Profitability-Based Power Allocation for Speculative Multithreaded Systems*

Polychronis Xekalakis ‡

Intel Barcelona Research Center
Intel Labs Barcelona – UPC
polychronisX.xekalakis@intel.com

Nikolas Ioannou, Salman Khan,
Marcelo Cintra

School of Informatics
University of Edinburgh
{nikolas.ioannou, salman.khan}@ed.ac.uk,
mc@staffmail.ed.ac.uk

ABSTRACT

With the shrinking of transistors continuing to follow Moore's Law and the non-scalability of conventional out-of-order processors, multi-core systems are becoming the design choice for industry. Performance extraction is thus largely alleviated from the hardware and placed on the programmer/compiler camp, who now have to expose Thread Level Parallelism (TLP) to the underlying system in the form of explicitly parallel applications.

Unfortunately, parallel programming is hard and error-prone. The programmer has to parallelize the work, perform the data placement, and deal with thread synchronization. Systems that support speculative multithreaded execution like Thread Level Speculation (TLS), offer an interesting alternative since they relieve the programmer from the burden of parallelizing applications and correctly synchronizing them.

Since systems that support speculative multithreading usually treat all threads equally, they are energy-inefficient. This inefficiency stems from the fact that speculation occasionally fails and, thus, power is spent on threads that will have to be discarded. In this paper we propose a power allocation scheme for TLS systems, based on Dynamic Voltage and Frequency Scaling (DVFS), that tries to remedy this inefficiency. More specifically, we propose a profitability-based power allocation scheme, where we "steal" power from non-profitable threads and use it to speed up more useful ones. We evaluate our techniques for a state-of-the-art TLS system and show that, with minimal hardware support, they lead to improvements in ED of up to 39.6% with an average of 21.2%, for a subset of the SPEC 2000 Integer benchmark suite.

1 INTRODUCTION

With devices continuing to scale according to Moore's Law, we have witnessed a shift towards the multi-core design paradigm for both industry and academia. This shift is mainly a consequence of our inability to scale the out-of-order cores in an efficient way, so that instead of investing the available transistor budget to build wider processors, we do so to replicate relatively simpler cores. This of course imposes challenges for the programmers/compilers who now have to devise thread-level-parallel applications, so as to exploit the available resources. Unfortunately parallel programming is hard and error-prone and compilers still fail to automatically parallelize all but the most regular programs.

Thread Level Speculation (TLS) [10, 13, 16, 23] allows the compiler/programmer to freely generate threads without having to consider all possible cross-thread data dependences. Parallel execution of threads then proceeds speculatively and the TLS system guarantees the original sequential semantics of the program by transparently detecting any data dependence violations, squashing the offending threads, and returning the system to a previously non-speculative correct state.

Under the TLS execution model, threads are speculatively executed with the hope that this does not result in dependence violations. Of course this is not the case always, and the offending threads have to be squashed and restarted. Although this optimistic view of the existence of dependences usually results in a better performance, it does so at the cost of an increase in power when speculation fails. It is interesting to note that even when speculation fails, we may still be able to gain benefits, due to prefetching effects as it was previously reported in [19] and [28].

[*] This work was supported in part by EPSRC under grant EP/G000697/1 and the EC under grant HiPEAC IST-004408.

[†] This work was done before the author joined Intel, while being at the University of Edinburgh.

[‡] The author was supported in part by a Wolfson Microelectronics scholarship.

978-1-4244-6443-2/10/$26.00 ©2010 IEEE

If we could separate profitable threads from the non-profitable ones, then we could allocate the power resources accordingly, and thus increase the energy efficiency of the system. This is the main ambition of this paper. More specifically, we first try to identify threads that provide neither TLP, nor ILP benefits and distinguish them from those that do. By classifying threads into profitable and non-profitable ones, we can increase the efficiency of our speculative system by allocating power according to their profitability. More specifically, threads predicted to be non-profitable are put in one of the low power modes, allowing us to spend the power saved to accelerate the profitable ones. We guide our scheme based on two predictors: a dependence predictor able to detect lack of TLP, and a memory boundedness predictor able to estimate lack of ILP.

Apart from being able to classify threads into profitable and non-profitable ones, we also require a mechanism to regulate the power resources accordingly. We could potentially implement this by supporting multiple types of cores (i.e., low power ones, high power ones, etc.) and migrate the threads accordingly. However, since our threads are by construction small, migration will likely have serious performance repercussions. We instead choose to implement different power modes by performing Dynamic Voltage and Frequency Scaling (DVFS) [15] on each core, which, when done by on-chip regulators, is a proven and fast way to trade power for performance [12].

Applying our profitability-based power allocation scheme to a state-of-the-art TLS system, we are able to achieve significant speedups with a reasonable increase in the power consumed. More specifically, by evaluating our technique for a subset of the SPEC2000 Int benchmarks, we show that with only minimal hardware support, we are able to achieve improvements in the overall Energy-Delay [1] (ED) of up to 39.6% with an average of 21.2%.

The rest of this paper is organized as follows: in the next Section we provide some background information. In Section 3 we outline our proposed scheme. In Section 4 we describe our methodology and in Section 5 we present our results. Finally, in Section 7 we discuss related work, and in Section 8 we conclude the paper.

