Energy Efficiency via the N-way Model

Romain Cledat, Santosh Pande

Energy Efficiency via the N-way Model

Romain Cledat
Georgia Institute of Technology
romain@cc.gatech.edu

Santosh Pande
Georgia Institute of Technology
santosh@cc.gatech.edu

Abstract

With core counts as well as heterogeneity on the rise, the sequential components of applications are becoming the major bottleneck in performance scaling as predicted by Amdahl's law. We are therefore faced with the simultaneous problems of occupying an increasing number of cores and improving sequential performance. In this work, we specifically focus on improving the *energy efficiency* of sequential algorithms through the *n-way* programming model.

In previous work, we introduced the *n-way* programming model which seeks to exploit the *algorithmic diversity* present in certain computations in order to speed-up or improve the quality-of-result. The core idea behind n-way parallelism is to launch a number of instances of a key computational step and benefit from either the algorithmic diversity present in the algorithm or the diversity in algorithms available to express the computation.

In this paper, we propose to combine metrics measuring the algorithmic progress of a computation with metrics measuring the energy expenditure to compute an efficiency metric. This metric can then be used to quickly pick which instance of a n-way computation is the most energy efficient and cull the inefficient ones.

The evaluation of our idea on sorting benchmarks shows that our technique is promising.

1. Introduction

Today, chipmakers are utilizing the increased number of transistors predicted by Moore's law to increase the number of cores, thereby increasing hardware parallelism. While parallel parts of applications are able to benefit from the increase in hardware parallelism, sequential portions are increasingly becoming a bottleneck as their performance will not improve with each new generation of processors [5].

Furthermore, heterogeneity is also becoming prevalent in processor design. The Cell processor [3, 6] is a prime example of a heterogeneous chips where many leaner smaller cores (the SPUs) are used as accelerators to a larger all-purpose processor (the PPU). GPUs are also being cast on the same die as CPUs such as in Intel's Westmere processor [9] and AMD's Fusion [1]. Similarly, Intel's Tolapai [8] integrates a network processor with the main processing core. For sequential codes, heterogeneity due to asymmetric multicores is more valuable as GPUs are designed to deal with massively parallel codes. For a given sequential code, heterogeneity therefore adds a choice of target but also complexity as choosing the most appropriate target is difficult and the subject of active research.

The increase in the number of cores makes it hard for the programmer to fully utilize the resources available in an efficient manner as he must now execute code on multiple parallel resources. This is particularly true for applications that are mostly sequential. Therefore, an important research question is how to make sequential code benefit from this increase in parallel resources (either homogeneous or heterogeneous). Previous work on homogeneous resources has focused on utilizing helper threads [24, 29] to assist a main sequential thread in its execution (by prefetching data for example). Other work in thread-level speculation [21] also exploits parallel homogeneous resources to speed-up a sequential thread of execution. For heterogeneous platforms, workload characterization can be used [28] to determine statically which target would be the most appropriate.

In this work, we seek to show that additional parallel resources, if they are unused, can be utilized to dynamically find the best *match* between the algorithm, the input data and the processing resource where "best" is defined as the most energy efficient.

Energy is indeed becoming a major concern from high-end data-centers down to mobile devices. Datacenters are trying to cut costs and mobile devices are concerned with battery life. Energy is such a concern that a separate list, Green 500, is used to rank supercomputers by their energy efficiency.

1.1 The matching problem

Finding the least energy consuming match between an algorithm, the input data passed to it and the processing resource it runs on is a recurring problem. Given a problem P and an input I, we argue that two dimensions can be explored in order to determine the most energy efficient combination:

- **Hardware diversity**: the same code may execute more or less efficiently on different computing resources. Differences in micro-architecture for example could lead to different costs for pipeline stalls, different branch misprediction rates, etc. Changes in cache configuration will also lead to changes in the number of accesses to main memory thereby increasing or decreasing the overall energy cost of running the code. More fundamentally, significant design differences, such as those between the SPUs and the PPU in a Cell processor, or small and large cores in asymmetric multicores, can lead to large differences in performance and energy consumption.

- **Algorithm diversity**: for the same problem P, there may be multiple ways of solving it. We refer to this as algorithmic diversity and detail it in Section 2. Intuitively, consider a randomized algorithm A used to solve the problem P. Given the element of randomness in the algorithm, its exact execution will differ from run to run thereby, in effect, impacting the platform differently and potentially using different amounts of energy.

1.2 The N-way model

In this paper, we will build on ideas introduced in [2] and briefly present the n-way programming model which allows a *dynamic selection* of *which* algorithm to run *where*. The model therefore exploits both the hardware diversity present in a platform as well as the algorithm diversity. The goal is to provide the programmer with a model allowing him to dynamically find the least energy consuming match of algorithm and processing resource for a given problem and input data. Note that our model does not concern itself with the task of compiling the code to run on heterogeneous platforms, rather we envision our model on top of lower level programming models such as OpenCL [11] which provide a unified programming language for heterogeneous platforms such as CPUs, GPUs and other accelerator-like devices.

