Towards Power Efficiency on Task-Based, Decoupled Access-Execute Models

Konstantinos Koukos, David Black-Schaffer, Vasileios Spiliopoulos, Stefanos Kaxiras
{*konstantinos.koukos,david.black-schaffer,vasileios.spiliopoulos,stefanos.kaxiras*}@it.uu.se

***Abstract*—This work demonstrates the potential of hardware and software optimization to improve the effectiveness of dynamic voltage and frequency scaling (DVFS). For software, we *decouple* data prefetch (*access*) and computation (*execute*) to enable optimal DVFS selection for each phase. For hardware, we use measurements from state-of-the-art multicore processors to accurately model the potential of per-core, zero-latency DVFS. We demonstrate that the combination of decoupled access-execute and precise DVFS has the potential to decrease EDP by 25-30% without reducing performance.**

The underlying insight in this 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. For the memory-bound access phase, where we prefetch data into the cache from main memory, we can run at a reduced frequency and voltage without hurting performance. Thereafter, the execute phase can run much faster, thanks to the prefetching of the access phase, and achieve higher performance. This *decoupled* program behavior allows us to achieve more effective use of DVFS than standard *coupled* executions which mix data access and compute.

To understand the potential of this approach, we measure application performance and power consumption on a modern multicore system across a range of frequencies and voltages. From this data we build a model that allows us to analyze the effects of per-core, zero-latency DVFS. The results of this work demonstrate the significant potential for finer-grain DVFS in combination with DVFS-optimized software.

I. Introduction

Designing for power efficiency is necessary for all devices from mobile to server. Due to the exponential increase of leakage at lower voltages , Dennard scaling [1] no longer provides constant power density per device. As a result, the effective voltage range available for DVFS is expected to shrink in future technology nodes [2]. The ability to dynamically scale both frequency and voltage, however, is a fundamental technique for reducing power consumption, promising quadratic power savings for at most linear performance degradation. One promising approach to prolonging DVFS's effectiveness is to exploit the non-linear relationship between frequency scaling and performance that arises when the processor is stalled waiting for memory.

It is well known that we can reduce CPU frequency in memory-bound programs without affecting their performance [3], since such programs spend their time waiting for memory. In this case we can expect an improvement in power-efficiency metrics such as EDP or ED^2P due to reducing the frequency. However this is only feasible for a few predominantly memory-bound programs. In most programs, although there are considerable opportunities to scale frequency while waiting for memory, we are unable to benefit 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 is prohibitive for modern CPUs to 2000ns for the hardware (without including kernel/driver overhead), 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 DVFS at a finer granularity than the whole program. If program execution can be divided into distinct phases with homogeneous memory behavior, then the optimal frequency for each phase can be calculated separately [4]. The finer the division, the more effective DVFS can be. But there is a limit due to the DVFS transition latency. 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 [5] in combination with guided DVFS. We execute programs as a series of asynchronous *tasks*, where each task is a C/C++ function. Each task is split into 2 fine grain phases: *access* (data prefetching) and *execute* (original computation). The key idea is that decoupling data access from computation allows us to make different voltage-frequency decisions for each phase (access vs. execute). Prefetching data in the access phase transforms most of cache misses into hits for the execute phase, thereby improving its performance. 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. In the execute phase most of the cache misses have been eliminated by the prefetching behavior of the access phase. This reduces stalls and makes highest frequency the best fit in terms of EDP.

To obtain the best power efficiency, modern CPUs require applications to be parallelized and scaled to many cores. For that 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, parallel execution is divided into tasks that can be scheduled independently. Because of this, we can easily break 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, while the execute phase is the whole task. Although this leads to redundant execution of all address calculations and

memory access instructions, it requires minimal programmer or compiler effort. The redundant execution of all memory access code makes our results conservative, and presents the possibility of significantly better results if this redundancy can be eliminated.

We use a custom task based parallel runtime system where each task is a C/C++ function and its arguments. From that function we create the access phase by removing all result computation. In terms of correctness, the access phase cannot cause any side effects, given that it has no stores. This can be either generated automatically by the compiler or manually by the programmer. The unmodified task function (execute phase) is executed on the same core after the access phase. 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 modern systems using accurate, fine-grain, power measurements taken directly from the processor power rails and model 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 [6] to enable low-latency per core DVFS [7].