2 BACKGROUND

2.1 Background on TLS

Under the *thread-level speculation* (also called *speculative parallelization* or *speculative multithreading*) approach, sequential sections of code are speculatively executed in parallel hoping not to violate any sequential semantics [10, 13, 16, 23]. The control flow of the sequential code imposes a total order on the threads. At any time during execution, the earliest thread in program order is *non-speculative* while the others are *speculative*. The terms *predecessor* and *successor* are used to relate threads in this total order. Stores from speculative threads generate unsafe *versions* of variables that are stored in some sort of *speculative buffer*. If a speculative thread overflows its speculative buffer it must stall and wait to become non-speculative. Loads from speculative threads are provided with potentially incorrect versions. As execution proceeds, the system tracks memory references to identify any cross-thread data dependence violation. Any value read from a predecessor thread, is called an *exposed read*, and it has to be tracked since it may expose a Read-After-Write (RAW) dependence. If a dependence violation is found, the offending thread must be *squashed*, along with its successors, thus reverting the state back to a safe position from which threads can be re-executed. When the execution of a non-speculative thread completes it *commits* and the values it generated can be moved to safe storage (usually main memory or some shared lower-level cache). At this point its immediate successor acquires non-speculative status and is allowed to commit. When a speculative thread completes it must wait for all predecessors to commit before it can commit. After committing, the processor is free to start executing a new speculative thread.

2.2 Background on DVFS

The basic dynamic power equation: $P = CV^2Af$ clearly shows that there is a great opportunity to save power by adjusting voltage and frequency. By reducing the voltage by a small amount, we reduce power by the square of that factor. Unfortunately, reducing the operating voltage means that the transistors need more time in order to switch on and off, which also forces a reduction in the operating frequency. Dynamic Voltage and Frequency Scaling (DVFS) [15] techniques try to exploit this relationship by reducing the voltage and the clock frequency when they discern that they can do so without causing a proportional reduction in performance.

Adjusting the voltage and frequency is done by means of a DC-DC converter, which changes the voltage to the desired levels. The new operating voltage is then used to drive the frequency generator, which provides the chip with the operating frequency for the corresponding voltage level. Having a means of changing the voltage and frequency, one has to decide whether to put the DC-DC converter off-chip [18, 27] or on-chip [1, 11]. Placing the converter off-chip, we are limited in that we can only change the voltage and frequency of the entire chip. A second related issue is that off-chip regulators are generally slow. However, they consume less power and require a smaller hardware budget than their on-chip counterparts. On the other hand, this additional area

[1] Energy Delay is a combined metric, that is the product of the energy a system expended to perform an operation and the time required to perform it.

and power consumption grants on-chip regulators faster response times and allow changes of the voltage and frequency to parts of the chip.

3 PROFITABILITY-BASED POWER ALLOCATION

3.1 Basic Idea

Most modern processors have support for DVFS in order to save power when cores are underutilized or to avoid thermal emergencies [8]. As experiments done in [9] showed, it is advantageous to reduce the CPU frequency for memory intensive tasks, but not for the CPU intensive ones. In fact, for tasks with high CPU utilization, the system performance is linearly dependent on the operating frequency, and thus decreasing the frequency results in significant performance loss. On the other hand, for memory intensive tasks, decreasing the operating frequency has only a mild effect on the overall performance, since these tasks typically stall while waiting for the memory requests to be serviced. It thus makes sense, from an energy efficiency perspective, to lower the frequency for a memory intensive tasks, and increase it only when running CPU intensive ones.

Unfortunately, for TLS systems this is not enough. A real-life scenario for systems that support these systems, is that a significant fraction of the threads have to be rolled-back due to dependence violations. This suggests that from a performance point of view, some of the threads are profitable, while some others are not. In fact much of the energy inefficiency of TLS stems from the fact that we spend the same amount of power to execute threads that will procure performance benefits and those that will not. In this paper we propose to try to adapt the power consumed by threads based on their expected profitability.

We leverage the fact that modern processors, like Intel's Nehalem [7], are able to increase the operating frequency of a core if the rest of the cores are either idle or on one of the low-power modes. Instead of relying on the OS to decide how to allocate the power resources to each of the cores, we use hardware predictors to guide the power allocation at run-time. In our system, we assume four power modes: the *very-low-power* mode, the *low-power* mode, the *normal-power* mode and the *high-power* mode. Each mode corresponds to a different frequency-voltage pair. We assume that the normal-power mode is the operating mode when all of the cores are busy. We also assume that, as with Intel's Nehalem, the high-power mode cannot be used if all of our cores are operating at normal-power mode, due to power delivery issues.

More specifically, since the only thread that will surely commit in a TLS system is the safe thread, it intuitively makes sense to try to put it in high-power mode when this is possible. As we previously mentioned, we are only able to do so if one of the other threads is in one of the low-power modes (Figure 1(a)), or clock-gated (Figure 1(b)). On the other hand, we put threads in one of the low-power modes if we predict that they will squash due to a violation of the sequential semantics or if we estimate that they are memory bound, and as such reduced frequencies/voltages will only result in minor performance loss (Figure 1(c)). We keep the remaining threads in the normal-power mode.

Although allocating more power to the thread that acquires the safe token is straightforward, finding the threads that will be victimized is not. One of the difficult aspects of finding the *non-profitable* threads stems from the fact that our predictions are used to guide the execution speed of the different threads and the same thread can potentially be profitable or non-profitable based on our decisions. Fortunately, by using hardware predictors like the ones presented in the following sections, our system adapts to this run-time behavior. We use two predictors: one able to predict if a thread has performed a load that will cause it to squash, and one able to estimate if the thread is memory bound. Threads predicted to squash go into the low-power mode, while threads predicted to squash for more than three times go into the very-low-power mode. Similarly, threads estimated to be memory bound go to low-power mode.

Note that as is shown in Figure 1(d), when the safe token is passed to the next thread, so do the power resources. As we will show in Section 5.3, this results in more uniform distribution of the power consumed than that of a normal TLS system, and thus our scheme enjoys a better thermal behavior. In fact as we will show in the same section, although we increase the overall power consumed, we reduce the temperature of the chip significantly. This result suggests that our scheme would experience less thermal emergencies and as such it could potentially be clocked at even higher frequencies.

Note also that if our predictions are right and the threads do squash or are memory bound, we have saved some power. This is important since this extra power can be spent to speed up safe threads. At the same time the extra power consumed executing safe threads in high-power mode, does not increase significantly the average power consumed. On the other hand, if our predictions are wrong, we slow down useful threads and allocate more power to non-profitable threads. Due to the central role these predictors play in our design, they have to be quite accurate. In the next two sections we describe how they operate and discuss various design options.

