Predicting Performance Impact of DVFS for Realistic Memory Systems

Rustam Miftakhutdinov[†] Eiman Ebrahimi[‡] Yale N. Patt[†]
[†]The University of Texas at Austin [‡]Nvidia Corporation
{rustam,patt}@hps.utexas.edu ebrahimi@hps.utexas.edu

Abstract

Dynamic voltage and frequency scaling (DVFS) can make modern processors more power and energy efficient if we can accurately predict the effect of frequency scaling on processor performance. State-of-the-art DVFS performance predictors, however, fail to accurately predict performance when confronted with realistic memory systems. We propose CRIT+BW, the first DVFS performance predictor designed for realistic memory systems. In particular, CRIT+BW takes into account both variable memory access latency and performance effects of prefetching. When evaluated with a realistic memory system, DVFS realizes 65% of potential energy savings when using CRIT+BW, compared to less than 34% when using previously proposed DVFS performance predictors.

1. Introduction

Dynamic voltage and frequency scaling (DVFS) [1, 15] enables significant improvements in power and energy efficiency of modern processors. With DVFS support, a processor can alter its performance and power consumption on the fly by changing its frequency and supply voltage. This ability allows the processor to continuously adapt to dynamically changing application characteristics.

Exploiting the full potential of DVFS requires accurate performance and power prediction. If the processor can accurately predict what its performance and power consumption would be at any operating point, it can switch to the optimal operating point for any efficiency metric (e.g., energy or energy-delay-squared).

Existing DVFS performance predictors, however, fail to accurately predict performance under frequency scaling due to their unrealistic view of the off-chip memory system. Recently, two DVFS performance predictors have been proposed: *leading loads* [12, 20, 34][1] and *stall time* [12, 20]. Both assume a linear DVFS performance model, which, as we show in Section 3.2, does not model the performance effects of prefetching. In addition, leading loads was inspired by a simplified constant access latency view of memory and breaks down when confronted with a more realistic variable latency memory system. Figure 1 illustrates how the fraction of potential energy savings[2] actually realized by leading loads and stall time on memory-intensive workloads decreases as we increase the realism of the modeled memory system.

In this paper, we propose CRIT+BW, the first DVFS performance predictor for an out-of-order processor with a realistic DRAM system and a streaming prefetcher. We focus on the realism of the memory system because the effect of chip frequency scaling on performance depends largely on memory system behavior (as described in Section 2.2). Therefore, any DVFS performance predictor must be designed for and evaluated with a realistic memory system.

We develop CRIT+BW in two steps. First, we address variable memory access latency—a key characteristic of modern DRAM systems ignored by leading loads. To this end, we design CRIT, a DVFS performance predictor that accounts for variable memory access latency. The key idea is to predict the memory component of execution time by measuring the *critical path* through memory requests (hence the name "CRIT"). Second, we show that in the presence of prefetching, performance may be limited by achievable DRAM bandwidth—an effect ignored in the linear DVFS performance model used by leading loads and stall time. We develop a new *limited bandwidth* DVFS performance model that accounts for this effect and extend CRIT to use this performance model; CRIT+BW is the result ("BW" is shorthand for "bandwidth").

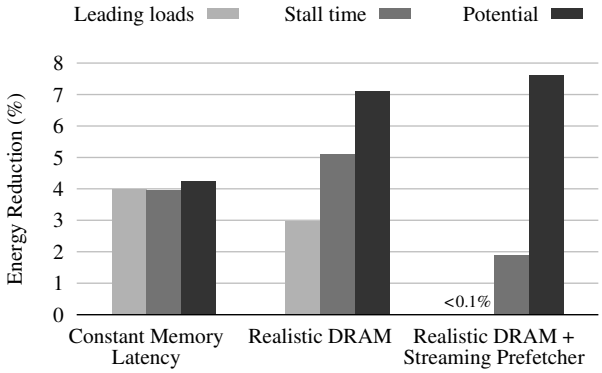

Figure 1: Energy savings realized by leading loads and stall time versus potential energy savings on 13 memory-intensive SPEC 2006 benchmarks

We evaluate CRIT+BW on an out-of-order processor capable of scaling the chip frequency from 1.5 GHz to 4.5 GHz, featuring a streaming prefetcher and a modern 800 MHz DDR3 SDRAM memory system. Across SPEC 2006, CRIT+BW realizes 65% of potential energy savings, compared to 34% for stall time and 12% for leading loads.

