ISBN: 978-1-4503-2130-3

URL: http://dx.doi.org/10.1145/2464996.2465012

Towards More Efficient Execution: A Decoupled Access-Execute Approach

Konstantinos Koukos, David Black-Schaffer, Vasileios Spiliopoulos, Stefanos Kaxiras
Department of Information Technology, Uppsala University
Lägerhyddsvägen 2, 752 37, Uppsala, Sweden
konstantinos.koukos@it.uu.se, david.black-schaffer@it.uu.se,
vasileios.spiliopoulos@it.uu.se, stefanos.kaxiras@it.uu.se

ABSTRACT

The end of Dennard scaling is expected to shrink the range of DVFS in future nodes, limiting the energy savings of this technique. This paper evaluates how much we can increase the effectiveness of DVFS by using a software decoupled *access-execute* approach. *Decoupling* the data *access* from *execution* allows us to apply optimal voltage-frequency selection for each phase and therefore improve energy efficiency over standard *coupled* execution.

The underlying insight of our work is that by decoupling access and execute we can take advantage of the memory-bound nature of the access phase and the compute-bound nature of the execute phase to optimize power efficiency, while maintaining good performance. To demonstrate this we built a task based parallel execution infrastructure consisting of: (1) a runtime system to orchestrate the execution, (2) power models to predict optimal voltage-frequency selection at runtime, (3) a modeling infrastructure based on hardware measurements to simulate zero-latency, per-core DVFS, and (4) a hardware measurement infrastructure to verify our model's accuracy.

Based on real hardware measurements we project that the combination of decoupled access-execute and DVFS has the potential to improve EDP by 25% without hurting performance. On memory-bound applications we significantly improve performance due to increased MLP in the access phase and ILP in the execute phase. Furthermore we demonstrate that our method can achieve high performance both in presence or absence of a hardware prefetcher.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: System architectures

Keywords

Task-Based Execution, Decoupled Execution, Performance, Energy, DVFS

ICS'13, June 10–14, 2013, Eugene, Oregon, USA.
Copyright 2013 ACM 978-1-4503-2130-3/13/06 ...$15.00.

1. INTRODUCTION

Power efficiency has become one of the most important design parameters for hardware, due to limited battery runtime on mobile devices and energy cost on data-center servers and supercomputers. Dennard scaling [2], however, can no longer provide us with constant power density per device, due to the exponential increase of leakage power at lower voltages. As a result, the effective voltage range available for dynamic voltage frequency scaling (DVFS) is expected to shrink in future technology nodes [8]. This undermines DVFS, which has been one of our most powerful techniques to reduce power consumption, due to its quadratic energy savings for at most linear performance degradation. Since the effectiveness of DVFS is threatened by the inability to significantly scale voltage, we must shift our attention to the non-linear relationship between frequency scaling and performance[1] in order to improve DVFS efficiency.

DVFS techniques try to exploit this non-linear relationship in memory-bound programs to reduce power consumption without affecting their performance [10]. This is possible because such programs spend much of their time waiting for memory, and are therefore insensitive to frequency changes. In such cases we can expect an improvement in power-efficiency metrics such as EDP or ED^2P from reducing the frequency. However this is only feasible for programs which are predominantly memory-bound. In most programs, although there are considerable opportunities to scale frequency while waiting for memory, we are unable to benefit from DVFS because memory operations are interspersed with *(or tightly-coupled to)* arithmetic computation, whose performance is tied to frequency.

In such programs, optimal DVFS could be achieved if on a cache miss we could instantly scale down the frequency and instantly scale up after the miss is resolved. Unfortunately, the transition latency for modern CPUs is prohibitive (2000 nsec for the hardware alone), which makes it impossible to apply DVFS at an instruction granularity. The common approach today is to apply a global frequency based on the overall application behavior, which is far from the optimal in many cases. A better approach would be to apply

[1] $Power \propto ACfV^2$ where C is the load capacitance, V is the supply voltage, A is the activity factor and f is the operating frequency. AC is referred as effective capacitance C_{eff}. The relationship between frequency (f) and performance highly depends on the application memory behavior and can range from highly correlated (computation-bound) to uncorrelated (memory-bound). This also creates a non-linear relationship between power consumption and performance.

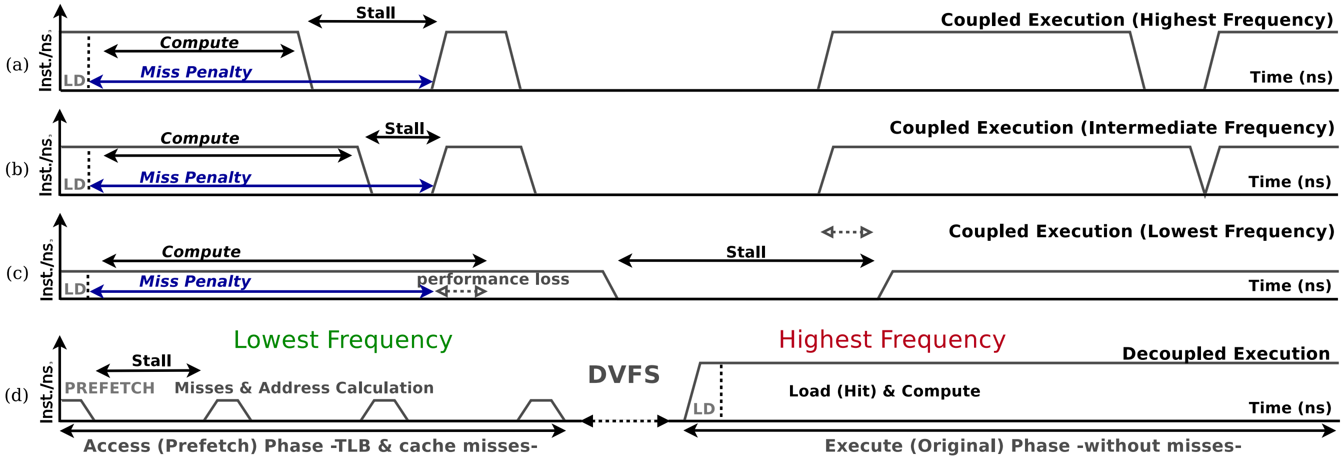


Figure 1: Example of coupled execution under different core frequencies (a,b and c) and decoupled execution (d)

DVFS at a finer granularity than the whole program but coarser than that of a few instructions. If program execution can be divided into distinct phases with homogeneous memory behavior, then the optimal frequency for each phase can be predicted separately [16]. The finer the division, the more effective the DVFS can be (in practice there is a limit due to the DVFS transition latency that adds overhead). Thus in any execution model where memory operations are tightly coupled to arithmetic computations we can only hope to exploit a fraction of the potential DVFS benefit.