3.2 Adapting to TLP

We first target threads that are not going to be useful in terms of TLP. We base our classification on a *Squash Pre-*

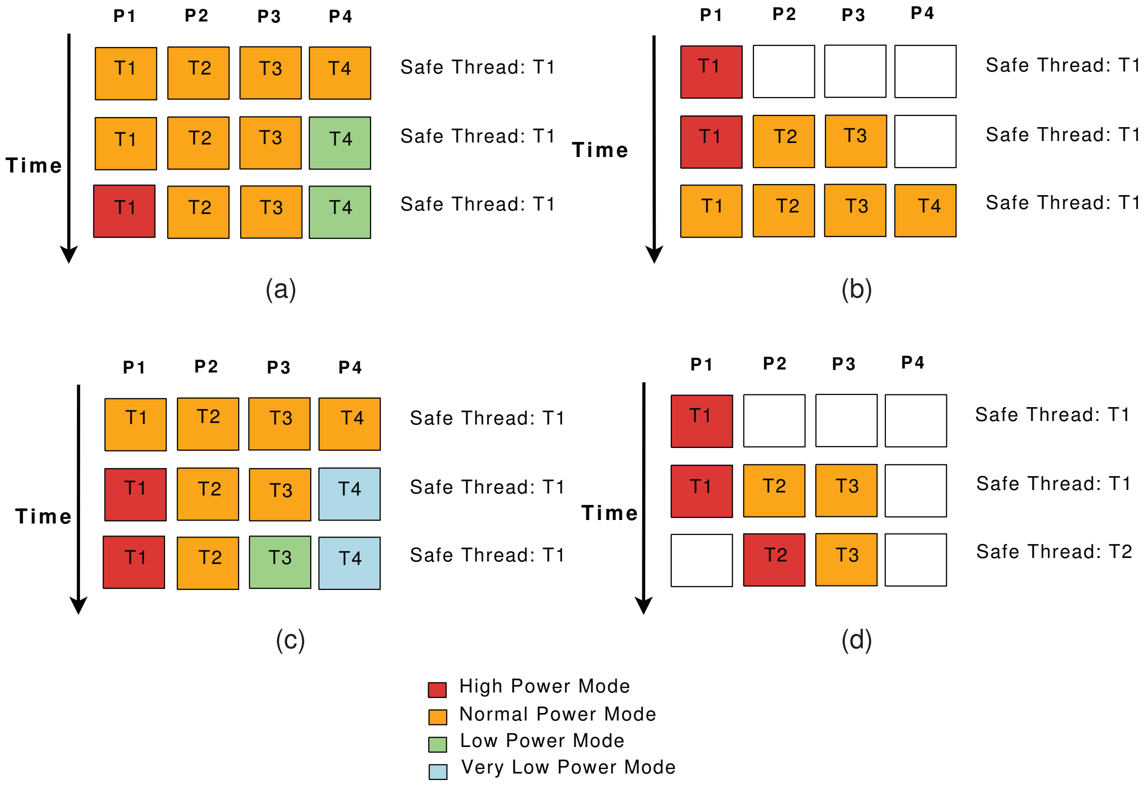

Figure 1. Profitability-based power allocation: (a) When all cores are occupied, only when one thread goes in low-power mode we put the safe thread in high-power mode. (b) While there are free processors (clock gated), the safe thread *T1* is set in high-power mode. (c) When a thread is predicted to squash or to be memory-bound it goes in low-power mode, when it is predicted to squash and to be memory bound it goes into very-low-power mode. (d) If a safe thread (*T1*) finishes and the subsequent thread becomes safe (*T2*), the *high-power core* becomes the one holding the current safe thread.

dictor: a dependence predictor able to predict whether a specific load will squash the thread or not. A dependence predictor has been previously proposed in [17], but it relies on global information about possibly conflicting loads and stores. Such a centralized scheme is not practical for a multi-core system like the one we use. This has been previously noted in [6], where the authors designed dependence predictors for a directory-based CC-NUMA system. Instead of using a centralized scheme, they extended the directories with a few bits so as to capture the dependence behavior using only memory addresses.

We opt for a similar solution to the one presented in [6]. More specifically, we maintain a simple table of three bit saturating counters per core as is shown in Figure 2(a). When a speculative thread (i.e., all threads but the safe one) tries to execute a load instruction, we perform a bit-wise *XOR* of the memory address and the five least significant bits from the load's program counter and form an index. We use this index to look up the corresponding counter from the table we mentioned before. If the value of the corresponding counter is larger than three, we then predict that the specific load is being performed prematurely and will thus cause the thread to squash, otherwise we predict that the specific load will not cause any problems. At the same time we also update the tag of the cache line that holds the specific memory address, with the five least significant bits of the program counter. The predictor is updated when we perform a store. The memory address of the store is propagated to all the cores, since it is used by the TLS protocol to uncover any dependence violations by checking whether any of the caches holds a violating load. If the store does reveal a dependence violation, we read out the five bit field that holds the program counter of the load that last touched the cache line. These bits are *XORed* with the memory address of the store and are used to index the table of saturating counters, which we increment by two. If a thread becomes safe, and thus can no longer be squashed, it updates the predictor when each of the lines that belong to it are written back. As with the previous case, we read out the PC bits and use them along with the address of the write-back request to index the counter and decrement it by one. We choose to perform this lazy update of the squash predictor since in this way we can ensure that for each of the memory addresses for the threads that commit, we only update the predictor once (as opposed to updating on every store that does not cause a squash). This allows us to employ a small number of bits per entry in the saturating counter table.

Although the main focus of this paper is not in creating novel dependence predictors, but rather on using them so as to guide power allocation, as we will show, the proposed predictor is better than the previously proposed ones. Note that similarly to [6] we slightly augment the cache lines, but we only do so to hold information about the program counter that performed the specific load. As we will show in a subsequent section, this improves the prediction accuracy, since our predictor is able to disambiguate accesses to the same memory location from different sections of the code.