The core idea of the n-way programming model is to exploit algorithmic and hardware diversity by simultaneously running in isolation multiple *ways* to solve the same problem $P(I)$. A monitoring runtime can then quickly choose the "way" that will, overall, use the smallest amount of energy, let it proceed and terminate all others. Our model can therefore be qualified as "run for a little while and choose"; it eliminates the need to statically select a match between algorithm, input and processing resource.

To do so, we introduce a measure, the *algorithmic energy efficiency* which is loosely defined as the ratio of the amount of progress the algorithm has made over the energy expended to obtain that progress. Section 3 presents this measure in more detail gives more detail.

Note that our approach is counter-intuitive in the sense that while all the ways are running in parallel, we will actually be wasting energy as multiple ways to solve the same problem will be simultaneously executed. However, we believe that if a decision can be made early enough, overall energy savings will be achieved, in particular if the diversity present is large.

1.3 Contributions

We make the following contributions in this paper:

- We recognize the existence of algorithmic diversity in many important computational steps.
- We introduce an *algorithmic energy efficiency* metric based on measuring *algorithmic progress* and performance counters. We show how this metric can be used to compare algorithms based on their energy efficiency.
- We show that algorithmic and hardware diversity can be exploited through the n-way model, which we present, to select the most energy efficient algorithm thereby improving sequential code performance with the help of multicore platforms.
- We demonstrate the feasibility of our idea with well-known implementations of sorting algorithms showing how we can effectively predict the most efficient implementation very quickly.

Section 2 reviews past uses of hardware diversity and defines and illustrates *algorithmic diversity*. Section 3 describes the rationale for the algorithmic energy efficiency (AEE) measure we introduce and Section 4 shows how both diversity and the AEE measure can be combined to reduce the energy consumption of sequential code. We present motivational results in Section 5 before reviewing related work in Section 6 and concluding in Section 7 with future work.

2. Diversity

In this Section, we review past uses of hardware diversity and introduce the notion of algorithmic diversity.

2.1 Hardware diversity

In the past, when hardware diversity was only present across machine (and not within the same one), architectures could be custom matched for a specific set of algorithms; FPGAs and ASICs are extreme cases of this where the architecture is optimally tuned for an algorithm (for example to utilize the least amount of power). This customization is also present, to a lesser extent, in the general computing arena where the introduction of vector-friendly SSE instructions improved the performance and energy consumption of certain types of algorithms. In all these cases, the architecture was *statically* matched and optimized for specific algorithms.

More recently, work has been done to try to dynamically match algorithms to specific computing resources in heterogeneous systems. In [13], Kumar et al. shows how processor power dissipation can be reduced in a single-ISA heterogeneous system. They show how an oracle scheduler could reduce power consumption significantly. In [23], Snowdon et al. show how a pre-characterized model can be used to predict, at runtime, the energy consumption of a piece of code. They use this information to dynamically scale the frequency of the processor to provide the requested performance to energy trade-off. Luk et al, in [16], describe a system to dynamically map tasks to either the GPU or the CPU based on how well they will perform on each. They use a learning method based on training data as well as past executions.

All of this previous work shows the importance of finding the best match between architecture and algorithm. It also shows that this is not straightforward and most of the work relies on some static or learned knowledge about the applications running. As hardware diversity becomes more common and applications able to exploit this diversity more wide-spread, this approach will be difficult to continue and a more dynamic approach may be required. Furthermore, for algorithms that are heavily input data dependent, the approaches in previous works will not be sufficient as they fail to consider the input data to the code in their algorithms. In this work, we propose an alternative solution to matching the hardware to the algorithm by evaluating algorithms on multiple cores simultaneously for a short period of time and making a rapid decision about the one that is the best.

2.2 Algorithmic diversity

We define a *diverse problem* as follows:

Definition Let P be a problem. P is called a *diverse problem* if P can be executed in multiple "ways" for a given input I where each way will produce an solution to P.

The different ways may solve P slightly differently, some operating faster than others or with a higher quality of result. Furthermore, each way will utilize more or less energy depending on its impact on the hardware (number of branch mispredictions, cache misses, or other measurable event). Note that the granularity of P is irrelevant in the definition. P could be an entire application or a small basic building-block kernel such as a "sort" problem. Finding diversity at the granularity of a kernel will allow larger problems that depend on the kernel to also benefit from the diversity present in the kernel.

Unpredictable effects of diversity To focus solely on algorithmic diversity, we consider that all ways are run on the same type of computing resource. Suppose W_1 through W_n are all statically known ways to solve P. Ideally, an oracle would be available to determine, for a given input I, which way will use the least amount of energy and the programmer could elect to run only that way. However, in the real world, given the high data-dependence of most algorithms, such an oracle would most likely end up being just as computationally intensive as the ways themselves and therefore serve no purpose.