To attack this problem, we propose a software *decoupled* access-execute (*DAE*) model [15] in combination with guided DVFS. We execute programs as a series of *tasks*, where each task is split into two fine grain phases: *access* (data prefetching) and *execute* (original computation) where the execute phase is scheduled for immediate execution after the access on the same core. The key idea is that decoupling data access from computation allows us to make different voltage-frequency decisions for each phase (access vs. execute). The access phase prefetches the data, which transforms most cache misses into hits for the execute phase, thereby improving performance of the latter. The access phase spends only a small fraction of its time computing addresses and most of its time waiting for data from memory. As a result it is not affected by core frequency, and we can therefore run it at minimum frequency to save power without hurting performance. In the execute phase most of the cache misses have been eliminated by the prefetching behavior of the access phase. Minimizing stalls on the execute phase makes highest frequency the best fit in terms of EDP.

Figure 1 demonstrates an execution example of coupled (*CAE*) vs decoupled (*DAE*) execution. Diagrams 1a, 1b and 1c show coupled execution under different frequencies, while 1d shows decoupled execution. In any execution model the main source of power inefficiency are the stalls because the processor is spending power waiting for memory. These diagrams show how coupled DVFS methods tries to eliminate stalls and thereby improve power efficiency. Stalls are created when computation cannot hide long miss penalties. Reducing the frequency slows computation down making it overlap more miss penalty and reduce stall length. Figure 1a shows coupled execution at the highest frequency. Although this execution is the fastest available, it is also the least power-efficient due to the energy wasted running the processor at full speed while waiting for DRAM.

DVFS can significantly improve efficiency by scaling down frequency-voltage to save power as shown in Figure 1b. Scaling down frequency will make the execution intervals between stalls longer but because miss penalty overlaps with execution the total execution time remains unaffected until they are fully overlapped. In the case shown in Figure 1c the execution has been unnecessarily slowed down because the execution interval becomes longer than the miss penalty. Additionally, although the stall from the first miss was successfully removed the second miss could not be overlapped by a long computation interval. This demonstrates the inability to choose a global optimal frequency when accesses and execute are interleaved which is common in nearly all applications.

Figure 1d demonstrates decoupled execution to solve this problem. In this case we prefetch data into private caches during the *access* phase and therefore avoid most cache and TLB misses in the *execute* phase due to this prefetching behavior. Decoupling can transform and split an interval of instructions in two phases: one purely memory-bound (access) followed by a purely compute-bound (execute). This allow us to adjust program behavior to DVFS granularity, exploiting more efficiently memory slack by reducing the frequency when waiting for memory and benefit from the future CPU DVFS capabilities.

Obtaining good performance on modern CPUs requires efficiently parallelized, and scalable, code. For this reason our work focuses on parallel workloads. An important question that arises is how to implement a decoupled access-execute model for parallel programs. For this, we turn to task-based programming models, which lead to an elegant solution for implementing DAE. In a task-based programming model, parallelism is expressed through tasks that can be scheduled independently. In our approach a task is defined as a function and because of this, we can easily replicate and transform each task into an access and an execute phase. We do this the simplest way possible: the access phase of a task is the task without any computation or data stores (e.g., just address calculation and loads), while the execute phase is the original unmodified task. Although this leads to redundant execution of all address calculations and memory access instructions, it requires minimal programmer or compiler effort.

The task-based model affords us considerable latitude in exploring decoupled access-execute. By controlling the input data size of the tasks we control the granularity of prefetch and DVFS. Thus, we can amortize the DVFS overhead with how much data we can prefetch into the cache during the access phase. We perform our experiments on state-of-the-art systems using accurate, fine-grain, power measurements taken directly from the processor power rails and project the results as if we had instantaneous per-core DVFS. We are restricted to this approach because state-of-the-art CPUs do not yet feature on chip voltage regulators [11] to enable low-latency, per-core DVFS [12]. Our work shows that decoupling access from execute can improve the effectiveness of DVFS and achieve performance comparable to CAE at max frequency and at the same time maintain optimal EDP.

2. RELATED WORK

The idea of decoupling access from execution was initially proposed by Smith [15] in 1982. In his approach the execution units were unaware of address calculation and were only capable of performing an arithmetic operation on the next available operands. The use of hardware prefetchers in modern architectures to hide memory latencies has a similar effect. Kamruzzaman et. al. [9] presented a technique to parallelize the memory accesses of single-threaded applications, using a decoupled approach with helper threads to speculatively prefetch data. The goal was to reduce execution time compared to the sequential execution with minimal compiler or programmer effort. This approach suffers from reduced efficiency compared to parallel execution because they parallelize only the memory accesses (up to 60% speedup using 4 cores), at the cost of linear power increase when enabling cores (up to 4x power consumption).

Similarly Ibrahim et. al. [6] demonstrate the benefit of a slipstream execution on multiprocessor systems. Keramidas et. al. [10] presented tools and techniques for efficient DVFS on coupled execution while Spiliopoulos et. al. [16] further improve that work and embed it in the Linux kernel, using performance monitoring unit (PMU) based models to detect and scale voltage-frequency on application phases. In our approach, instead of trying to adjust DVFS granularity to program behavior we modify program behavior to DVFS granularity.

3. METHODOLOGY

Our experimental setup consists of (1) the runtime system that handles parallel execution and profiling of workloads using coupled and decoupled execution, (2) applications with manually created access phases, (3) a power model to estimate per task-phase power consumption and simulate low-latency, per-core DVFS, and (4) a hardware infrastructure to verify the accuracy of the model.

As mentioned in Section 1 our method tries to exploit the non-linear relationship between frequency scaling and performance of memory bound applications (or phases). A primary prerequisite of our method is that we can reduce power consumption by reducing the core frequency without compromising the available bandwidth from the main memory (off-core). Thus, we can achieve optimal power savings without any performance degradation in memory-bound phases of the application. Figure 2 shows the aggregate bandwidth delivered from the main memory under different access patterns and CPU frequencies for Intel's state-of-the-art processors with the use of a parallel micro-benchmark we developed. Figure 2a shows that the aggregate bandwidth remains unaffected under different core frequencies, which suggests significant room for EDP improvement for memory bound phases.