Having predicted which threads will get squashed, we now have to decide how much to slow them down. One option would be to stall them completely. Albeit simple, this approach can be very detrimental in terms of performance. As was pointed out in [28] even threads that do squash may be useful for prefetching reasons. By stalling threads that squash, we remove much of this desirable side-effect of TLS execution. An additional reason why this approach hurts performance is that although fairly accurate, our squash predictor may be wrong. Since the cost of being wrong is high, we would have to make our predictor fairly conservative in predicting that a thread will squash. This results of course in lost opportunity to put threads in low-power mode (i.e., stall them in this case), which in turn results in not being able to put the safe thread in high-power mode. We thus put these threads in low-power mode instead. When a thread is predicted to squash more than three times, we can be more aggressive and put the thread in very-low-power mode. Note that in this way we wrongly put a thread in very-low-power mode only when we mispredict three times in a row, in one task. We thus keep a two-bit counter per core where we account for the number of times the thread has been predicted to squash.

3.3 CPI-Based Adaptation

Although by using the squash predictor we can potentially improve energy efficiency, we can still do a better job by reducing the consumed power according to how much this will affect the thread's performance. In fact, for memory-bound threads, most of their time is spent waiting for memory operations to be serviced. Executing them in normal-power mode is wasteful since they will consume valuable power without achieving any performance benefits out of this.

It is thus clear that we need a mechanism to decide whether a thread will wait for costly memory operations or not, since this provides an additional source of energy inefficiency. Our estimator, depicted in Figure 2(b), uses per core, two five bit saturating counters, a five bit multiplier, and a comparator. In order to predict whether a thread is memory bound or not, when a thread is executing we keep track of how many times we have had an unresolved memory access reaching the head of the ROB. These memory accesses are the ones that are important since they stall the pipeline. At the same time we also keep track of how many instructions we have successfully committed. When we wish to make a prediction we multiply the miss counter with half the size of the instruction window (in number of in-flight instructions it can accommodate concurrently) and compare this with the number of instructions committed. If the product is larger than the number of instructions, we predict that we have spent most of our execution time waiting for memory and thus our thread is memory bound, if it is smaller we predict otherwise. Threads predicted to be memory bound are put in low-power mode. We consult the estimator, and thus decide on the ”memory-boundedness” of the thread, each time we have a miss in the shared L2 cache.

There are two interesting points to make here. The first is that although this mechanism is triggered on every L2 miss, we only allow it to have an effect if the thread is believed to be memory bound. That is, if a L2 miss happens and the prediction is that the thread is not memory-bound, no action is taken. By allowing each evaluation to have an effect, we could have corrected some erroneous estimations by putting the threads that are in low-power mode, back to normal-power mode if this was not the necessary. In practice we found that if we allow a given number of cycles before we take any decision, we do not have to do this. As such the memory-boundedness predictor only operates after 1k cycles have elapsed since the beginning of the execution of the thread. The second interesting thing to note, is that the squash predictor may have already placed a thread in the very-low-power mode. In such cases the memory- boundedness predictor, will not put the thread in the low-power-mode even though it may be memory bound.

4 Evaluation Methodology

We conduct our experiments using the SESC simulator [21]. The main microarchitectural features are listed in Table 1. The system we simulate is a multi-core with 4 processors, where each processor is 4-issue out-of-order superscalar. The power consumption numbers are extracted using CACTI [24] and wattch [3]. The thermal simulations are per-

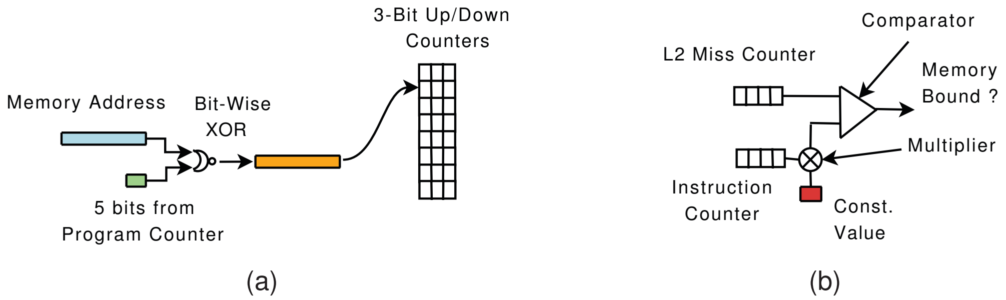

Figure 2. (a) Squash Predictor: The components we need are a small table of up/down counters and five bits from the PC of all loads that will be predicted. (b) Memory-Boundedness Estimator: The components we need are a level 2 cache miss counter, an instruction counter, a multiplier and a comparator.

formed using Hotspot [22]. For the TLS protocol we assume out-of-order spawning [19].

Each one of the cores along with its associated L1 caches form a separate voltage/frequency domain. The shared L2 cache together with the interconnection network belong to a different domain (which is fixed). On-chip regulators are placed per core so as to implement the different power domains, in a similar fashion to [12]. In order to synchronize communication between the distinct domains that operate asynchronously to each other we use the mixed-clock FIFO design proposed in [5].

We only assume four voltage and frequency domains, as shown in Table 1, similarly to the offered domains in current commercial designs (i.e., the Super Low Frequency Mode, Low Frequency Mode, Normal Frequency Mode and High Frequency Mode used in [7]). All cores operate at the normal-power mode except if our predictions dictate we should do otherwise. The cost for changing a power mode depends on the voltage swing and is modeled to be 1 ns per 10mV in accordance to [12].