The effects of diversity are therefore potentially highly unpredictable. In other words, dynamically, when the input to P is known, it is not always possible to determine which of the different ways will use the smallest amount of energy.

2.2.1 Prevalence of diversity

Algorithmic diversity can potentially be present in three different forms: **i)** across multiple distinct algorithms to solve the same problem, **ii)** within an algorithm that can be parametrized, and **iii)** with an algorithm that exploits randomness.

Compilers may also introduce a degree of diversity by compiling the same code with different compiler options or even with different compilers which may perform different types of optimizations. The effect of the different flags does not always have a predictable effect [18] however, their relative behavior may not vary dynamically with input data. For example, a code compiled with the "-O2" flag will most likely always execute faster than one executed with the "-O0" flag. The unpredictable effect of diversity is therefore potentially lost with compiler-induced diversity.

Diversity across algorithms For many real-world problems, finding an exact solution in a reasonable amount of time is impossible. NP-hard and NP-complete problems are prime examples of this but even problems for which a polynomial-time algorithm exists may be difficult to efficiently solve in practice due to the problem's large size. For such problems and others, the user is more interested in an *acceptable* solution rather than a specific one or all the solutions. For example, when finding a path between two nodes in a graph, any path is an acceptable solution although some may be *preferred* over others (shorter paths for example). Acceptance of a wider set of solutions enables the use of a variety of algorithms to solve the same problem. Approximation algorithms [27] and heuristic algorithms have been utilized for just such a purpose.

Diversity in parametrized algorithms Apart from distinct algorithms, even if only a single algorithm exists to solve P, diversity may still be present if the algorithm can be parametrized. For example, the CPLEX solver includes over 100 parameters [26] that can be used to tune the effects of the algorithm. Each pair (A, Param) can be considered to be a distinct algorithm and we can therefore map this case to the previous one.

Diversity due to randomness Randomized algorithms [17, 19] utilize a degree of randomness as part of their logic. A randomized algorithm seeks to achieve good performance on the average case. Due to its random nature however, its exact behavior on a given hardware is impossible to predict. For example, memory access patterns may be different from one run of the algorithm to the next thereby impacting the number of misses in the cache and the overall energy consumption.

Therefore, similarly to parametrized algorithms, randomized algorithms also inherently contain diversity. One can view the random seed supplied to the algorithm as a form of implicit parameter passed to the algorithm. The random choices made by these algorithms confer to them a diversity in exact execution path. However, the parameter space is innumerable (for all practical purposes).

2.2.2 Hardware diversity: a multiplier effect

For a given problem P with multiple ways to execute W_1 to W_n, hardware diversity causes a multiplicative effect in the number of possibilities to execute P as each way can potentially be executed on different cores. Instead of having just W_1 to W_n as ways to execute P, we now have the pairs (W_1, C_1), (W_1, C_2), $\dots$, (W_1, C_m), $\dots$, (W_n, C_m) where C_1 to C_m are the different cores available on the platform ("smaller" cores, etc.).

3. Algorithmic energy efficiency

In Section 2, we demonstrated how for a given problem P on input I, different ways (W_1, C_1) to (W_n, C_m) could exist to solve it and that it is generally difficult to statically determine which of the ways would consume the smallest amount of energy to solve P for the input I. Trivially, we could of course run each (W_i, C_j) and measure its total energy consumption after it terminates. However, for our purposes, we require a solution that is *online* and *very fast*.

Measuring only the energy consumed by (W_i, C_j) is not enough because even though (W_i, C_j) may be consuming very little energy, it may do so for a very long time, thereby driving up its total energy consumption. We instead propose to measure **i)** an algorithmic property (*progress*) and **ii)** a hardware impact measure (here energy). The combination of the two allows us to define an *algorithmic energy efficiency* which measures how much algorithmic progress the way makes for the amount of energy expended.

3.1 Required measures

We define the algorithmic energy efficiency *eff* as $\mathit{eff} = \frac{\mathrm{prog}(A)}{\mathrm{E}(A)}$ where prog is a measure of progress and E a measure of energy and we describe both of these terms in the following sections.

3.1.1 Algorithmic progress

We define *algorithmic progress* as the amount of work that has been done towards solving the problem. For example, in a sorting algorithm, progress could be measured as the number of inversions that have been removed by the algorithm (an inversion exists between any two elements that are out of order with respect to the sorted order) or as the number of elements "moved". Note that we define progress in terms of the problem to be solved, not in terms of low-level operations used by the algorithm. This is crucial as it allows different algorithms implementing the same problem to be compared.

Problems where greedy constructive algorithms are used are very good examples of problems where it is easy to define a notion of progress as both the current amount of work done and the final amount required to solve the problem are well defined.