2. Background

2.1. Dynamic Voltage and Frequency Scaling

Dynamic voltage and frequency scaling (DVFS) [1, 15] helps increase power and energy efficiency of modern processors. DVFS does so by allowing the processor to switch between *operating points* (voltage/frequency combinations) at runtime. This capability gives rise to the problem of choosing the optimal operating point at runtime.

Traditionally, DVFS has been applied at the chip level only; recently, however, other DVFS domains have been proposed. David et al. [9] propose DVFS for off-chip memory and Intel's Westmere [23] supports multiple voltage/clock domains inside the chip. In this work, we focus on chip level DVFS.

2.2. DVFS Performance and Power Prediction

Estimating the performance impact of changing the chip's operating point is critical to choosing the optimal operating point. Which operating point is optimal depends on the chosen efficiency metric, e.g., energy or energy-delay-squared. All commonly used efficiency

[1] These three works propose very similar techniques. We use the name "leading loads" from Rountree et al. [34] for all three proposals.

[2] Section 4.3 describes the *dynamic optimal* DVFS policy used to calculate potential energy savings.

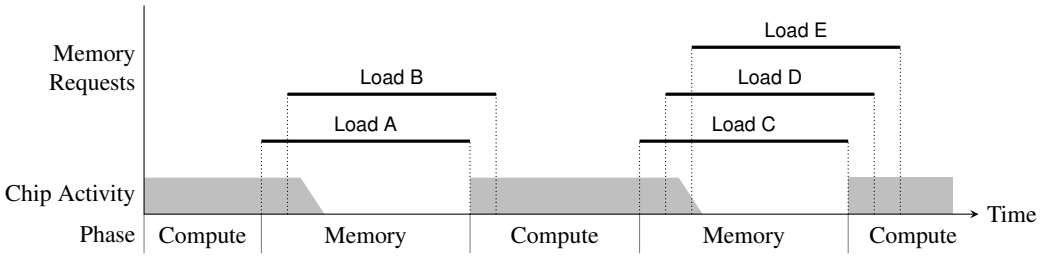

Figure 2: Two-phase abstract view of out-of-order execution used by leading loads

metrics are functions of execution time and power. Hence, choosing an optimal operating point requires a prediction of both performance and power at each of the available operating points. In this paper, we focus on performance prediction.

Predicting performance at different chip frequencies is made particularly difficult by the interaction between frequency scaling and the memory system's effect on performance. While memory latencies (as measured in seconds) are not affected by chip frequency scaling, they do scale with chip frequency in terms of processor cycles. The impact of these delay fluctuations on processor performance depends on the application, which further complicates DVFS performance prediction.

A DVFS performance predictor generally employs a *performance model* that predicts performance across a range of frequencies based on parameters measured at runtime. Specifically, the predictor measures these parameters during an execution interval and feeds them into the performance model to produce a performance estimate for every available frequency. These estimates, together with the corresponding power estimates, are then used to select the estimated optimal operating point for the next execution interval. Once the next execution interval ends, the process repeats.

Most published DVFS performance predictors [5–8, 10, 25, 29] rely on existing performance counters as inputs to their performance models. Many [5–8,25] use statistical regression analyses to correlate measured parameters with observed performance. An alternative approach is to design new hardware counters based on insight into the microarchitectural effects of frequency scaling, as done by the leading loads and stall time mechanisms described below.

2.2.1. Leading Loads. Leading loads [12, 20, 34] is a state-of-the-art DVFS performance predictor for out-of-order processors. The leading loads predictor was designed based on two simplifying assumptions about the memory system:

1. all memory requests have the same latency, and
2. after an instruction fetch or a data load misses in the last level cache and generates a memory request, the processor continues to execute but eventually runs out of ready instructions and stalls before the memory request returns.

Figure 2 shows the abstract view of execution implied by these assumptions. In this view, the out-of-order processor splits its time between two alternating phases: compute and memory. In the *compute* phase, the processor runs without generating any memory requests due to instruction fetches or data loads. As soon as the processor generates the first such memory request, the compute phase ends and the *memory* phase begins. At first, the out-of-order processor continues to execute instructions independent of the original memory request and may generate more memory requests. Eventually, however, the processor runs out of ready instructions and stalls. Since the processor generated the memory requests at roughly the same time, and the memory requests have the same latencies, they return data to the chip at about the same time as well. As soon as the first memory request returns, the memory phase ends and another compute phase begins.