Our benchmarks show that using DAE we can achieve EDP results comparable to or even better than the optimal EDP of a *coupled* access-execute (CAE), without any performance degradation if we have per core DVFS and low latency DVFS transitions. This demonstrates that decoupling access from execute can improve the effectiveness of DVFS. Our evaluation further shows that we can achieve performance comparable to CAE at max frequency (and in many cases better) and at the same time reduce EDP.

II. Related Work

The idea of decoupling access from execution was initially proposed by Smith [5] 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 hierarchy latencies has similar effect. Kamruzzaman et.al. [8] presented a technique to parallelize the memory accesses for single-threaded applications using a decoupled approach with helper threads to speculatively prefetch data. Their work targeted execution time reduction compared to the sequential execution with a minimal compiler or programmer effert. In terms of energy and EDP efficiency, their work shows reduced scalability compared to parallel execution because they parallelize only the prefetch phase (up to 60% speedup using 4 cores), at the trade-off of linear power increase when enabling cores (up to 4x power consumption). Karlsson and Hagersten [9] discuss run-ahead execution for conserving memory bandwidth. Decoupled execution preserves bandwidth by improving prefetch accuracy and at the same time preserves power by enabling per phase DVFS.

Dark silicon [10] describes a problem on which technology advances allow us to increase the number of transistors on chip linearly per generation creating thermal-power constrains that restricts us from being able to turn all of them on at the same time. Turning on and off cores implies significant overhead compared to that of DVFS. This is a strong motivation to create program or task phases with different power demands to keep high performance and low overall TDP (Thermal Design Power) on multicores. Keramidas et.al. [3] presented tools and techniques for efficient DVFS on CAE. Spiliopoulos et.al. [4] further improve that work and embed it in the linux kernel, using MSHR based models to detect and DVFS application phases. Instead of trying to adjust DVFS granularity to program demands, in our approach we adjust program demands to DVFS granularity.

III. Methodology

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

A. 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 collects information on the IPC of the tasks and their execution times, which are then used by our power model to determine the optimal DVFS setting for each task and compute its power and execution time. This allows the programmer to adjust the application power behavior from the runtime by selecting different DVFS policies.

In our runtime a task is a C/C++ function that can be executed asynchronously and in parallel. The runtime stores the function pointer and the arguments of the function internally and schedules the task for parallel execution. 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. To support the execution of decoupled tasks we provide an interface that defines two functions for a task: one for access and one for execute.

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 tasks 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 synchronization and *point to point* for fine 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 for an empty task being only 400 core cycles on Intel SandyBridge processor.

B. Generating access tasks

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 x86 *PREFETCH* instruction [11] because it does not stall instruction retirement and does not require a destination register. The access tasks can prefetch both load and store data into the cache. However only loads can stall the pipeline and are therefore performance critical. Stores are less likely to create stalls as they are buffered.

In DAE we are forced 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. This includes not only the extra prefetches but also the address calculations for them. In terms of performance, the duplicate address calculation is overlapped with the long latencies of prefetch misses thereby reducing its impact. From an energy perspective increasing the number of instructions executed on the core results in a direct power increase. In practice this affects power significantly less than frequency but still remains a source of inefficiency.

C. Power Model

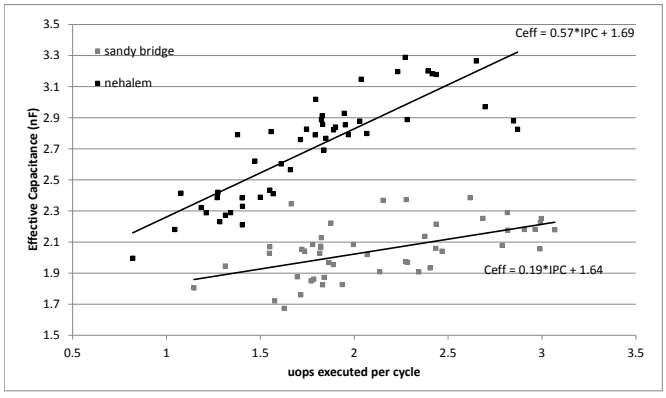

Fig. 1: Correlation between effective capacitance-uops executed per cycle for nehalem and sandy bridge. Methodology described in [4]

The fine granularity of our task-based approach (each task lasts from a few microseconds up to a millisecond) does not allow us to measure power consumption per task directly. Current power measurement infrastructures (both integrated on-chip power sensors and external measurement hardware) provide sampling periods at the order of milliseconds. To overcome this limitation, we employ an IPC-based model to estimate power consumption. Similar to [4], we assume that effective capacitance correlates linearly with the micro-ops executed by the processor per cycle. In order to derrive the parameters that best describe this dependence between effective capacitance and uops executed per cycle, we execute the whole SPEC2006 benchmark suite and measure the power consumed by the processor. By subtracting the static power (measured when the processor is idle and thus not consuming any dynamic power) we obtain the dynamic power for each benchmark.