Many other problems (in particular optimization problems) also work by finding partial solutions which are subsets of the final solution. Problems solved using dynamic programming, incremental algorithms (such as Dijkstra's single source, single destination shortest path algorithm) are examples of this. For such cases, the progress can be quantified in terms of the size of the partial solution found.

Progress is much harder to define for other problems where algorithms such as back-tracking are used as it is not clear exactly what constitutes progress (since the algorithm may back-track if it comes to a dead-end). In other words, even if the algorithm is "progressing", it may have to wipe-out some of this progress making the comparison of progress metrics unreliable. However, even in those situations, a certain notion of progress can be defined. For example, consider the traditional branch-and-bound SAT solver. While progress cannot be defined as the number of clauses that are satisfied (as they may become unsatisfied in the future), it can be defined as the amount of solution-space that has been pruned. This does not completely solve the issue as the amount of progress to be made to solve the problem is also unclear. Considering again the SAT solver problem, an algorithm that has explored 90% of the space is indeed more likely to find a solution than one that only explored 5% but there is no guarantee as the second algorithm may get "lucky" and find a satisfying solution without needing to explore the entire space. The *rate* of progress can however still be compared and this more approximate measure can also be used as a substitute for an exact progress measurement.

Formal definition of progress Formally, we define a progress function $\mathrm{prog}\,(A)$ on an algorithm A as:

- A non-decreasing monotonic function
- If $\mathrm{prog}\,(A_1) > \mathrm{prog}\,(A_2)$ then A_1 has made more progress than A_2 towards finding the solution. If the algorithm is greedy, it means that A_1 is closer to finding the solution than A_2

This definition therefore allows different algorithms solving the same problem to be compared. The monotonicity condition is important to allow progress metrics to be reliably compared and the second condition ensures that progress metrics between two algorithms are comparable.

General notion of progress In general, the exact nature of progress is very domain dependent and only a domain expert will be able to define such a measure. The monotonicity requirement that we impose makes finding a suitable measure all the more difficult. However, we believe that most important problems allow a notion of progress to be defined:

- **Constructive problems**: problems which require the construction of a solution have a natural monotonic progress metric.
- **Streaming problems**: problems where data has to be processed in sequence (such as media encoders and decoders) also have a natural monotonic progress metric.
- **Search-based problems**: problems which explore a search space to reach a solution have to eliminate portions of the space and the amount of space eliminated constitutes a monotonic progress metric.

The examples above are not exhaustive but show that across a wide range of domains, a notion of progress can be defined.

Measuring progress in practice In practice, progress can be measured by periodically updating a data-structure that can be used to compute progress. In most cases, the data-structure may consist of a single counter (such as the number of elements correctly placed in a sort). We provide support for such data-structures in our n-way framework described in Section 4.

3.1.2 Energy metric

Now that we have described how to measure progress, we describe how to measure, in an online manner, the energy expended by an algorithm. Although the only precise measurement is through a watt-meter and a stopwatch, it is possible to approximate the value of the energy consumed for a computation through the use of the performance monitoring counters (PMC). Indeed, the power a chip utilizes is dependent on the events it has to deal with. For example, a last-level cache miss requires more power to service as the data needs to be fetched from memory. Similarly, a branch misprediction requires more power as the pipeline needs to be flushed and restarted.

In [22], Singh et al. describe a methodology that can be used to construct a formula relating power consumption to readings from PMCs. The methodology relies on measuring the actual power consumed with a watt-meter as well as the evolution of the PMCs for a series of micro-kernels and establishing correlations between them. The method extracts the most significant (in terms of their impact on power consumption) PMCs and quantifies their contribution. Although the paper computes this formula for the Intel Q6600 chip, we believe the methodology is general enough to be applied to any chip that exposes performance counters to the user. CPUs and GPUs both fall in this category. Note that the exact way to measure power is not the main point of this paper; it is sufficient that there exist approximate ways to estimate power in real-time with PMCs. The total energy consumed can then be estimated by multiplying the estimated instantaneous power consumed by the interval over which it was measured. If the measurement intervals are small enough, we can consider the power used to be constant over the interval.

3.2 Putting it all together

By simultaneously measuring algorithmic progress and estimating energy, we can estimate a cumulative algorithmic energy efficiency for the algorithm which is the ratio of the total progress over the total energy expended. The AEE measures both the diversity due to algorithms, which translates in particular in the progress made, as well as the diversity due to hardware, which translates in different energy being consumed. As the different ways are concurrently running on the different cores (homogeneous or heterogeneous), a runtime can periodically compare the efficiency for each way and, once it becomes clear which one will overall use the least amount of energy, the runtime can terminate all other ways and solve the problem solely with the selected way on the selected core.