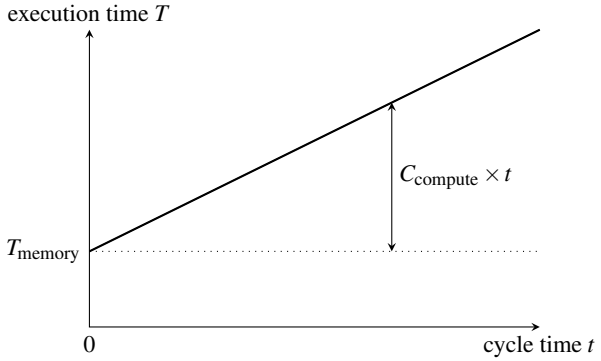

Figure 3: Linear DVFS performance model

This two-phase view of execution predicts a linear relationship between execution time T and chip cycle time t. To show this, we let

$$T = T_{\text{compute}} + T_{\text{memory}},$$

where T_{compute} denotes the total length of the compute phases and T_{memory} denotes the total length of the memory phases. As t changes due to DVFS, the number of cycles C_{compute} the chip spends in compute phases stays constant; hence

$$T_{\text{compute}}(t) = C_{\text{compute}} \times t.$$

Meanwhile, T_{memory} remains constant for every frequency. Thus, given measurements of C_{compute} and T_{memory} at any cycle time, we can predict execution time at any other cycle time:

$$T(t) = C_{\text{compute}} \times t + T_{\text{memory}}. \quad (1)$$

Figure 3 illustrates this linear model.

Leading loads introduces a hardware counter that continually accumulates T_{memory}. In each memory phase, the latency of the first memory request generated by a load is added to the counter; hence the name "leading loads." To estimate C_{compute}, leading loads employs existing performance counters to measure $T(t)$ and calculates

$$C_{\text{compute}} = \frac{T(t) - T_{\text{memory}}}{t}.$$

Note that, even though leading loads is derived from a simplified constant access latency view of memory, the mechanism can still be applied to more realistic memory systems.

2.2.2. Stall Time. Like leading loads, the *stall time* [12, 20] DVFS predictor uses the linear DVFS performance model. The key idea is

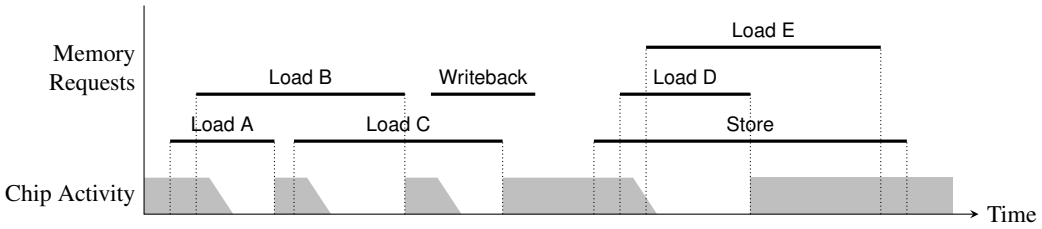

Figure 4: Abstract view of out-of-order processor execution with a variable latency memory system

simple: the time the processor spends unable to retire instructions due to an outstanding off-chip memory access should stay roughly constant as chip frequency is scaled (since this time depends largely on memory latency, which stays constant). The stall time predictor uses this time as the memory component of execution time (T_{memory} in Equation 1).

Unlike leading loads, the stall time predictor is not based on an abstract view of execution. The connection between retirement stalls due to memory accesses and T_{memory} is intuitive but not mathematically precise.

2.3. Realistic Memory System Architecture

Modern memory systems employ dynamic random access memory (DRAM) and streaming prefetching.

2.3.1. DRAM. In modern DRAM systems, contrary to leading loads' simplified constant access latency view of memory, memory request latency varies based on the addresses of the access stream [28]. Every memory address is statically mapped to one of several memory *banks* and one of many *rows* within its bank. Requests that map to different banks can be serviced in parallel, while those that map to the same bank have to be serviced serially. Among requests mapped to the same bank, requests that map to the same *row*, a 2–8 KB aligned block of memory, can be serviced faster than those that map to different rows. Memory request latencies also vary due to their wait time in the memory controller's queues.