Our scheme requires on top of the baseline TLS support, the necessary hardware to perform the squash prediction and the CPI estimation (per core). More specifically, for the squash predictor we need to augment the tags of the cache lines to hold the five least significant bits of the program counter performing the access. Using CACTI we found that the overhead of these extra bits was small enough, so as not to affect the number of cycles needed to access it. We additionally need a 5-bit *XOR* and one small table of up/down counters. For the CPI estimator, we require two 5-bit counters, a 5-bit *XOR*, a 5-bit multiplier and a comparator.

Since the proposed scheme changes performance and power simultaneously, we need to use a combined metric so as to be able to quantify the different design points. One such metric proposed in [4] is the Energy Delay product (ED), which allows us to quantify both power and execution time simultaneously. Since the energy component of ED already uses the execution time (i.e., Energy = Delay x Power), the metric emphasizes more on execution time than it does on power, and as such it is suitable to evaluate TLS systems which aim at improving performance.

4.1 Benchmarks

We use the programs from the SPEC CPU 2000 Integer benchmark suite running the Reference data set. We use the entire suite except *eon*, which cannot be compiled because our infrastructure does not support C++, and *gcc* and *perlbmk*, which failed to compile in our infrastructure. For reference, the sequential (non-TLS) binaries where obtained with unmodified code compiled with the MIPSPro SGI compiler at the O3 optimization level. The TLS binaries were obtained with the POSH infrastructure [14]. In order to directly compare them, we execute a given number of simulation marks, which pinpoint specific code segments. This is necessary because the binaries are different, due to rearrangements of the code by POSH. We simulate enough simulation marks so that the corresponding sequential application graduates more than 750 million instructions.

5 EXPERIMENTAL RESULTS

5.1 Profitability-Based Scheme Versus Static Schemes

Figure 3 depicts the ED when we compare our scheme against three static power modes, namely the *very-low-power*, the *low-power* and the *normal-power* modes. Note that we do not compare against the *high-power* mode since we assume that having all the cores operating in the high-power mode is not possible due to physical constraints.

As Figure 3 shows, the normal-power mode is better than both very-low-power mode and low-power mode, because

Parameter	TLS (4 cores)
Fetch/Issue/Retire Width	4, 4, 4
L1 ICache	16KB, 2-way, 2 cycles
L1 DCache	16KB, 4-way, 3 cycles
L2 Cache	1MB, 8-way, 10 cycles
L2 MSHR	32 entries
Main Memory	500 cycles
I-Window/ROB	80, 104
Ld/St Queue	54, 46
Branch Predictor	32Kbit OGEHL
BTB/RAS	2K entries, 2-way, 32 entries
Minimum Misprediction	12 cycles
Cycles from Violation to Kill/Restart	20
Cycles to Spawn	20

Extra Hardware per Core	
Squash Predictor	2K Entries / 3bit
Instruction Counter	5bits
L2 Miss Counter	5bits
Bitwise XOR	10 x 2-input XOR gates
Comparator	1

Power Modes	
High Power Mode	5.0GHz / 1000mV
Normal Power Mode	4.0GHz / 950mV
Low Power Mode	3.0GHz / 900mV
Very Low Power Mode	1.0GHz / 700mV

Table 1. Architectural parameters used along with extra hardware required for the proposed scheme, and the different power modes available in the simulated system.

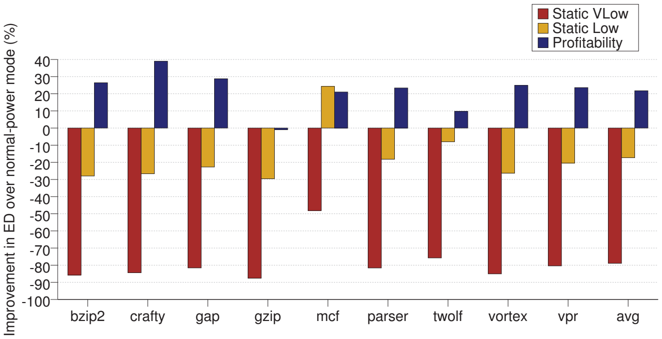

Figure 3. Improvement in ED over normal-power mode for very-low-power mode, low-power mode and the profitability-based scheme (%).

although both of them consume considerably less power, they are too slow. Interestingly, for the memory-bound *mcf* the low-power more is better than all the other schemes, since for that frequency the threads execute in a way that reduces the number of squashes significantly (values are consumed right after they are produced). We see that our profitability-based scheme is able to provide a better tradeoff, and it is thus 21.2% better on average than the best static scheme. Note that for some benchmarks, like *crafty* and *gap*, we are able to improve ED by 39.6% and 29.4% over the normal-power mode. The reason for this is that for these benchmarks our squash prediction scheme works really well. In the next sections, we provide a detailed analysis of how our system is able to achieve these significant benefits in ED.

5.2 Performance-Power Analysis

Figure 4(a) depicts the speedup (or slowdown) of the static power schemes and our profitability-based scheme over the normal-power mode. Note that frequency changes are detrimental in terms of performance. In fact, the very-low power mode is almost 67% slower than the base frequency mode, whereas the low-power mode is 16% slower. On the other hand, the performance of the profitability-based scheme is always better than that of the base operating frequency. On average the profitability scheme is 11% faster, with *bzip2, crafty* and *vortex* achieving speedups close to 20%.

At the same time, the power consumed is 2.4% on average more than that of the normal-power mode. Note that the other static schemes are able to save far more power (49% for the very-low power mode and 16% for the low-power mode on average). However, as we showed in Figure 4(a) this comes at a fairly large cost in terms of performance. We thus believe that our profitability based scheme is a far more reasonable approach in terms of energy efficiency than any of the static schemes.