The worst performing pattern in Figure 2 is *random*, which uses a random walk with distances mostly larger than a page. In addition to cache misses it also causes TLB misses and irregular accesses to memory pages. This pattern hardly achieves 6 GB/s aggregate bandwidth from all cores which is 85% lower than a *sequential* access pattern that can achieve up to 11 GB/s. The processor though has a peak bandwidth limit of 21 GB/s shared among all 4 cores. This is approached by a linear access micro-benchmark using the *prefetcht0* instruction, which achieves up to 80% of the peak bandwidth. Figure 2b shows the aggregate bandwidth under different number of threads. We observe that bandwidth scales sub-linearly up to 3 threads

3.1 Task parallel runtime system

The runtime system is responsible for the parallel execution, synchronization, scheduling, and load balancing of tasks. For our experiments, the runtime also profiles time and monitors hardware counter events per task using PAPI [13]. In a normal execution the runtime selectively collects statistics using sparse sampling for the IPC and the execution time of each task phase. This information is then used by a power model (discussed in section 3.3) that predicts the optimal execution frequency for each task phase based on previous execution knowledge of the same task. The runtime can then apply the predicted frequency using the *cpufreq utilities* [3] interface. Although the functionality to DVFS each task phase independently is implemented we cannot use it because current processors do not feature per-core DVFS and their DVFS transition latency is too high. We expect that this limitation will be addressed in future processors. Our goal is therefore to predict the benefits of applying DVFS to a decoupled program by using real hardware measurements and model the expected power and performance of per-core, low-latency DVFS.

In our runtime a task is a C/C++ function that can be executed asynchronously by any active core in parallel. Each task can have two phases: execute and (optionally) access, where each one of them is expressed as a different function. The runtime stores the function pointer(s) and the arguments of the function(s) internally and schedules the task for parallel execution. The task is executed as a single unit by the runtime calling the access phase (if available) and immediately after that the execute phase on the same core. The scheduling of the task is a runtime decision that tries to balance load across all cores. We also support manual binding of tasks to cores to enable application defined scheduling policies.

The runtime uses a single (main) thread that issues tasks to multiple queues that other (worker) threads poll. Each active thread has a private and a shared queue to store and schedule task descriptors. Load balancing is enabled by work stealing of tasks through shared queues, while private queues enforce local FIFO execution. The runtime supports two synchronization primitives: *barriers* for global synchro-

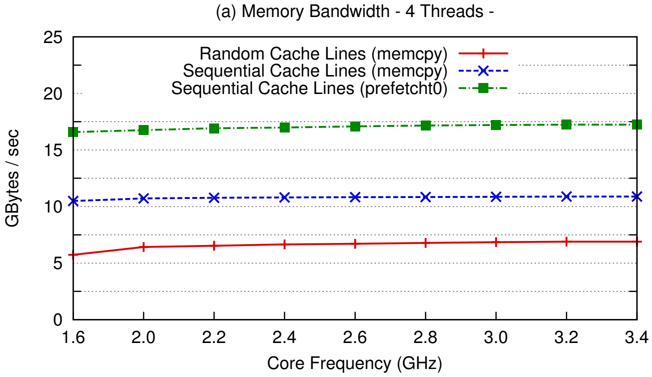


Figure 2: Measured memory bandwidth for Intel Sandybridge using DDR3 at 1333 MHz for: (a) 4 threads at different core frequencies, (b) different number of threads at highest core frequency. We need at least 3 threads to saturate bandwidth.

nization and *point to point* for synchronization across tasks. Synchronization primitives and internal memory allocators use lock free algorithms to reduce overhead. The runtime is designed to have very low execution overheads with total overhead per task of only 400 core cycles on Intel Sandybridge CPU.

3.2 Generating access phase

The access phase serves to prefetch the data for the execute phase into the cache. To create the access phase for a task we remove all stores and arithmetic calculations and keep only the loads and address calculation required for the loads. Eliminating stores guarantees that there will be no side effects from the access phase. To optimize prefetching we replace load instructions with the builtin *prefetcht0* x86 instruction [7] because it does not stall instruction retirement and does not require a destination register. This explains why *prefetcht0* can achieve higher bandwidth over *loads* as shown in Figure 2. The access phase can prefetch both loads and stores into the cache because both are memory accesses. However only loads can stall the pipeline which makes them performance critical. Stores are less likely to create stalls as they are buffered and therefore we don't prefetch them.

In our implementation we have to perform the address calculation once for the access phase and once again for the execute phase. The impact of this is to execute extra instructions compared to the initial coupled execution. In terms of performance, the duplicate address calculation is overlapped with the long latencies of prefetch misses thereby reducing their impact. From an energy perspective increasing the number of instructions executed on the core results in a direct power increase subject to the linear equations shown in Figure 3. In practice the IPC of the access phase is less than 0.5 so it can affect power significantly less than frequency scaling but still remains a source of inefficiency. More sophisticated splitting of tasks to access and execute phases can reduce or even eliminate redundant address calculations and therefore improve our results but it is left for future work.

3.3 Power Model

The fine-granularity of our approach (tasks range from a few microseconds up to a millisecond) and the overlapping of access and execute phases on different cores, prevents us from directly measuring per-task power consumption on the hardware. Current power measurement infrastructures (both integrated on-chip power sensors and external measurement hardware) provide sampling periods on the order of milliseconds. Furthermore, current hardware does not provide per-core DVFS, with some machines having all cores under a single clock domain (Intel) and others with multiple clock domains but a single voltage domain (AMD). These hardware restrictions are expected to change with the introduction of on-chip voltage regulators [12] in future generations to support nano-scale DVFS [11].

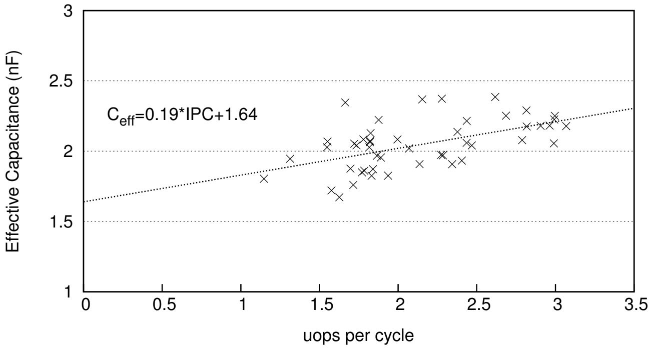


Figure 3: Correlation between effective capacitance and uops executed per cycle for Intel Sandybridge processor. Methodology described in [16]

To overcome these limitations, we employ an IPC-based power model to estimate power consumption. This approach enables us to estimate power consumption of any execution interval, however short in duration. Our model is built by measuring per-core dynamic power, and static power for the whole processor (per-core static plus shared L3 power) using multiple instances of the same workload running independently on different cores. To measure the power consumed by the processor we use the measurement infrastructure described in [17]. We express static power P_{static} as linear function of f, V for each number of active cores. To measure the static power we use a micro-benchmark with an infinite loop and use *nanosleep* system call to run at an IPC close to zero, while this minimum activity prevents the core from entering a deep sleep state.