Since dynamic power is given by $P = fCV^2$, dividing the derived dynamic power with the the product of $f*V^2$ results in the average effective capacitance of each benchmark. Figure 1 shows the effective capacitance-IPC pairs for the SPEC2006 suite. The figure also shows how effective capacitance can be approximated as a linear function of IPC. This equation describes the behavior of each of the 4 cores of our processor. Effective capacitance (and consequently dynamic power) can then be estimated for any task executed in a processor core. If we know its *IPC*, f and *V* the total processor power can then be derived by adding the corresponding power estimation for each of the cores and the static power, depending on the number of active cores. This approach enables us to estimate power consumption of any interval of a given IPC, however short in its duration with an average error less than 5%.

D. 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 energy and execution time. These two limitations are: 1) the inability to measure per-task energy consumption for fine-grain, simultaneously executing tasks, and, 2) the inability of current hardware to provide per-core, low-latency DVFS. Per-task energy measurement is impractical on modern hardware due to the very short execution time of our tasks and overlapping of different tasks across different cores. Our hardware measurement infrastructure provides only per-socket power measurements at a much coarser time granularity, and therefore tends to report average power consumption across many tasks. For DVFS, the current overhead for frequency and voltage switching is between 2 and 10 usec, which implies great overhead in applications that require fine grain tasks. 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 in future generations [6].

To overcome the above restrictions we have developed a model that allows us to accurately predict the energy and

performance we could obtain if we added per-core, low-latency DVFS to modern processors. This model uses our hardware-calibrated IPC-based power model to predict the power consumption of our fine-grained tasks based on their measured execution IPC across all available frequencies and voltages. Because this model is calibrated on our test hardware, we expect the power and performance estimates to accurately reflect the performance we would obtain if we could add per-core, low-latency DVFS to current processors. To verify these results, we ran all applications at all available frequencies and measured the power with our external measurement infrastructure [12]. This information was then used to compare the modelled total execution energy with the measured energy for each frequency, and demonstrates an error below 5%.

IV. Evaluation

The goal of the decoupled access-execute model is to obtain optimal power efficiency through DVFS with minimal performance degradation. We implement two DVFS policies: (1) *naive*, where the access phase runs always at lowest frequency and the execute phase always at highest and (2) *optimal EDP* in which the runtime 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 saving of access phase. In some memory-bound applications with very irregular memory access patterns, the total execution time is significantly lower over coupled execution due to prefetching accuracy of the access phase, which leads to an additional EDP improvement. For the optimal EDP policy the runtime tries to intelligently DVFS each phase based on power model thus total performance may be reduced because the execute phase is allowed to run slower in order to improve overall EDP. As a baseline for our experiments we use the original coupled execution at highest frequency. To demonstrate the full potential of our technique we additionally compare both DAE policies with an optimal EDP coupled execution, found by brute force exploration of available frequencies.

Increasing parallelism in any class of applications make the application more memory-bound as it increases the number of requests to the shared DRAM resulting in longer stalls per request due to memory bandwidth saturation. This limits program scalability. Our technique eliminates these stalls by prefetching data into the access phase at low frequency, thereby achieving improved power efficiency. For Intel's latest architectures we have found that bandwidth is not affected by the core frequency. This behavior makes our model a natural fit for these machines because we can efficiently reduce the power spent by the cores with DVFS in the access phase, without sacrificing bandwidth.

TABLE I: Application characteristics and task configuration

Application	% T_{access}	Task Size	Access Pattern
LU	3.3	16K - 48K	Tiled
Cholesky	5.6	16K - 48K	Tiled
FFT	10.5	16K - 128K	Butterfly
LBM	51.0	38K	Stream Collide
LibQ	56.9	8K - 16K	Regular
Cigar	66.2	32K	Indirection

A. Evaluation framework

For the evaluation framework we ported FFT [13], LU [14] and Cholesky from the SPLASH2 [15] micro-benchmarks to our runtime. We optimize the execution kernels using SSE. These applications represent rather computationally intensive applications. Additionally we parallelized and ported two applications from SPEC CPU2006 [16], 470.lbm and 462.libquantum. Both of these applications can be characterized as memory-bound. Finally we include CIGAR as described in [17]. This set of benchmarks covers a wide range of memory access patterns and bounds, from computation-bound represented by LU and Cholesky to memory-bound (CIGAR and libquantum). FFT and LBM have an intermediate behaviour.