5.3 Thermal Analysis

Figure 5 depicts the transient thermal behavior for the base TLS system operating in normal-power mode and the profitability-based one for *parser*. As the figures reveal, *core0* consumes most of the power and as such has the highest average temperature. Note that even though the profitability-based scheme consumes more power on average than the base TLS, the transient behavior is much better. This results in a significant reduction in the temperatures observed, while it is also interesting that the thermal gap between the processors becomes smaller as well. We

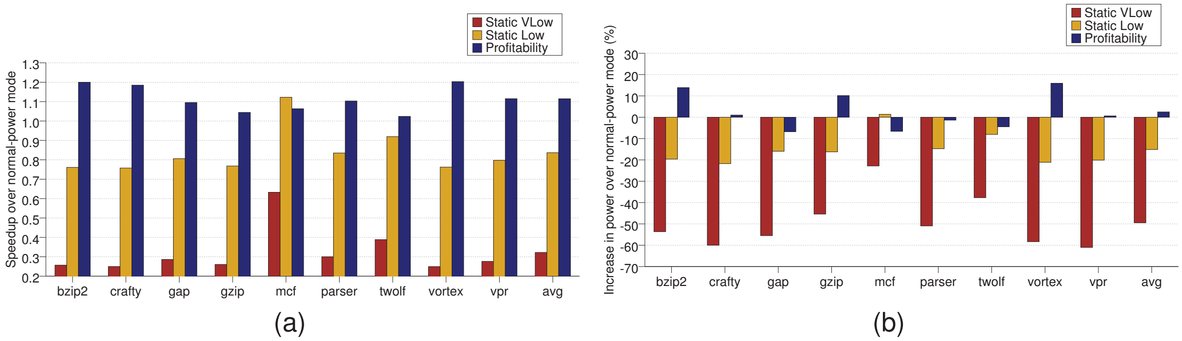

Figure 4. Comparing the three static power schemes, the very-low-power, the low-power and the normal-power mode, and our profitability-based one in terms of: (a) Speedup (normalized over normal-power mode). (b) Power Consumed (normalized over normal-power mode).

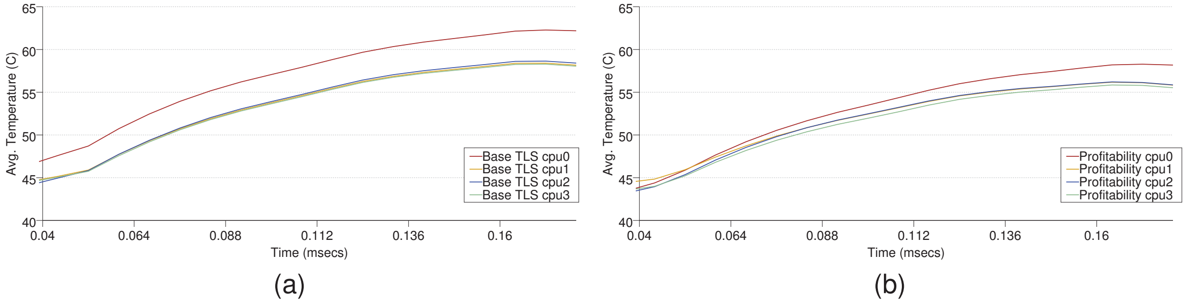

Figure 5. Thermal behavior per core for *parser* for: (a) Base TLS operating at normal-power mode. (b) The profitability-based scheme.

believe that these two figures present a strong motivation in employing the proposed form of adaptivity. The results for the rest of the applications exhibit similar, if not better behavior. Note that as the temperature of the two systems reaches steady state, it is much lower for the profitability-based scheme, than it is for the normal-power mode. The same trends hold for the rest of the benchmarks.

5.4 Effectiveness of the Squash Predictor

Figure 6 shows how our scheme works when it is guided by the Squash Predictor only. We only use the squash predictor so as to compare the effectiveness of the one we proposed in this paper and one that only uses memory addresses, without interference from the memory boundedness estimator. The left bar in each graph shows the percentage breakdown of the power modes for a memory address only predictor, while the right bars show the same for our memory address plus PC prediction scheme. In Figure 6(a) we see that for the threads that commit, the two predictors exhibit a similar behavior. However in Figure 6(b) we see that our scheme performs better than the memory address only scheme, and it is able to put more threads that will get squashed in low-power mode. What is interesting to point out, but is not shown in these graphs, is that when we also use the memory boundedness estimator, the memory-based only scheme performs much worse than ours. The reason for this is that it is far more sensitive in the thread ordering than the proposed predictor. Of course if adding five bits for each cache line is prohibitive for a specific design, our profitability-based scheme could still perform better than any static one even with the memory address only predictor.

5.5 Effectiveness of Memory-Boundedness Predictor

Figure 7 shows the ED for the profitability-based scheme with and without the memory-boundedness predictor. Because the only truly memory-bound application we have in our set of benchmarks is *mcf*, we expect the predictor to have

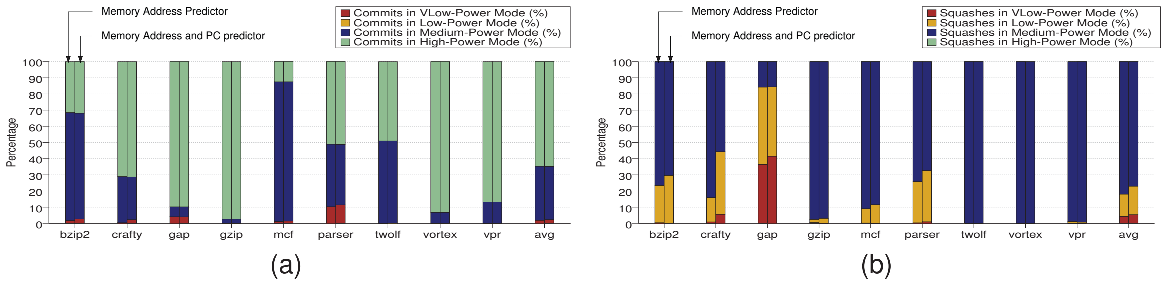

Figure 6. Guiding our allocation scheme using only a Squash Predictor (memory address only and our combined) for: (a) Threads that commit. (b) Threads that squash.