2.3.2. Streaming Prefetcher. Streaming prefetchers are used in many commercial processors [2, 16, 24] and can greatly improve performance of memory intensive applications that stream through contiguous data arrays. Streaming prefetchers do so by detecting memory access streams and generating memory requests for data the processor will request further down stream. A well-performing streaming prefetcher significantly increases processor demand for memory bandwidth.

3. DVFS Performance Prediction on Realistic Memory Systems

We develop CRIT+BW, our DVFS performance predictor for realistic memory systems, in two steps.

First, we design CRIT, a DVFS performance predictor for a processor with a realistic DRAM system but no prefetcher. Like leading loads and stall time, CRIT measures the memory component of execution time within the confines of the linear DVFS performance model.

Second, we extend CRIT to account for performance effects of prefetching. We show that timely prefetching exposes the limiting effect of memory bandwidth on performance and develop a new DVFS performance model that accounts for this effect. The complete CRIT+BW predictor consists of the limited bandwidth DVFS performance model and hardware mechanisms that measure its parameters.

3.1. Realistic DRAM System with No Prefetching

Introducing a realistic DRAM system breaks the leading loads' abstract view of execution based on constant latency memory. Specifically, memory requests can now have very different latencies depending on whether they contend for DRAM banks and whether they map to the same row. Hence, the abstract view of processor execution relied on by leading loads becomes incorrect and, as we demonstrated in Figure 1, the predictor becomes ineffective.

Still, in the absence of prefetching, the other premise of leading loads (and stall time) still applies: after sending out a few instruction fetch or data load memory requests the processor eventually stalls. Figure 4 illustrates the abstract view of processor execution when memory latency is allowed to vary. Note that the processor eventually stalls under fetch and load memory requests.

This observation implies that execution time can still be modeled as the sum of a memory component whose latency remains constant under DVFS, and a compute component whose latency under DVFS changes in proportion to cycle time. Hence, the linear DVFS performance model (Equation 1) still applies in the case of a variable access latency memory system.

The introduction of variable memory access latencies, however, complicates the task of measuring the memory component T_{memory}. We must now calculate how execution time is affected by multiple memory requests with very different behaviors. Some of these requests are serialized (the first returns its data to the chip before the second one enters the memory controller). This serialization may be due to:

1. program dependencies (e.g., pointer chasing), or
2. limited core resources (e.g., if the out-of-order instruction window is too small to simultaneously contain both instructions corresponding to the two memory requests).

Other requests, however, overlap freely.

To estimate T_{memory} in this case, we recognize that in the linear DVFS performance model, T_{memory} is the limit of execution time as chip frequency approaches infinity (or, equivalently, as chip cycle time approaches zero). In that scenario, the execution time equals the length of the longest chain of dependent memory requests that stall the processor (i.e., data loads and instruction fetches). We refer to this chain as the *critical path* through the memory requests.

To calculate the critical path, we must know which memory requests are dependent (and remain serialized at all frequencies) and which are not. We observe that independent memory requests almost never serialize; the memory controller schedules independent requests as early as possible to overlap their latencies. Hence, we make the following assumption:

> If two memory requests are serialized (the first one completes before the second one starts), the second one depends on the first one.

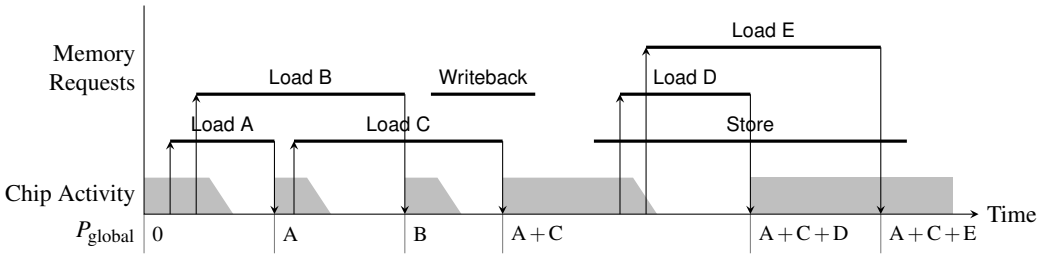

Figure 5: Critical path calculation example

3.1.1. Hardware Mechanism. We now describe CRIT, the hardware mechanism that uses the above assumption to estimate the critical path through load and fetch memory requests. CRIT maintains one global critical path counter P_{global} and, for each outstanding DRAM request i, a critical path timestamp P_i. Initially, the counter and timestamps are set to zero. When a request i enters the memory controller, the mechanism copies P_{global} into P_i. After some time ΔT the request completes its data transfer over the DRAM bus. At that time, if the request was generated by an instruction fetch or a data load, CRIT sets $P_{\text{global}} = \max(P_{\text{global}}, P_i + \Delta T)$. As such, after each fetch or load request i, CRIT updates P_{global} if request i is at the end of the new longest path through the memory requests.