B. Performance and EDP evaluation

The overall idea of DAE is to eliminate cache misses from the execute phase to allow it run efficiently at the highest frequency. Ideally the total execution time for the naive DAE policy would be very close to the original at max frequency. Fig. 2 shows performance (left) and EDP (right) results. The data is normalized to the coupled execution at the highest frequency (e.g., a value of 1.0 is equivalent to the non-decoupled program with the default linux frequency governor). The first bar shows the performance (or EDP for the right plot) of *CAE* at the frequency that has the optimal EDP. The second and third bars show the performance (or EDP respectively) for the *naive* and *optimal* EDP DAE policies, respectively. The results in Fig. 2 show an average performance improvement of 5% compared to the baseline and over 15% compared to CAE at optimal EDP. DAE keeps EDP slightly lower than that of the optimal EDP of a coupled execution, which itself is 25% better than the baseline. The average execution time for DAE optimal EDP policy is 5% worse than in the DAE naive policy but still 10% better than the CAE optimal EDP. DAE optimal EDP policy achieves an average of 30% EDP improvement over the baseline and 5% over the best coupled execution.

Fig. 2 also shows that all of our applications (with an exception of Cholesky) have an execution time less or equal to the baseline. More specifically DAE improves execution time by 12% for *libquantum* and 17% for *Cigar* and EDP by 46% and 54%, respectively. This is because both applications are memory-bound (as shown in Table I) and have a memory access pattern that the hardware prefetcher cannot

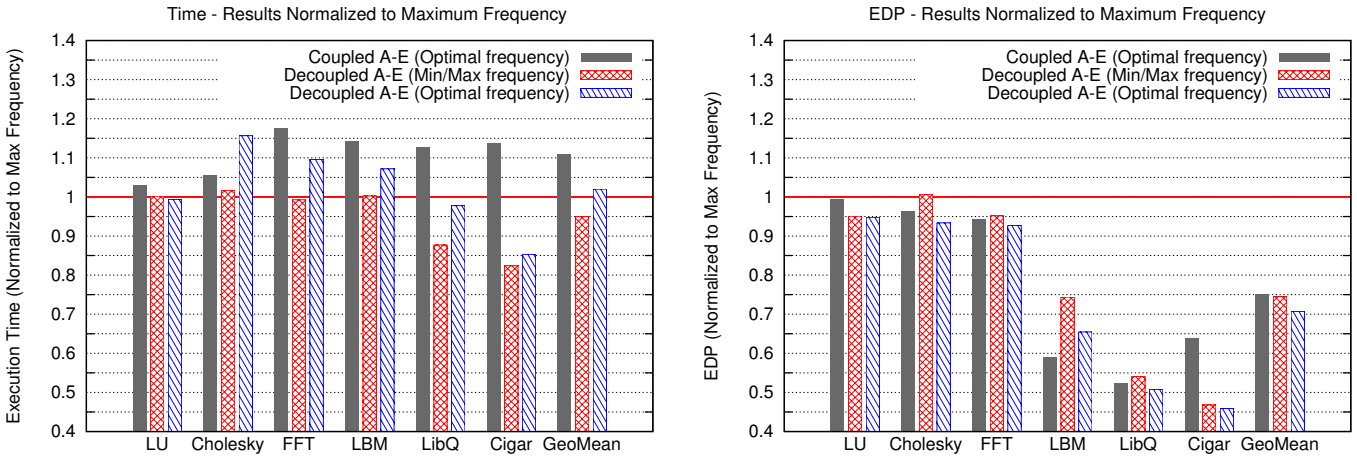

Fig. 2: Overall execution runtime and EDP using coupled and decoupled models using 4 cores, Intel Sandybridge

detect. The software prefetching of DAE can therefore improve performance (and also EDP). Libquantum by default has a regular memory access pattern that dynamic scheduling and load balancing at fine grain tasks can convert into rather irregular for the hardware prefetcher.