For the dynamic power (similar to [16]), we assume that effective capacitance C_{eff} correlates linearly with the micro-ops executed per cycle. As shown in Figure 3, we derive a linear approximation to the effective capacitance by measuring the power and IPC across SPEC2006 [4]. Having f, V and C_{eff} (as a function of IPC) we estimate per-core dynamic power using $P_{dynamic} = C_{eff} f V^2$ equation. In a parallel execution the total processor power is the sum of the dynamic per-core power plus the total static power of the processor.

To validate the accuracy of our model we compare the total power estimate against the measured value for each of the SPEC2006 benchmark suit (on all frequencies and number of cores). For the SPEC2006 benchmark suit, the average error is 3.1% while the maximum error is 9.1%[2]. For the applications we use to evaluate DAE, the average and maximum errors for estimating the total energy is 4.5% and 7% respectively. More accurate performance counter-based models, such as those proposed by Spiliopoulos et. al. in [17], could be used to improve accuracy. We have chosen this model as it provides sufficient accuracy and requires just a single performance counter event (for the instructions) which is available in most architectures.

3.4 Putting it all together

The methodology described above for estimating power consumption is necessary for quantifying the energy benefits of our DAE programming model. Using this approach we are able to overcome two key limitations of current hardware, and thereby accurately estimate per task-phase energy. These two limitations are: 1) the inability to measure per task-phase energy consumption due to its fine granularity and the overlapping of different tasks access and execute phases on different cores, and, 2) the inability of current hardware to provide per-core, low-latency DVFS.

State of the art processors feature high overhead for frequency and voltage switching (between 8 and 10 usec[3]), which is prohibitive for applications that require fine-grain tasks. In our workloads the average access-phase duration per task is between 3 and 16 usec as shown in Table 1. Although our runtime is able to DVFS each phase of a decoupled access-execute task as discussed in Section 1, this overhead and the lack for per-core DVFS support prevents us from using this at runtime on current hardware.

To overcome the above restrictions and project our model benefits to future processors we disable per-phase DVFS at runtime and develop a model (Algorithm 1) to simulate zero-overhead, per-core DVFS. For that we execute our workloads both for CAE and DAE at all available frequencies and collect detailed *time* and *IPC* statistics as provided by the runtime. For time statistics we measure T_{access}, $T_{execute}$ per task and the total execution time of the application. For the IPC statistics we measure IPC_{access}, $IPC_{execute}$ respectively. Additionally we measure the number of tasks executed. Providing both *time* and *IPC* to our power model we can estimate per-phase energy for any voltage-frequency combination. Using profiling information for performance and the results of the power model for energy we can accurately estimate the benefit of our technique for future processors with low-latency, per-core DVFS. We also collect coarse grain power measurements and compare the overall power estimate of the model against them to verify its accuracy.

Algorithm 1 DAE overall methodology

```
for all $f, V$ do
    - Measure average $Time_{access}$ $(AVG(T_A))$
    - Measure average $Time_{execute}$ $(AVG(T_X))$
    - Measure average $IPC_{access}$ $(AVG(IPC_A))$
    - Measure average $IPC_{execute}$ $(AVG(IPC_X))$
    - Measure total execution time ($T_{total}$) and total energy
    ($E_{measured}$) and verify model accuracy
end for
Naive policy:(Access=$f_{min}$, Execute=$f_{max}$)
$E_A$ = PowerModel($AVG(IPC_A^{f_{min}})$)*$AVG(T_A^{f_{min}})$
$E_X$ = PowerModel($AVG(IPC_X^{f_{max}})$)*$AVG(T_X^{f_{max}})$
OptEDP policy:
$E_A$ = PowerModel($AVG(IPC_A^{f_{opt}})$)*$AVG(T_A^{f_{opt}})$
$E_X$ = PowerModel($AVG(IPC_X^{f_{opt}})$)*$AVG(T_X^{f_{opt}})$
```

The goal of the decoupled access-execute model is to provide optimal EDP efficiency through DVFS with minimal performance degradation. For that we implement two DVFS policies: (1) *naive*, where the access phase always runs at the lowest frequency and the execute phase always runs at the highest frequency, and (2) *optEDP* in which the runtime intelligently adjusts the frequency of each phase based on IPC and power model to obtain the best EDP. We expect that the *naive* approach will keep total execution time very close to the execution time of coupled execution at highest frequency because the access phase performance is not affected by DVFS and the execute phase runs at the highest frequency. The EDP improvement for this policy is limited to the energy we can save by running the access phase at lower frequency. In memory-bound applications with irregular memory access patterns, the total execution time can be reduced over coupled execution due to the accuracy of software prefetching and the increased MLP[4] in the access phase, which leads to an additional EDP improvement.

In the *optEDP* policy the runtime tries to intelligently DVFS each phase based on the power model, thus total performance may be reduced because the execute phase is allowed to run slower in order to improve overall EDP. Although in the general case stores can not stall the execution and the address calculation runs on such a low IPC that is not affected by core frequency, there are some rare cases in which the address calculation is very complex and a slightly higher (than the lowest) frequency could improve EDP, or stores have very irregular access pattern followed by TLB misses and therefore a slightly lower (than the highest) frequency could also improve EDP. As a baseline for our experiments we use the original coupled execution at highest frequency.

[2] The error shown in Figure 3 refers only to dynamic power. Including static power to estimate total power can improve accuracy since it is deterministic.

[3] The DVFS transition latency is 8 usec on AMD (Bulldozer) using *powernow-k8* driver and 10 usec for Intel (Sandybridge and Nehalem) using *acpi-cpufreq* driver

[4] The use of *prefetch*{*t0-t2*} x86 instructions over normal loads can increase effective bandwidth as explained in section 3.2 and shown in Figure 2

Table 1: Application characteristics and task configuration. $\%T_A$: Average fraction of the application's execution time spent in the access phase and T_A(usec): Average duration of the access phase.