3.2.1 Non-uniform energy efficiency

A crucial issue with our technique is how to determine *when* it is clear that a particular way will have a lower overall energy use than the other ways. This determination has to be made early enough to actually allow overall energy saving but late enough to make an accurate determination. Indeed, the energy efficiency measure is not guaranteed to be uniform as some algorithms will do a lot of preparatory work up front (thereby having a low energy efficiency at the start) to enable them to make more rapid progress later on (thereby having a very high energy efficiency at the end). Since various algorithms may not exhibit the same behavior in terms of energy efficiency phases, it is difficult to accurately say with certainty when a particular way is going to overall use the least amount of energy. We are still investigating this problem and propose several ideas in Section 7.

3.2.2 Accurate accounting

Note that since multiple ways are going to be running in parallel we need to **i)** attribute the PMC counts to the correct way and **ii)** account for the fact that the ways will potentially interfere with each other (cache sharing for example). The first point is simplified by the fact that each way only runs on one resource (core) and does not share it with other ways. If each core has its own PMCs, the counters can be attributed to the correct way.

The second point is relevant only for cores that share a common resource (for example a last-level cache in today's multicores). It is difficult to compensate for as it is hard to efficiently and at runtime measure the interference between ways. However, our system does not require a precise measurement of power but rather it simply requires that the measurements be comparable. This assumption is not broken even if the ways interfere with each other.

4. N-way programming model

We introduced the n-way model in [2] and give here the operational details required to exploit algorithmic and hardware diversity to *dynamically select which algorithm to run on which core*. The focus of this paper is not on the description of the n-way model and we therefore give a brief overview of the model here.

4.1 The n-way model

The key idea of the n-way model is to utilize the idle cores to execute in isolation and in parallel the different ways of a given diverse problem P. Consider the sample program illustrated in Figure 1(a). A, B and C are three sequential computations. In this example B exhibits diversity. Traditionally, A will execute followed by B followed by C. In our model, A will execute followed by the ways B_1 to B_4, all executing in isolation and in parallel followed by a selection of one of the ways (let us say B_2), followed by C's execution. As far as C is concerned, only B_2 executed. This is illustrated in Figure 1(b). Note that the selection of B_2 can happen at any point during the execution of B (ie: not necessarily after B_2 finished executing). The n-way model guarantees semantic equivalence between the two executions chiefly due to the isolation guarantee.

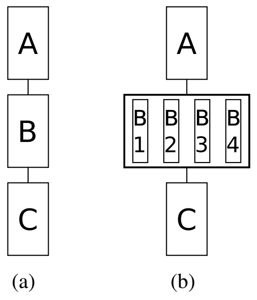

Figure 1. Sequential and n-way flow for a simple program

In our previous work, we showed how the n-way model could be used to exploit algorithmic diversity to speed-up or otherwise improve the quality of result of a computation. In this work, our goal is to select the way that consumes the least amount of energy. This selection will most likely translate into an algorithm that is also fast as it will be subject to few architectural bottlenecks which cause energy consumption to increase. An important point to note is that n-way seeks to utilize *idle* cores and will therefore not over-subscribe them. Therefore, there will be, at most, one way running per core.

To successfully exploit diversity to select the way that consumes the least amount of energy, the model supports:

- **Way isolation**: isolation is a key condition to semantic equivalence between both the n-way execution and a sequential execution of one of the ways.
- **Progress monitor**: each way must report on its progress and we introduce progress monitors as the mechanism by which a runtime can monitor in a non-intrusive manner the progress of the ways.
- **Dynamic choice and culling**: when it is clear which way will use the least amount of energy, that way is selected and all other ways are terminated.

Isolation Isolation is mainly required to ensure that data is not accessed in a conflicting manner by two ways. In heterogeneous environments where OpenCL like languages are used, data must be explicitly packaged with the tasks which makes the identification, wrapping and duplication for each way of such data trivial to implement.

However, for programs written in regular C/C++, variables must be manually (or through a compiler) identified and wrapped. Our current implementation makes use of versioned data-structures to enable isolation between ways.

Isolation is crucial in providing semantic equivalence between the different ways. However, it is not the only element and certain restrictions on the ways must be placed. In particular, the ways cannot perform any non-sandboxed action (such as interfacing with a network). This ensures that the ways can be terminated at any time without any lasting effect on the application.

Progress monitors Our model relies on a runtime to periodically monitor the progress of the different ways. When each way launches, it is passed a private copy of a *progress monitor* which encapsulates a user-defined progress data-structure. As stated earlier, this can be as simple as a counter. Throughout its execution, the way accesses the progress data-structure through the progress monitor and updates it in a way to indicate its progress. To limit overheads, our system thus relies on the good behavior of the ways to periodically update their progress.