Figure 5 illustrates how the mechanism works. We explain the example step by step:

1. At the beginning of the example, P_{global} is zero and the chip is in a compute phase.
2. Eventually, the chip incurs two load misses in the last level cache and generates two memory requests, labeled Load A and Load B. These misses make copies of P_{global}, which is still zero at that time.
3. Load A completes and returns data to the chip. Our mechanism adds the request's latency, denoted as A, to the request's copy of P_{global}. The sum represents the length of the critical path through Load A. Since the sum is greater than P_{global}, which is still zero at that time, the mechanism sets P_{global} to A.
4. Load A's data triggers more instructions in the chip, which generate the Load C request. Load C makes a copy of P_{global}, which now has the value A (the latency of Load A). Initializing the critical path timestamp of Load C with the value A captures the dependence between Load A and Load C: the latency of Load C will eventually be added to that of Load A.
5. Load B completes and ends up with B as its version of the critical path length. Since B is greater than A, B replaces A as the length of the global critical path.
6. Load C completes and computes its version of the critical path length as A + C. Again, since A + C > B, CRIT sets P_{global} to A + C. Note that A + C is indeed the length of the critical path through Load A, Load B, and Load C.
7. We ignore the writeback and the store because they do not cause a processor stall.
8. Finally, the chip generates requests Load D and Load E, which add their latencies to A + C and eventually result in P_{global} = A + C + E.

We can easily verify the example by tracing the longest path between dependent loads, which indeed turns out to be the path through Load A, Load C, and Load D. Note that, in this example, leading loads would incorrectly estimate T_{memory} as A + C + D.

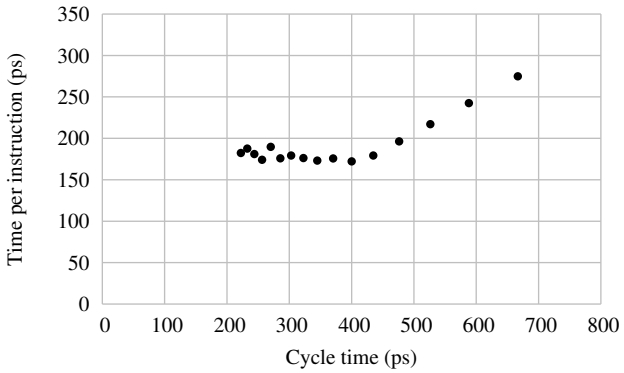

Figure 6: Time per instruction versus cycle time for `bwaves` with a streaming prefetcher enabled

3.2. Realistic DRAM System with Prefetching

Adding a prefetcher to the system changes the effect of DVFS on performance. Figure 6 shows time per instruction (TPI) for 100K retired instructions from `bwaves` at sixteen different cycle times (666–222 ps or 1.5–4.5 GHz) with a streaming prefetcher enabled. Note that these data points do not admit a linear approximation. This example is one of many where the linear performance model used by leading loads, stall time, and CRIT fails in the presence of prefetching.

The linear performance model fails due to the special nature of prefetching. Unlike demand memory requests, a prefetch request is issued in advance of the instruction that consumes the request's data. A prefetch request is *timely* if it fills the cache before the consumer instruction accesses the cache. Timely prefetches do not cause processor stalls; hence, their latencies do not affect execution time. Without stalls, however, the processor may generate prefetches at a high rate, exposing another performance limiter: the rate at which the memory system can satisfy memory requests (i.e., the memory bandwidth).

3.2.1. Limited Bandwidth Performance Model. We now describe a performance model, illustrated in Figure 7, that takes into account the performance limiting effect of finite memory bandwidth exposed by prefetching. This model splits the chip frequency range into two parts:

1. the low frequency range where the DRAM system can service memory requests at a higher rate than the chip generates them, and
2. the high frequency range where the DRAM system cannot service memory requests at the rate they are generated.