Application	% T_A	T_A(usec)	Task Size(KB)	Access Pattern
Cholesky	5.6	3.09	16 - 48	Tiled
LU	3.3	2.94	16 - 48	Tiled
FFT	10.5	15.89	16 - 128	Butterfly
LBM	51.0	7.94	38	Stream Collide
CG	43.0	3.38	32	Indirection
LibQ	56.9	2.94	8 - 16	Regular
Cigar	66.2	5.88	32	Indirection

4. EVALUATION

4.1 Evaluation framework

For the evaluation framework we ported and optimized using SSE three SPLASH2 [18] kernels: FFT, LU and Cholesky, two SPEC CPU2006 [5] applications: 470. lbm and 462. libquantum, CG from the NAS parallel benchmarks [14] and CIGAR [1]. This set of benchmarks covers a wide range of memory access patterns and behaviors, from compute-bound (LU, Cholesky and FFT) to memory-bound (CIGAR and libquantum). CG and LBM have an intermediate behaviour. Table 1 lists the behavior and configuration used for the evaluation of the applications. Although there is a large variation in the duration of the execute phase across different applications (e.g., 87 usec for Cholesky and 4.5 usec for CG), for the access phase the duration is mostly affected by the data size and only slightly by the access pattern.

4.2 Performance/Energy Evaluation

Increasing parallelism in any class of applications can make the application more memory-bound as it increases the rate of requests to DRAM resulting in longer stalls per request due to memory bandwidth saturation. This limits program scalability. Our technique relieves the execute phase from these stalls by prefetching data in the access phase. Since in the access phase we use *prefetch* vs. *load* instructions, for memory-bound applications we measure an average speedup of 6.9% and 8.4%, for 2 and 4 threads respectively. The improvement in total execution time directly leads to a reduction in EDP. Additionally, decoupled execution gives us the opportunity to lower the core frequency in the access phase in order to further improve power efficiency.

Figure 4 demonstrates the benefit of decoupling using CG as an example. In this figure we show time (left graph) and energy (right graph) as a function of frequency and core count. In each graph we show on the left half coupled access-execute (CAE) at various frequencies (grouped into a cluster) and core counts (3 clusters using 1,2 and 4 cores) and on the right half decoupled access-execute (DAE) in a similar manner. The difference between CAE and DAE is the following: while the whole program in CAE runs at a specific frequency f_k, for DAE this holds only for its execute phase while the access phase runs at a constant minimum frequency f_{min}. This gives rise to the following interesting properties:

Time (figure 4 left): At f_{min}, CAE and DAE exhibit almost same total time. As we scale frequency higher, time in CAE and the execute phase of DAE shrinks. But time in the access phase of DAE remains constant, since the access phase is always executed at f_{min}. Yet, the total time of DAE closely tracks (or even improves on) the total CAE time. This is because memory accesses are *inelastic* to processor frequency scaling regardless of whether they are tightly coupled and embedded within execution or decoupled and separated from it. In other words, waiting for memory in CAE and DAE costs the same and is not affected by scaling the frequency. As a result, scaling frequency in the access phase only marginally affects its time.

Energy (figure 4 right): Energy in CAE is proportional to Tf_kV^2, for its whole execution, resulting in a steep energy increase when f_k increases. However, a significant part of DAE execution, the access phase, *has a constant energy*, because it runs at constant frequency (f_{min}), costing significantly less than the corresponding CAE part that is interspersed throughout CAE execution. For DAE the total energy is given by $E_A^{f_{min}} + E_X^{f_k}$ and it can be modelled with the methodology described in Algorithm 1. If we were to scale frequency in the DAE access phase in sync with the execute phase we would observe the same steep increase in energy as in the CAE case. It is this difference in the energy increase, and the fact that access phase time is unaffected by frequency, that gives the decoupled-access execute model its advantage.

4.2.1 *Understanding Performance*

The result of DAE is to eliminate cache misses from the execute phase, thereby improving its ILP and allowing it to run with significantly reduced stall time in the pipeline. This makes the highest frequency the more efficient selection in terms of EDP for this phase. On the other hand, the access phase is memory-bound which makes it performance intensive to core DVFS, and therefore the lowest frequency (or one close to that depending on the complexity of address calculations) can be the optimal selection. Ideally the total execution time for the *naive* DAE policy would be very close to the original at max frequency.

Figure 5 verifies this assumption by showing the performance degradation caused by DVFS on a coupled execution and the benefit of decoupling access from execute using both *naive* and *optEDP* policies. Figure 5a shows the results with hardware prefetcher enabled while figure 5b shows the impact of turning off the hardware prefetcher on Intel's Sandybridge processors. Normalization is performed using the execution time of each application at highest frequency with the hardware prefetcher enabled as baseline. In Figure 5a we observe that the *naive* DAE without a hardware prefetcher performs similarly, and in the cases of memory-bound applications significantly better, than a coupled execution at the highest frequency.

The *optEDP* policy chooses the frequency that will achieve the optimal EDP for each phase based on a brute force search. The performance of this policy depends highly on the application. It performs better when there are complex address calculations in the access phase so that the optimal EDP for that phase is achieved at a frequency slightly higher than the minimum frequency. It performs worse than naive when the optimal EDP for the execute phase is at a frequency lower than the maximum. This happens in applications where writes are highly coupled to the execute phase and able to stall the execution.

When disabling the HW prefetcher we observe that coupled execution suffers from significant performance degra-

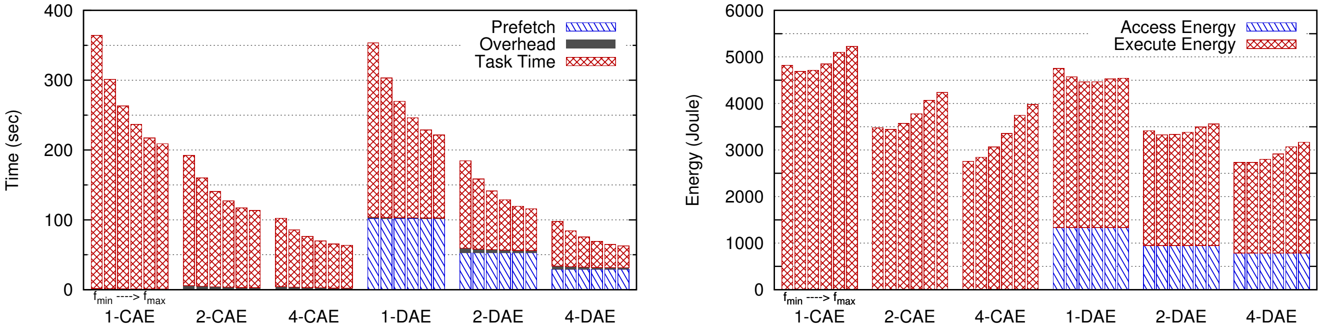
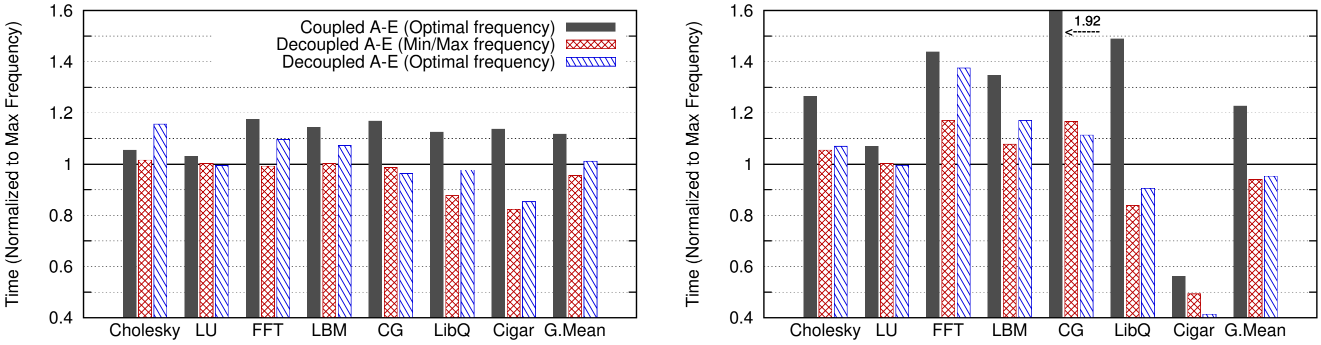