Dynamic choice and culling When the runtime determines that it has collected enough progress information (as well as PMCs) to be able to determine with a high level of certainty the way that will consume the least amount of energy, it will terminate all other ways and leave the selected way to proceed uninterrupted. Termination of the different ways will occur when they next try to update their progress monitor. A terminated way will free up the core it is occupying and given the isolation guarantee, this will have no impact on the continuation of the program.

4.2 N-way parallelism

An interesting aspect of the n-way model is that it allows a different type of parallelization of sequential codes. Instead of exploiting data or task parallelism in a section of code, n-way parallelism exploits algorithmic diversity as its source of parallelism. A programmer can therefore easily take a library of different implementations solving the same problem (for example different sort algorithms, or different SAT solving algorithms) and merge them into a n-way implementation that will be parallel.

Concretely, suppose that 2 different implementations for sorting are available `quick_sort` and `merge_sort`, each taking an array A as argument and returning the sorted array in place. Conceptually, the n-way `nway_sort` will look like this:

```
nway_sort(int* A) {
        register_way(quick_sort);
        register_way(merge_sort);

        launch_ways(A, lowest_energy);
}
```

The different implementation possibilities are made known to the runtime through the `register_way` API and the `launch_ways` API launches in isolation and in parallel the various ways picking the one that meets the programmer specified goal (here, using the lowest amount of energy).

5. Experimental validation

To validate the feasibility of our idea we used different sorting algorithms. The n-way framework can currently monitor the progress metric in an algorithm through the use of the progress monitors described in Section 4 and the results below motivate the utility of adding energy monitoring as well.

Note that we utilized sorting algorithms as the diversity present in sorting is well-known [14] and progress is easy to define.

5.1 Experimental setup

We ran all benchmarks on an Intel Core 2 Duo Q6700 running at 2.66 GHz with 2 GB of RAM. We utilized PAPI 4.0 software running on a patched 2.6.26 Linux kernel for PMC monitoring.

Progress in the "sort" problem We defined progress as the percentage of the total number of "moves" required to put all elements in their correct places. The quicksort algorithm, for example, moves each element once to its final correct position (when that element is used as a pivot). In a mergesort however, each element can be moved up to $\log_2(n)$ as the list an element belongs to will be merged $\log_2(n)$ times with another list. Note that although the exact number of moves varies, the notion of progress stays the same and we can therefore define progress independently of the implementation solely based on high-level knowledge about the sort problem. By design, progress is a monotonically increasing function and. Total progress is thus between 0 when no elements have been moved in place and 1 when the array is sorted. As the sort algorithm progresses, it updates a counter which is periodically polled.

Notion of energy We measured the average instantaneous power consumed over a small time increment as described in [22] for the Q6600 processor. The time increment is small enough to assume that the power consumed stays constant over the interval. Since the Q6700 and the Q6600 are very similar (except for clock-speed) we used it as is (changing only the clock-speed) and removed the temperature influence. Note that since the Q6700 only allowed 2 HW counters in 32 bit mode, the 4 counters required by the power formula are multiplexed (measured every other polling “tick”).

We kept track of the cumulative energy consumed by the processor over the execution of the entire benchmark.

Both progress and average power are computed every $1ms$ and the values are dumped at the end of the run.

Benchmarks We used different standard sort implementations: the standard `qsort`, an improved quicksort (`qui`) which performs an insertion sort for short arrays, an improved quicksort to sort linked lists (`quilist`), a shell sort (`shl`), a heap sort (`heap`) and a merge sort (`merge`). All algorithms were inspired from [7, 20].

5.2 Results

We do not seek to compare the performance of the different sort algorithms, rather we wish to show that it is possible to pick, early on, the most efficient algorithm to reduce the overall average energy consumption. Although the sort algorithms are simple and well understood, they illustrate this point convincingly.

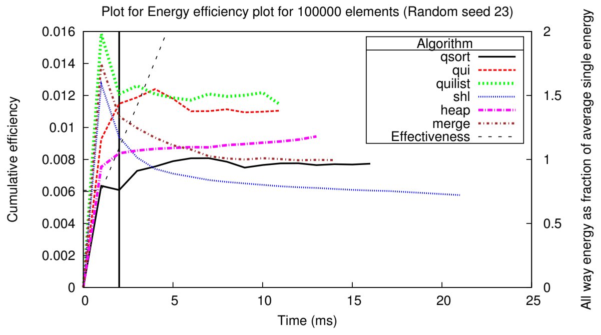

Figure 2. The sort algorithm on a small dataset ($100,000$ random elements)

In Figures 2 and 3 we present the cumulative algorithmic efficiency for a small randomly generated dataset ($100,000$ elements) and a large one ($50,000,000$ elements). The cumulative algorithmic efficiency corresponds to the running ratio of the algorithmic progress to the cumulative energy consumed. Note that certain lines finish earlier than others as not all algorithms take the same amount of time to run.