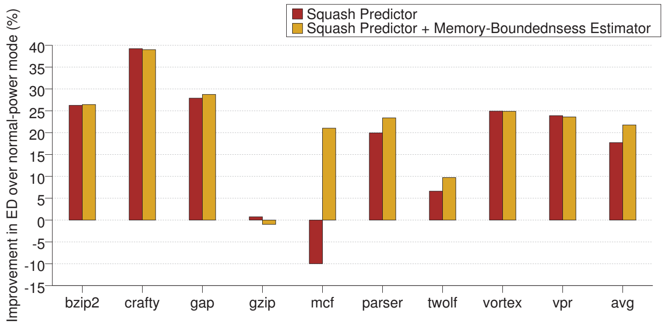

Figure 7. Normalized speedup over normal-power mode for the profitability-based scheme with and without the memory-boundedness predictor.

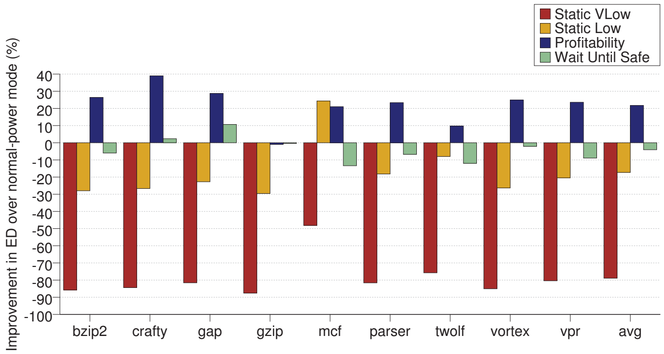

Figure 8. Normalized speedup over normal-power mode for low-power mode, high-power mode and our scheme where instead of putting non-profitable threads in low-power mode we stall them until they become safe.

an impact for this benchmark and not to affect significantly the rest. This intuition is supported by the depicted results. For the memory-bound application we can see that the addition of the predictor is fairly significant. For *gap* and *parser*, it helps albeit not significantly, while for the rest of the applications it has no effect. Note that because this graph compares ED, no change in the bars does not mean that the predictor does not have an effect. In fact this only means that the predictor is able to trade performance for power, while maintaining the same energy efficiency (or even improving it).

5.6 Stalling Non-Profitable Threads

A reasonable concern is that, if we can identify so accurately the threads that will get squashed, why not stall them until they become safe. As it can be seen from Figure 8, doing so results in a system with significantly lower performance than the normal-power one. The reasons for this are primarily twofold. Firstly, although our predictor is quite accurate it is far from perfect, and whenever it is wrong we lose performance due to TLP. Secondly, even threads which will eventually get squashed are not completely non-profitable. In fact, they can procure indirect performance benefits, either due to prefetching to the common L2 cache for safer threads, or due to warming up the branch predictors. Therefore, we argue that letting those threads run, albeit in low-power mode, is a sensible solution as it enhances the system's performance while consuming minimal power.

6 Discussion - Applying Profitability-Based Power Allocation to TM

In the previous sections we showed that allocating power according to profitability, provides significant benefits in terms of ED for TLS. Making speculative multithreading systems energy efficient is becoming increasingly important, especially for many-core systems like the ones we will have, according to projections made by both industry and academia. In fact better energy efficiency will allow more cores to operate at the same time and thus increase their throughput. We have shown how one can improve a state-of-art TLS system, despite the fact that it was already optimized by means of profiling (the POSH compiler used throughout this work optimizes for power as well).

Applying the proposed power allocation scheme to a TM system should be quite straight-forward. The only difference from a hardware perspective between TM and TLS is that under TM there is no implicit thread ordering. As such we cannot allocate more power to a specific thread based on it being safe. However, under TM systems there is usually a conflict resolution policy, which is different based on the TM flavor used. We can then leverage upon this mechanism to decide which is the "safe" thread (i.e., the one most likely to win in a conflict) and thus guide our scheme. The squash predictor and the memory boundedness estimator presented here can then be used in a similar fashion to the one described in previous chapters.

7 RELATED WORK

7.1 Related Work on TLS Systems

Thread-level-speculation has been previously proposed (e.g., [10, 13, 16, 23]) as a means to provide some degree of parallelism in the presence of data dependences. The vast majority of prior work on TLS systems has focused on architectural features directly related to the TLS support, such as protocols for multi-versioned caches and data dependence violation detection. All these are orthogonal to our work. In particular, we use the system in [19] as our baseline. The work in [20] argues that TLS systems can be more energy efficient than wide single threaded systems, but does not focus on improving the energy efficiency of TLS systems by using DVFS on non-profitable tasks.

7.2 Related Work on Power Allocation

The work most relevant to this scheme is the one in [25]. In that work there is only one core that is fixed in high-power mode and three that are fixed in low-power mode. Threads are *migrated* to the high power core when they are predicted to be critical. Predictions are made based on a task-level criticality predictor. We showed that with much simpler predictors one can achieve the same goals given per core voltage/frequency regulators. Recently [2] performed run-time adaptation based on criticality predictors. Although our paper shares the same ambitions, we not only cut down power but instead allocate it to the threads deemed to be profitable. An additional important issue we had to deal with is the fact that not all our threads commit their state (in contrast to the explicitly parallel applications used in [2]).

Per-core regulators like the one proposed in [26], have been demonstrated to be both fast and efficient. [12] showed that these regulators can be beneficial for fast architectural optimizations like ours. Our work assumes such regulators and it builds on top of work on synchronization among cores in different voltage/frequency isles, like the one in [5].

8 Conclusions

Speculative multithreaded systems like TM and TLS relieve the programmer/compiler from the tedious process of synchronizing explicitly parallel applications and of correctly parallelizing sequential ones. However, due to mispeculation, both systems are inefficient from a power perspective. In this paper we propose to classify threads into either profitable or non-profitable ones and allocate power resources accordingly.