Figure 4: The impact of core frequency on execution time and energy. CG execution using 1,2 and 4 threads. For CAE we use frequencies from f_{min} (1.6 GHz) to f_{max} (3.4 GHz) with a step of 400 MHz while for DAE we use f_{min} for the access phase and f_{min} to f_{max} with a step of 400 MHz for the execute phase.



(a) with hardware prefetcher

(b) without hardware prefetcher

Figure 5: Execution runtime of CAE and DAE for Intel Sandybridge using 4 threads. Baseline is coupled execution (CAE) at maximum frequency with the hardware prefetcher enabled.

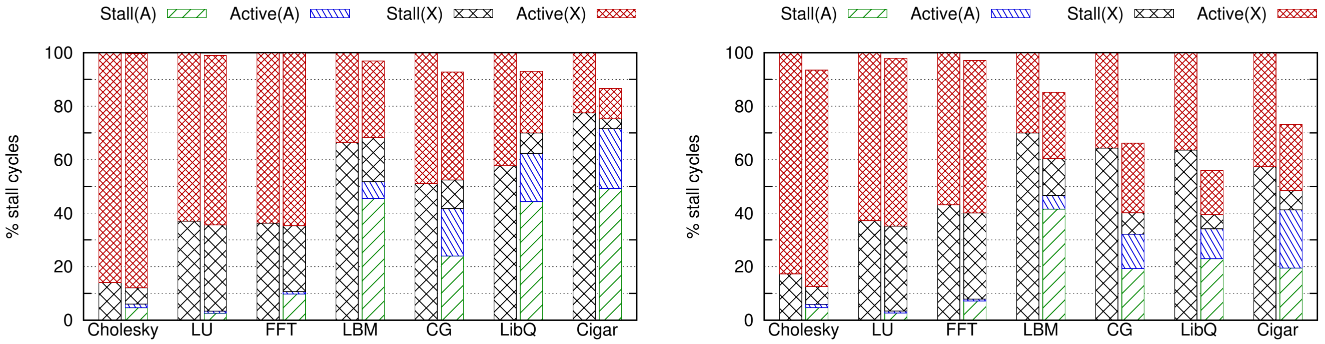
dation. An exception to this is *cigar* where the hardware prefetcher cannot detect its irregular pattern and therefore consumes bandwidth fetching unnecessary data which reduces performance. DAE performance on the other hand is less susceptible to misbehavior of the hardware prefetcher because the access phase accurately prefetches the data. With the exception of cigar, all other applications perform better when DAE is applied on a machine with hardware prefetcher, mostly because the access phase of DAE can provide hints to the hardware prefetcher and can therefore improve its accuracy.

4.2.2 Understanding Stalls

The key idea of decoupling is to move the memory slack from computation (execute phase) to a separate phase (access) that we can run at lower frequency. Memory slack is defined as the time that execution units are stalled due to off-chip misses. To evaluate how effectively we decouple real applications we need to measure slack in the initial program. For that we need to be able to measure stall cycles in the pipeline due to LLC misses. Unfortunately there is no such hardware counter. Another approach is to use a miss penalty approximation on all cache levels but it is rather inaccurate due to speculative, out-of-order execution. To overcome these limitations we use a combination of available hardware counter events, including stall cycles and hit ratio at each level of the cache.

Figure 6 shows the percentage of stall cycles in the dispatch unit for CAE and the two phases of DAE. These results can (a) characterize how memory-bound each application is and (b) how much of the stall time from coupled execution can be moved to the access phase in decoupled execution. Using Figure 6 we can characterize Cholesky, LU and FFT as compute-bound because they spend most of their execution time in coupled execution keeping the pipeline active while the rest can be characterized as memory-bound. We observe that for the compute-bound applications DAE eliminates only a small amount of the stall existing in the coupled execution. This is because these applications are optimized using SSE which creates pipeline stalls that are not associated with memory accesses.

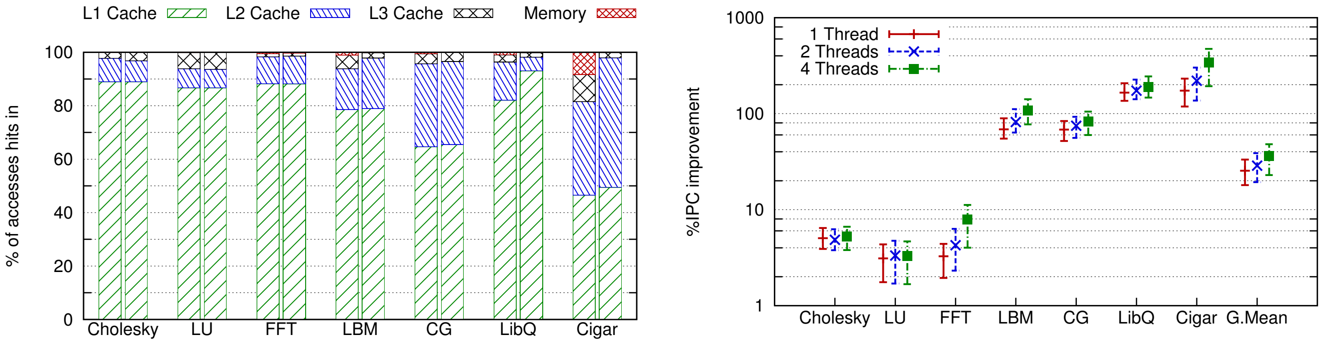
For memory-bound applications we observe (a) small overall speedup of DAE over CAE and (b) significant amount of stall time moved from CAE to DAE access phase which is the reason for the energy savings achieved. This supports our initial hypothesis that decoupling accesses and scheduling them ahead of the computation can significantly reduce stalls. The percentage of time spent on address calculation (*active(A)*) is relatively larger for memory-bound applications. This can be either because the task duration on these applications is significantly smaller than in compute-bound



(a) with hardware prefetcher

(b) without hardware prefetcher

Figure 6: Percent pipeline stalls for CAE (left bar) and DAE (right bar) on Intel Sandybridge with 4 threads



(a) Hit ratio improvement of DAE using 4 threads. Right bar: DAE execute phase. Left bar: coupled execution (CAE).