Before validating our technique, we note that the graphs behave as one would expect. In particular, the quicksort type of sorts have a plateau of an efficiency of 0 at the beginning which corresponds to the algorithm

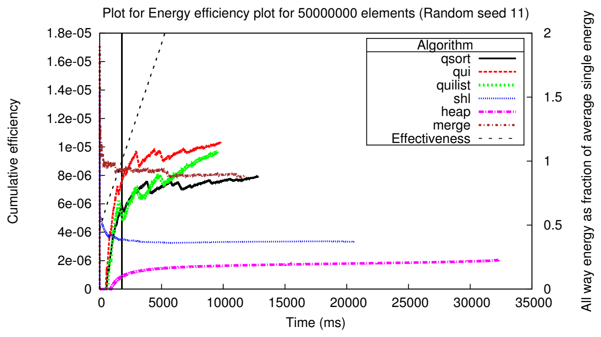

Figure 3. The sort algorithm on a large dataset ($50,000,000$ random elements)

“dividing” the problem before conquering. Indeed, it first needs to get to the leaf partitions before moving a single element into place. Similarly, we see that the rate of increase of the heap-sort efficiency increases. This can be explained by the fact that placing elements gets easier and easier as the heap is smaller and smaller.

The key question in our technique is whether the choice for the most efficient way can be made early enough to actually save energy. We represent this as the `effectiveness` line in the Figures. The line corresponds to the ratio of $P_t(X)$ over P_a where $P_t(X)$ is the sum of the energy used by all the ways up until time X plus the minimum energy required to complete the computation and P_a is the average energy used by a single way. When we can pick the most efficient way when $P_t(X) < P_a$ we have saved energy on average. This corresponds to an `effectiveness` of less than 1 and is represented by the vertical solid line. Therefore, if we can determine the most energy efficient way at a time to the left of that line, the technique is viable. We note that this is very clear for the small data-set (picking the `quilist` algorithm) but for the large data-set we would most likely pick the `merge` algorithm instead of the `qui` one. This particular case is due to the fact that merge-sort has a very high efficiency at the beginning. This has to do with the notion of progress we defined as the merge sort starts moving elements around much more quickly than the quicksort algorithms (although it moves them around more often). Therefore, its efficiency is much higher to begin with. A different measure of progress could mitigate this issue. In any case, even the choice of merge-sort would produce a second best energy efficient choice.

6. Related work

Machine learning techniques have been used to try to learn the best match between a specific algorithm and a hardware target [16] or between different algorithms and a hardware target [14]. Our approach is less about using learned information to adapt but rather about selecting dynamically and just in time the most energy efficient algorithm. Learning can also play a part in our approach by allowing for a smarter and smaller set of possibilities from which to choose from and therefore reducing the overall number of "ways" to run simultaneously.

The problem of matching the computation to heterogeneous resources was also partially addressed by the GLIMPSES [25] approach that allows selection of partitions to be offloaded to the SPUs of a Cell processor. The idea behind GLIMPSES is to obviate the need for a model by selecting partitions that meet *all* of a predefined set of conditions: low percentage of conditional branches, high amount of auto-vectorizable loop and a memory partition which fits in the local SPU's memory (to obviate the need for a potentially inefficient software cache). However, GLIMPSES does not deal with dynamic conditions nor does it analyze the tradeoffs when some of these conditions are not met (such as a high percentage of vector loops but also a high percentage of conditional branches).

The evidence of algorithmic diversity, which we exploit here to obtain more energy efficient algorithms, is widespread [10, 15, 17]. Diversity has also been exploited in some very specific situations (for example, ManySAT [4]) to obtain speedups. However, to our knowledge, the utilization of diversity to select a good algorithmic fit for a given hardware is novel.

Utilizing the PMC to monitor energy or power consumption for a thread is not new and has been used mainly for scheduling [22] or determining a "bias" towards a particular type of core [12]. However, our work is novel because we do not only monitor power but associate it with an algorithmic metric indicating the progress of the algorithm. Our approach is therefore able to compare different algorithms solving the same problem whereas a pure energy based approach would not.

7. Conclusion and future work

In this position paper, we have motivated the need and usefulness of monitoring a novel metric: *algorithmic energy efficiency* which captures how well an algorithm is performing on a platform for a given input data. We have described how the n-way paradigm could be adapted to exploit this metric and utilize idle cores to select the most energy efficient algorithm.

Note that our technique also applies to finding the algorithm that *best matches* a given platform. Instead of using the PMC to estimate power consumption, they can be used to evaluate a "matching" criteria to the platform. This could be particularly useful for heterogeneous platforms. For example, GPUs are very efficient for vector computations but highly inefficient when there are too many branches. A hardware impact metric taking into account the number of branches could help select the appropriate algorithm for the platform.

As future work, we will investigate how best to deal with algorithms which have very non-uniform energy efficiency. We believe that the relative behavior of different phases of an algorithm could be characterized offline and used to weigh the energy efficiency metric appropriately. We could therefore define a "progress profile" for an algorithm and greatly reduce the effect of non-uniformity in algorithmic progress.