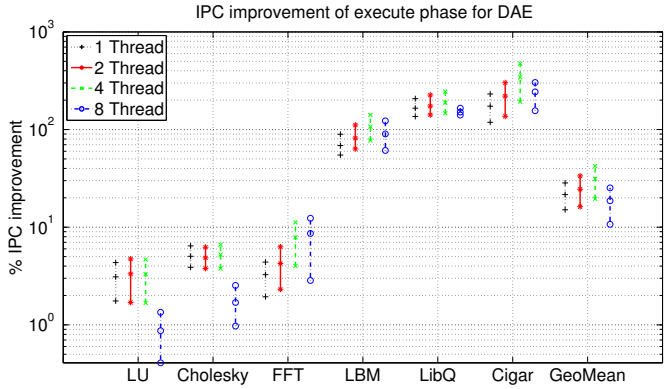

Fig. 3: IPC improvement of DAE - execute phase over coupled execution for Intel SandyBridge

C. Understanding performance

An increase in the IPC of the execute phase under DAE is the result of locality improvement caused by the access phase. We have verified a significant decrease at L1 and L2 miss rate of the execute phase as a result of data prefetching from the access phase. Fig. 3 shows the IPC increase of the execute phase due to the access phase. For computation-bound applications such as LU, Cholesky and FFT, we observe only a slight increase in IPC of 5% on average. For these applications there is no significant performance benefit from DAE and the time spent to the access phase is typically less than 10%.

For the memory-bound applications (LBM, libquantum and Cigar) we achieve a significant IPC increase due to DAE. The time spent in the access phase for these applications is significant and is shown in Table I. For these applications more than half of the time is spent in the access phase and the rest in execute and runtime overhead. For LBM the IPC improvement of the execute phase for 4 threads is more than 100% as shown in Fig. 3. Libquantum and CIGAR have an average IPC increase of 200% and 350%, respectively due to prefetching from the access phase. For this applications the average IPC of both (access and execute) phases is increased over the initial CAE IPC. This explains the total execution time reduction shown in Fig. 2, which in turn improves EDP for these applications. For correct operation of the model is not necessary to increase performance and in practice it is only feasible for memory access patterns that are very irregular and impossible for the hardware prefetcher.

V. Conclusions

In this work we explored the DVFS potential of decoupling access from execute 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 on a machine with low latency per core DVFS could achieve a 25% EDP reduction without sacrificing performance. Our optimal EDP policy can further reduce EDP by 5% over optimal CAE policy at the trade-off of 5% performance degradation.

References

[1] 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*, vol. 9, no. 5, pp. 256 – 268, oct 1974.

[2] ITRS. (2011) Design 2011 edition. [Online]. Available: http://www.itrs.net/Links/2011ITRS/2011Chapters/

[3] 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*, ser. CF '10. New York, NY, USA: ACM, 2010, pp. 287–296. [Online]. Available: http://doi.acm.org/10.1145/1787275.1787338

[4] 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*, ser. IGCC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/IGCC.2011.6008552

[5] J. E. Smith, "Decoupled access/execute computer architectures," *SIGARCH Comput. Archit. News*, vol. 10, no. 3, pp. 112–119, Apr. 1982. [Online]. Available: http://dl.acm.org/citation.cfm?id=1067649.801719

[6] 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*, vol. 47, no. 1, pp. 206 –219, jan. 2012.

[7] 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*, feb. 2008, pp. 123 –134.

[8] 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*, ser. ASPLOS '11. New York, NY, USA: ACM, 2011, pp. 393–404. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950411

[9] M. Karlsson and E. Hagersten, "Conserving memory bandwidth in chip multiprocessors with runahead execution," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, march 2007, pp. 1 –10.

[10] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *SIGARCH Comput. Archit. News*, vol. 39, no. 3, pp. 365–376, Jun. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024723.2000108

[11] Intel. (2012) Intel 64 and ia-32 architectures optimization reference manual, pp.366-369. [Online]. Available: http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf

[12] 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*, aug. 2012, pp. 241 –250.

[13] D. H. Bailey, "Ffts in external of hierarchical memory," in *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 1989, pp. 234–242.

[14] S. C. Woo, J. P. Singh, and J. L. Hennessy, "The performance advantages of integrating block data transfer in cache-coherent multiprocessors," in *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 1994, pp. 219–229.

[15] 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*. New York, NY, USA: ACM, 1995, pp. 24–36.

[16] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006. [Online]. Available: http://doi.acm.org/10.1145/1186736.1186737

[17] "Cigar - case injected genetic algortihm," http://ecsl.cse.unr.edu/. [Online]. Available: http://www.cse.unr.edu/ sushil/class/gas/code/cigar/