as to present a load less than or equal to 45% of the total load. Hence, loads were classified as light or heavy. After that, a thread is spawned for each element in the queue. Clock rates were dynamically adjusted so that cores with light loads would run at 52% of the maximum frequency. The graph in Figure 2 presents the energy consumption monitoring for the three instances of the problem: without application-level scheduling and maximum clock rate, with energy-aware application-level scheduling, and without application-level scheduling and minimum clock rate. The horizontal axis presents time, and the vertical axis shows energy consumption. The energy-aware scheduler was 1.4× more efficient in energy consumption than running with maximum clock rate. The application-level scheduler was also 1.53× more efficient than running with minimum clock rate. As for execution times, it can be observed that our strategy took an average of 54.3 s to execute the problem, while the average time with maximum clock rates was 50.1 s. This corresponds to a loss of 8.3% in performance, which is acceptable given the gain in energy consumption.

5.2 Critical Path Heuristic

The second case study explores the critical path in a DCG as the scheduling heuristic. The recursive Fibonacci pseudocode shown in Figure 3 was implemented in a parallel way, with each call to Fibo generating a new thread. This implementation has two interesting characteristics. First, it exposes the scheduler to a highly parallel application. Second, it produces an unbalanced graph, with heavier load on its left part, representing the threads that calculate the Fibonacci of \( n-1 \). In the implementation, Fibonacci of less than 26 is calculated in a sequential way.

```c
int Fibo( int n ) {
    if( n >= 2 ) return 1;
    else return Fibo(n-1)+Fibo(n-2);
}
```

Figure 3. A recursive Fibonacci pseudocode.

In the scheduling of this application, one of the four cores was dedicated to the execution with higher priority of threads in the left-most part of the graph, where the critical path can be found. The other cores had their clock rates reduced to 66% of the maximum. Figure 4 shows the speedup for the parallel versions of Fibonacci for 40 and 45. The graph shows the average of 20 executions for each configuration, with and without affinity support. Several runs were made varying the number of virtual processors. The sequential version was executed over a single virtual processor and took 3.86 s for the Fibonacci of 40 and 32.63 s for the Fibonacci of 45. The results show that it is possible to combine energy management and performance in a single scheduling heuristic. In these cases, performance was achieved through the knowledge of the regular structure of the graph for the program. In particular, it can be observed that the execution with the extended runtime scheduler obtained better performance than the regular one. We attribute this to the better exploitation of locality in the execution of threads. Accesses to the DCG were monitored and less synchronization is required among application threads executing on different virtual processors.

Figure 5 illustrates the energy consumption of a Fibonacci execution. This figure presents the energy consumption trace for the parallel Fibonacci of 43 considering both regular and extended scheduler. The X axis presents the time in seconds from the program start, while the Y axis shows the estimated energy consumption for an interval of a second. Notice that monitoring the energy consumption introduces a high degree of overhead and degrades the performance. The higher peaks of energy consumption in the results without affinity nor frequency control makes control of power dissipation harder.
6 Conclusions

This paper presented an implementation of the Anahy runtime scheduler that is energy-aware, i.e., tries to maximize performance while minimizing the spent energy, using an application-level scheduling strategy that takes into account the critical path of the DAG for an application. The Anahy runtime implements a scheduling strategy based on a critical path list strategy. A heuristic to reduce energy consumption was introduced in this basic strategy, assuming that threads that do not belong to the critical path can be executed by processors with lower clock rates in order to reduce to total amount of energy spent without much performance loss.

For an implementation in the Anahy’s runtime, we discussed performance of two case studies. We have exploited a service furnished by the operating system to set affinity of threads to processors and also dynamically controlled the speed of cores according to estimated loads. In the case studies, we have observed that reducing spent energy is not necessarily contradictory with the desire of good performance (execution time, in this case). In particular, we highlight the results obtained in the second case study: the Fibonacci application.

The recursive implementation of Fibonacci generates, at execution time, a graph with a regular structure. Since this structure can be predicted, a list scheduling, such as the one implemented by the Anahy runtime, may estimate where is the most probable region in the graph to find threads in the critical path. Such kind of application is the most suitable to the proposed approach given that some knowledge about the application behavior must to be taken into account to identify the threads that require more or less processing power.

In future work, we intend to study the impact of using the extended runtime kernel of Anahy in other multithreaded programs. The future work includes also the development of a schema to detect and to adapt dynamically the frequency of the cores in function of the deployment graph.

References