We are also currently working on expanding our testing framework to other benchmarks to see how widely applicable our idea is. In particular, although we are only considering sequential codes at this point, we believe that diversity is also present in the way a program is parallelized with differences in energy consumption being caused by different types of parallelization approaches.

Acknowledgments

The authors gratefully acknowledge the support of NSF grants CCF-0702286 and CCF-0916962. The authors would also like to thank the reviewers for their comments which greatly improved the final version of this paper.

References

[1] AMD Corporation. Amd fusion family of apus. `http://fusion.amd.com`, 2010.

[2] R. Cledat, T. Kumar, J. Sreeram, and S. Pande. Opportunistic computing: A new paradigm for scalable realism on many cores. In *HotPar 2009: 1st USENIX Workshop on Hot Topics in Parallelism*. USENIX, 2009.

[3] A. E. Eichenberger, K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the cell broadband engine$^{\text{tm}}$ architecture. *IBM Systems Journal*, 45(1):59–84, 2006.

[4] Y. Hamadi, S. Jabbour, and L. Sais. Manysat: Solver description. Technical Report MSR-TR-2008-83, Microsoft Research, May 2008.

[5] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41:33–38, 2008.

[6] H. P. Hofstee. Power efficient processor architecture and the cell processor. In *HPCA '05*, pages 258–262, Washington, DC, USA, 2005. IEEE Computer Society.

[7] P. Hsieh. Sorting revisited. `http://www.azillionmonkeys.com/qed/sort.html`, April 2010.

[8] Intel Corporation. Intel embedded processor for 2008 (tolapai). http://download.intel.com/technology/quickassist/tolapaisoc2008.pdf, 2008.

[9] Intel Corporation. Westmere family of processors. http://download.intel.com/pressroom/kits/32nm/westmere/32nm_WSM_Press.pdf, 2010.

[10] S. K. Iyer, J. Jain, M. R. Prasad, D. Sahoo, and T. Sidle. Error detection using bmc in a parallel environment. In *CHARME*, pages 354–358, 2005.

[11] Khronos. Opencl. http://www.khronos.org/opencl/, 2010.

[12] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *EuroSys*. ACM, 2010.

[13] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 81, Washington, DC, USA, 2003. IEEE Computer Society.

[14] X. Li, M. J. Garzarán, and D. Padua. A dynamically tuned sorting library. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 111, Washington, DC, USA, 2004. IEEE Computer Society.

[15] M. Luby and W. Ertel. Optimal parallelization of las vegas algorithms. In *STACS '94*, pages 463–474. Springer, 1994.

[16] C.-K. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 45–55, New York, NY, USA, 2009. ACM.

[17] M. Mitzenmacher and E. Upfal. *Probability and Computing*. Cambridge University Press, 2005.

[18] T. Moseley, D. Grunwald, and R. Peri. Chainsaw: Using binary matching for relative instruction mix comparison. In *PACT*, pages 125–135, 2009.

[19] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[20] T. Niemann. Sorting and searching algorithms. http://epaperpress.com/sortsearch/download/sortsearch.pdf, April 2010.

[21] C. G. Q. nones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on precomputation slices. In *PLDI '05*, pages 269–279, New York, NY, USA, 2005. ACM Press.

[22] K. Singh, M. Bhadauria, and S. A. McKee. Prediction-based power estimation and scheduling for cmps. In *ICS '09*, pages 501–502, New York, NY, USA, 2009. ACM.

[23] D. C. Snowdon, E. Le Sueur, S. M. Petters, and G. Heiser. Koala: a platform for os-level power management. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 289–302, New York, NY, USA, 2009. ACM.

[24] Y. Song, S. Kalogeropulos, and P. Tirumalai. Design and implementation of a compiler framework for helper threading on multi-core processors. In *PACT '05*, pages 99–109, Washington, DC, USA, 2005. IEEE Computer Society.

[25] J. Sreeram and S. Pande. GLIMPSES: A profiling tool for rapid SPE code prototyping. In *New Horizons in Compilers*, 2007.

[26] TomLab. Cplex parameters interface. http://tomopt.com/docs/cplexug/tomlab_cplex014.php, March 2010.

[27] V. Vazirani. *Approximation Algorithms*. Springer, 2001.

[28] Workshop on the Interaction between Operating Systems and Computer Architecture, in conjunction with ISCA. *Scheduling on Heterogeneous Multicore Processors Using Architectural Signatures*, 2008.

[29] X. Zhuang, A. E. Eichenberger, Y. Luo, K. O'Brien, and K. O'Brien. Exploiting parallelism with dependence-aware scheduling. In *PACT '09*, pages 193–202, Washington, DC, USA, 2009. IEEE Computer Society.