In the low frequency range, shown to the right of $t_{\text{crossover}}$ in Figure 7, the prefetcher runs ahead of the demand stream because the DRAM system can satisfy prefetch requests at the rate the prefetcher

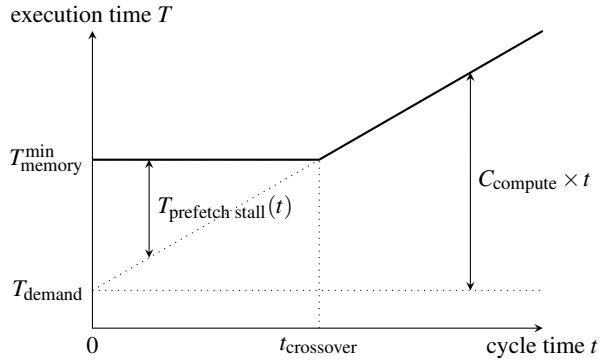

Figure 7: Limited bandwidth DVFS performance model

generates them. Hence, most prefetches are timely and instructions that use prefetched data result in cache hits. Execution time in this case is modeled by the original linear model, with only the non-prefetchable demand memory requests contributing to the memory component of the execution time, which we refer to as T_{demand}.

In the high frequency range, shown to the left of $t_{\text{crossover}}$ in Figure 7, the prefetcher fails to run ahead of the demand stream due to insufficient DRAM bandwidth. As the demand stream catches up to the prefetches, some demand requests stall the processor as they demand data that the prefetch requests have not yet brought into the cache. The delay due to these processor stalls is shown as $T_{\text{prefetch stall}}(t)$ in the figure.

Note that in the high frequency range the execution time is determined solely by $T^{\min}_{\text{memory}}$: the minimum time the DRAM system needs to satisfy all of the memory requests. Therefore, execution time does not depend on chip frequency in this case.

The limited bandwidth DVFS performance model shown in Figure 7 has three parameters:

1. the critical path through non-prefetchable demand memory requests T_{demand},
2. the number of cycles C_{compute} that the chip spends in the compute phase, and
3. the minimum time $T^{\min}_{\text{memory}}$ required by the DRAM system to satisfy the observed sequence of memory requests (both demands and prefetches).

Given the values of these parameters, we can estimate the execution time at any other cycle time t as follows:

$$T(t) = \max\left(T^{\min}_{\text{memory}},\ C_{\text{compute}} \times t + T_{\text{demand}}\right). \qquad (2)$$

3.2.2. Measuring Model Parameters. We now describe the hardware mechanisms to measure the parameters of the limited bandwidth DVFS performance model: T_{demand}, C_{compute}, and $T^{\min}_{\text{memory}}$. These mechanisms, together with the limited bandwidth DVFS performance model, comprise CRIT+BW—our complete DVFS performance predictor.

We measure T_{demand} in almost the same way as we measure T_{memory} in CRIT (Section 3.1): by calculating the critical path through memory requests. The only difference is that we exclude all prefetch requests and prefetchable demand requests from this calculation (just like we exclude stores and writebacks in CRIT). As shown in Figure 7, the extra chip stall time due to these prefetching-related requests, $T_{\text{prefetch stall}}(t)$, disappears at low frequencies. Therefore, this time is not a part of T_{demand}, which stays constant across frequencies.

Storage Component	Quantity	Width	Bits
Global critical path counter P_{global}	1	32	32
Copy of P_{global} per memory request	32	32	1024
Global DRAM slack counter	1	32	32
DRAM bus slack counter	1	32	32
Per DRAM bank slack counters	8	16	128
Prefetch stall counter	1	32	32
Total bits			1280

Table 1: Hardware storage cost of CRIT+BW

To calculate C_{compute} we recognize that

$$T(t) = T_{\text{demand}} + C_{\text{compute}} \times t + T_{\text{prefetch stall}}(t).$$

We can solve this equation for C_{compute} if we can measure $T_{\text{prefetch stall}}(t)$. To this end, we introduce a new hardware counter that tracks the time the processor is stalled while only prefetch requests and prefetchable demand requests are outstanding. With $T_{\text{prefetch stall}}(t)$ now known, we have

$$C_{\text{compute}} = \frac{T(t) - T_{\text{demand}} - T_{\text{prefetch stall}}(t)}{t}.$$

Recall that $T^{\min}_{\text{memory}}$ is defined as the minimum time the DRAM system needs to satisfy all of the memory requests. We can calculate $T^{\min}_{\text{memory}}$ if we can measure the amount of slack $T_{\text{memory slack}}$ in the memory system, because

$$T^{\min}_{\text{memory}} = T(t) - T_{\text{memory slack}}(t). \qquad (3)$$

The description of the slack measurement hardware follows.