(b) IPC improvement of DAE execute phase over coupled execution (CAE) for Intel Sandybridge, across all frequencies.

Figure 7: Hit ratio and IPC improvement of decoupled over coupled execution models with hardware prefetcher enabled.

applications (LibQ) or because the access pattern is more irregular (cigar, CG). The key observation for memory-bound applications is that the overall speedup derives from the reduction of the non-stall time of execute phase while the total stall time including address calculation on the access phase in DAE is higher compared to the total stall time in CAE. Figure 6b shows the benefit of DAE without a hardware prefetcher. In this case we observe that the total stall time of DAE vs. CAE is reduced on every application, explaining the performance improvement shown in Figure 5b.

4.2.3 *Understanding cache behavior*

The accurate software prefetching of the access phase improves cache behavior of the execute phase for DAE applications. Figure 7a shows the impact on hit rate from decoupling for each application. Decoupling eliminates offchip traffic during the execute phase in all applications. In memory-bound applications it also manages to reduce offcore traffic by better utilizing private caches. Therefore for LibQ, LBM, CG and cigar we observe that a significant proportion of L2 misses have been eliminated. Performance improvements come primarily from the reduction of LLC misses because their penalties cannot be hidden by out-of order execution.

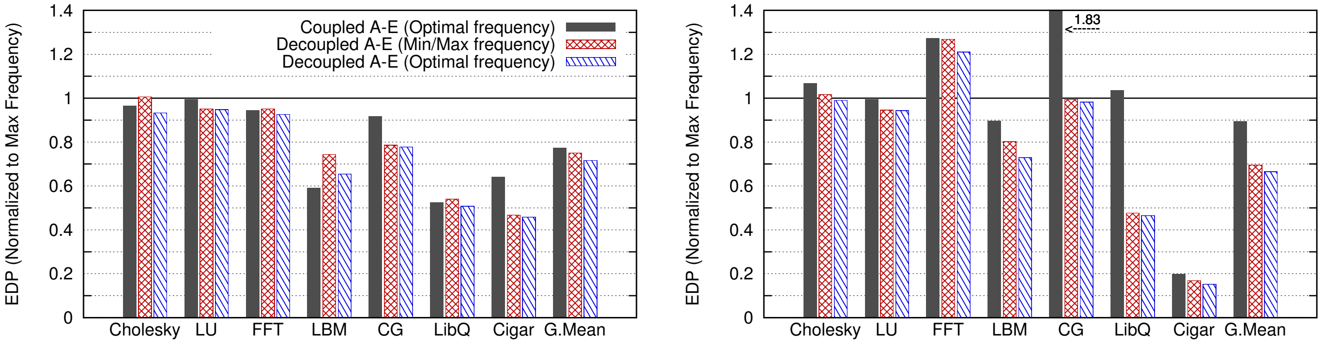
4.2.4 *Improving ILP*

In the previous section we have shown that DAE significantly improves the cache hit ratio of memory bound applications and eliminates most of the LLC misses from the execute phase. This reduces stall time and increases the IPC of this phase as shown in Figure 7b. For compute-bound applications such as LU, Cholesky, and FFT, we observe only a slight increase in IPC followed by a small performance improvement. For memory-bound applications, where we eliminate more LLC misses and reduce unnecessary off-chip traffic, we observe an IPC improvement ranging from 100% to 500%. This explains the performance gain of the execute active time, shown in Figure 6.

4.2.5 *Improving MLP*

The two previous sections explain the performance benefit of the execute phase. For the memory bound applications, the prefetch nature of the access phase can also improve memory-level parallelism (MLP). This is because in coupled execution, issuing of loads is limited by the size of the reorder buffer and the number of independent loads present in the reorder buffer. In decoupled execution, it is the prefetching queues (which are significantly larger than the reorder buffer) that limit MLP. Moreover, in DAE execution function units are fully available for performing address calculations, which also contributes to the increase of MLP.

Increased MLP contributes towards the increase of bandwidth utilization (as shown in Figure 9). Figure 2a shows that there is a potential of 30% higher bandwidth by using *prefetch* instructions over *loads*. To measure the bandwidth demands of our workloads we use performance counters and



(a) with hardware prefetcher

(b) without hardware prefetcher

Figure 8: EDP of CAE and DAE for Intel Sandybridge using 4 threads, normalized to CAE at maximum frequency

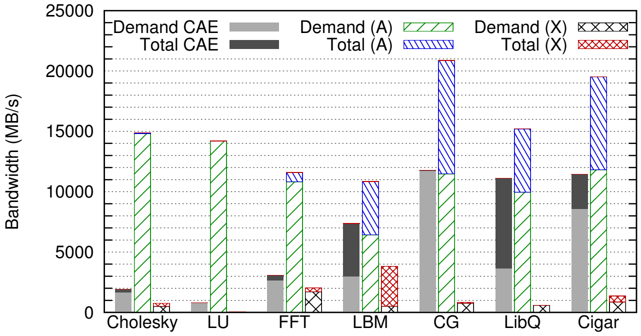


Figure 9: Application bandwidth on CAE and DAE models on Intel Sandybridge. Demand bandwidth does not include bandwidth generated by the hardware prefetcher while total includes it. Left bar shows CAE. Middle and right bars shows access (A) and execute (X) phases of DAE.

measure off-chip requests. Figure 9 shows the total traffic generated in each phase of each task on both CAE and DAE executions. “Demand” is the traffic generated by *LLC misses* and *software prefetches* while “total” includes the traffic generated also by the hardware prefetcher. Each triplet of bars corresponds to an application. The leftmost bar refers to coupled execution while the other two refer to access “(A)” and execute “(X)” phases of DAE respectively. The increase in traffic indicates that the access phase of DAE operates at a much higher MLP compared to CAE.