When we applied our scheme to a state-of-the-art TLS system, we were able to achieve significant speedups with reasonable increases in the power consumed. More specifically, for a subset of the SPEC2000 Integer benchmarks, we showed that with only minimal hardware support, we were able to achieve improvements in the overall ED of up to 39.6% with an average of 21.2%.

REFERENCES

[1] S. Abedinpour, B. Bakkaloglu, and S. Kiaei. "A Multi-Stage Interleaved Synchronous Buck Converter with Integrated Output Filter in a 0.18um SiGe Process." In *Intl. Solid-State Circuits Conf.*, pages 2164-2175, February 2006.

[2] A. Bhattacharjee and M. Martonosi. "Thread Criticality Predictors for Dynamic Performance, Power, and Resource Management in Chip Multiprocessors." *Intl. Symp. on Computer Architecture,* pages 290-301, June 2009.

[3] D. Brooks, V. Tiwari, and M. Martonosi. "Wattch: a Framework for Architectural-Level Power Analysis and Optimizations." *Intl. Symp. on Computer Architecture,* pages 83-94, June 2000.

[4] D. M. Brooks, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J. D. Wellman, V. Zyuban, M. Gupta, and P. W. Cook. "Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors." *IEEE Micro,*, vol. 20, no. 6, pages 26-44, November 2000.

[5] T. Chelcea and S. M. Nowick. "Robust Interfaces for Mixed-Timing Systems with Application to Latency-Insensitive Protocols." *Design Automation Conf.*, pages 21-26, June 2001.

[6] M. Cintra and J. Torrellas. "Eliminating Squashes Through Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors." *Intl. Symp. on High-Performance Computer Architecture*, pages 43-54, February 2002.

[7] Intel Corp. "Intel Turbo Boost Technology in Intel Core Microarchitecture (Nehalem) Based Processors." http://download.intel.com/design/processor/applnots/320354.pdf.

[8] Intel Corporation. *Intel Core2 Duo Processors and Intel Core2 Extreme Processors for Platforms Based on Mobile Intel 965 Express Chipset Family Datasheet*, 2008.

[9] G. Dhiman, and T. S. Rosing. "Dynamic Voltage Frequency Scaling for Multi-Tasking Systems Using Online Learning." *Intl. Symp. on Low Power Electronics and Design*, pages 207-212, August 2007.

[10] L. Hammond, M. Wiley, and K. Olukotun. "Data Speculation Support for a Chip Multiprocessor." *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems,* pages 58-69, October 1998.

[11] P. Hazucha, G. Schrom, H. Jaehong, B. Bloechel, P. Hack, G. Dermer, S. Narendra, D. Gardner, T. Karnik, V. De, and S. Borkar. "A 233-MHz 80%-87% Efficiency Four-Phase DC-DC Converter Utilizing Air- Core Inductors on Package." In *Intl. Solid-State Circuits Conf.*, pages 838-845, February 2005.

[12] W. Kim, M. Gupta, G.-Y. Wei, and D. Brooks. "System Level Analysis of Fast, Per-Core DVFS Using On-Chip Switching Regulators." *Intl. Symp. on High-Performance Computer Architecture,* pages 123-134, February 2008.

[13] V. Krishnan and J. Torrellas. "Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor." *Intl. Conf. on Supercomputing,* pages 85-92, June 1998.

[14] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. "POSH: a TLS Compiler that Exploits Program Structure." *Symp. on Principles and Practice of Parallel Programming,* pages 158-167, March 2006.

[15] P. Macken, M. Degrauwe, M. Van Paemel, and H. Oguey. "A Voltage Reduction Technique for Digital Systems." *Intl. Solid-State Circuits Conf.*, pages 238-239, February 1990.

[16] P. Marcuello and A. González. "Clustered Speculative Multithreaded Processors." *Intl. Conf. on Supercomputing,* pages 365-372, June 1999.

[17] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. "Dynamic Speculation and Synchronization of Data Dependence." *Intl. Symp. on Computer Architecture*, pages 181-193, May 1997.

[18] Y. Panov and M. Jovanovic. "Design Considerations for 12-V/1.5-V, 50-A Voltage Regulator Modules." *Transactions on Power Electronics,* 16(6), November 2001.

[19] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas. "Tasking with Out-Of-Order Spawn in TLS Chip Multiprocessors: Microarchitecture and Compilation." *Intl. Conf. on Supercomputing,* pages 179-188, June 2005.

[20] J. Renau, K. Strauss, L. Ceze, W. Liu, S. Saarangi, J. Tuck, and J. Torrellas. "Thread-Level Speculation on a CMP Can Be Energy Efficient." *Intl. Conf. on Supercomputing,* pages 219-228, June 2005.

[21] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. "SESC simulator." *http://sesc.sourceforge.net*.

[22] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. "Temperature-Aware Microarchitecture." *Intl. Symp. on Computer Architecture,* pages 2-13, June 2003.

[23] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. "Multiscalar Processors." *Intl. Symp. on Computer Architecture,* pages 414-425, June 1995.

[24] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. CACTI 4.0. Tech. report, Compaq Western Research Lab., 2006.

[25] J. Tuck, W. Liu, and J. Torrellas. "CAP: Criticality Analysis for Power-Efficient Speculative Multithreading." *Intl. Conf. on Computer Design*, pages 409-416, October 2007.

[26] J. Wibben and R. Harjani. "A High Efficiency DC-DC Converter Using 2nH On-Chip Inductors." *Symp. on VLSI Circuits*, 2007.

[27] W. Wu, N. Lee, and G. Schuellein. "Multi-Phase Buck Converter Design with Two-Phase Coupled Inductors." *Applied Power Electronics Conference and Exposition*, March 2006.

[28] P. Xekalakis, N. Ioannou and M. Cintra. "Combining Thread Level Speculation, Helper Threads, and Runahead Execution." *Intl. Conf. on Supercomputing,* pages 410-420, June 2009.