4.2.6 Understanding Power Efficiency

Our method improves EDP by improving both *execution time* (due to ILP improvement in the execute phase and MLP in the access phase) and *power* due to DVFS in the access phase. Figure 8 shows the EDP benefit of decoupled over coupled execution. In compute-bound applications there is limited opportunity for EDP improvement in the range of 10% but both coupled and decoupled executions manage to deliver improvements. The advantage of DAE is that it delivers this EDP benefit without any performance degradation. For the memory-bound applications DAE goes further and improves EDP by more than 50% in some cases. On CG and Cigar there is major advantage of DAE over CAE of 15%, mostly due to the performance gain.

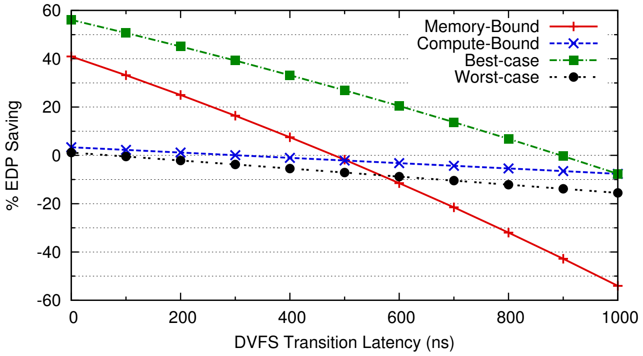


Figure 10: Simulation of future DVFS transition latency using 4 threads for the average of memory bound vs compute bound applications

On libquantum both CAE and DAE perform similarly while on LBM, CAE delivers better EDP than DAE. The reason is that decoupling helps only with the loads whereas LBM is constrained by stores that are coupled with computation in the execute phase which is amenable to DVFS. Thus, the *optEDP* DAE policy in Figure 8a improves EDP and approaches the optimal CAE. A comparison of Figure 8a with Figure 8b shows a significant contribution of the hardware prefetcher to the EDP improvement of almost 15% in coupled execution. Our method does not rely on the presence of a hardware prefetcher and does not conflict with it, therefore we observe that in both cases we obtain an average EDP improvement of 25%.

5. TOWARDS FUTURE PROCESSORS

As we explained in Ssection 3.4 our runtime is able to DVFS each phase but the benefit is limited, mostly because of the inability of per core DVFS on current processors. To overcome this limitation we modelled zero overhead, per-core DVFS. For that we use detailed time profiling measured by our runtime on all possible core frequencies hardware provides. We measure the number of tasks executed per core and the average duration of each phase (access and execute). Additionally we measure the overhead that the runtime itself creates for scheduling and any possible idle time created by application imbalances.

Having the number of tasks and the duration of each phase allows us to simulate any DVFS transition overhead by simply adding that overhead as shown in Figure 10. For the average of both memory and compute-bound applications we observe that we lose EDP benefit when DVFS transition latency overhead is larger than 500 nsec. Memory-bound applications are more promising for EDP benefit over compute-bound that are also less affected by DVFS transition overhead. The reason is that the execute phase of compute-bound applications is significantly larger than the access phase with the DVFS overhead. Figure 10 also shows the best (cigar) and worst (cholesky) applications behavior. For the best application we observe that we lose the EDP benefit of decoupled DVFS close to 1 usec which is double than the average, while for the worst application we have very limited potential for EDP improvement but also small impact of DVFS transition latency.

6. CONCLUSIONS

In this work we explored the potential of DVFS for decoupled access/execute applications in a task-based parallel environment. For applications with irregular memory accesses decoupling outperforms coupled execution both in terms of performance and power efficiency. For the computation-bound and moderately memory-bound applications, DAE can achieve equal EDP improvements to coupled execution at optimal frequency but without the trade-off of reduced performance. Using models to predict optimal EDP per task phase at runtime we can adjust the application performance and EDP dynamically. We have shown that by decoupling access from execute, a machine with low latency per-core DVFS could achieve on average 25% EDP reduction without sacrificing performance.

7. ACKNOWLEDGEMENTS

This work is supported, in part, by the Swedish Research Council UPMARC Linnaeus Centre and the EU Project: LPGPU FP7-ICT-288653.

References

[1] Cigar - case injected genetic algortihm. http://ecsl.cse.unr.edu/.

[2] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256 – 268, oct 1974.

[3] M. Dongili. cpufreq utilities. http://www.kernel.org/pub/linux/utils/kernel/cpufreq/.

[4] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, Sept. 2006.

[5] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, Sept. 2006.

[6] K. Ibrahim, G. Byrd, and E. Rotenberg. Slipstream execution mode for cmp-based multiprocessors. In *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 179–190, 2003.

[7] Intel. Intel ® 64 and ia-32 architectures optimization reference manual, pp.366-369, 2012.

[8] ITRS. Design 2011 edition, 2011.

[9] M. Kamruzzaman, S. Swanson, and D. M. Tullsen. Inter-core prefetching for multicore processors using migrating helper threads. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 393–404, New York, NY, USA, 2011. ACM.

[10] G. Keramidas, V. Spiliopoulos, and S. Kaxiras. Interval-based models for run-time dvfs orchestration in superscalar processors. In *Proceedings of the 7th ACM international conference on Computing frontiers*, CF '10, pages 287–296, New York, NY, USA, 2010. ACM.

[11] W. Kim, D. Brooks, and G.-Y. Wei. A fully-integrated 3-level dc-dc converter for nanosecond-scale dvfs. *Solid-State Circuits, IEEE Journal of*, 47(1):206 –219, jan. 2012.

[12] W. Kim, M. Gupta, G.-Y. Wei, and D. Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 123 –134, feb. 2008.

[13] P. J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[14] NASA. Nas parallel benchmarks, 1999. http://www.nas.nasa.gov/publications/npb.html.

[15] J. E. Smith. Decoupled access/execute computer architectures. *SIGARCH Comput. Archit. News*, 10(3):112–119, Apr. 1982.

[16] V. Spiliopoulos, S. Kaxiras, and G. Keramidas. Green governors: A framework for continuously adaptive dvfs. In *Proceedings of the 2011 International Green Computing Conference and Workshops*, IGCC '11, pages 1–8, Washington, DC, USA, 2011. IEEE Computer Society.

[17] V. Spiliopoulos, A. Sembrant, and S. Kaxiras. Power-sleuth: A tool for investigating your program's power behavior. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, pages 241 –250, aug. 2012.

[18] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, New York, NY, USA, 1995. ACM.