Whenever the memory controller schedules a DRAM command (e.g., "precharge" or "column access"), it must ensure that the command does not violate DRAM timing constraints. Hence, the memory controller can compute the *slack* of the DRAM command: how much earlier could the DRAM command have been scheduled without violating the DRAM timing constraints. The memory controller accumulates this slack separately for each DRAM bank and for the DRAM bus.

The presence of DRAM slack, however, does not always imply that the DRAM command could have been scheduled earlier. In fact, the slack may be due to the inability of the memory controller to schedule distant memory requests in parallel owing to the finite size of its scheduling window.

We account for this limitation when measuring slack in order to not overpredict the amount of reducible slack. To do this, we reset slack measurement every *slack measurement period*, which ends whenever the number of memory requests serviced within it reaches the size of the scheduling window. At the end of each slack measurement period, the memory controller finds the least slack among the banks and the bus. The memory controller adds the least slack amount to the global DRAM slack counter $T_{\text{memory slack}}$ and resets the bus and bank slack counters, starting a new period. From any cycle time t we can now calculate $T^{\min}_{\text{memory}}$ using Equation 3.

3.3. Hardware Cost

Table 1 details the storage required by CRIT+BW. The additional storage is only 1280 bits. The mechanism does not add any structures or logic to the critical path of execution.

Frequency		Front end		OOO Core		All Caches			ICache	DCache	L2
Min	1.5 GHz	Uops/cycle	4	Uops/cycle	4	Line size	64 B	Size	32 KB	32 KB	1 MB
Max	4.5 GHz	Branches/cycle	2	Pipe depth	14	MSHRs	32	Assoc.	4	4	8
Step	100 MHz	BTB entries	4K	ROB size	128	Repl.	LRU	Cycles	3	3	18
		Predictor	hybrid[a]	RS size	48			Ports	1R/1W	2R/1W	1

DRAM Controller		Bus		DDR3 SDRAM [28]				Stream prefetcher [40]			
Policy	FR-FCFS [33]	Freq.	800 MHz	Chips	8×256 MB	Row size	8 KB	Streams	64	Distance	64
Window	32 requests	Width	8 B	Banks	8	CAS[b]	13.75 ns	Queue	128	Degree	4

[a] 64K-entry gshare + 64K-entry PAs + 64K-entry selector.
[b] CAS $= t_{RP} = t_{RCD} =$ CL; other modeled DDR3 constraints: CWL, $t_{\{RC, RAS, RTP, BL, CCD, RRD, FAW, WTR, WR\}}$.

Table 2: Simulated processor configuration

4. Methodology

We compare energy saved by CRIT+BW to that of the state-of-the-art (leading loads and stall time) and to three kinds of potential energy savings (computed using offline DVFS policies). Before presenting the results, we justify our choice of energy as the efficiency metric, describe our simulation methodology, explain how we compute potential energy savings, and discuss our choice of benchmarks.

4.1. Efficiency Metric

Our choice of efficiency metric is driven solely by the need to evaluate DVFS performance predictors. As such, the efficiency metric must be implementable by a simple DVFS controller (so that most of the benefit comes from DVFS performance prediction) and must allow comparisons to optimal results. Note that we are not evaluating the usefulness of DVFS itself.

We choose *energy* (or, equivalently,[3] *performance per watt*) by eliminating the other metrics from the set of the four commonly used ones: energy, energy delay product (EDP), energy delay-squared product (ED^{2}P), and execution time.

We eliminate EDP and ED^{2}P because they complicate DVFS performance predictor evaluation by 1) requiring another predictor in the DVFS controller, and 2) precluding comparisons to optimal results. Specifically, these metrics have the undesirable property that the optimal frequency for an execution interval depends on the behavior of the rest of execution. Therefore, the DVFS controller must keep track of past long-term application behavior and predict future long-term application behavior *in addition* to short-term DVFS performance prediction we are evaluating. The necessity of this additional prediction makes it hard to isolate the benefits of DVFS performance prediction in the results. This undesirable property also makes simulating an oracle DVFS controller infeasible, precluding comparisons to optimal results. Sazeides et al. [35] discuss these issues in greater detail.

We eliminate execution time as not applicable to chip-level DVFS. In this scenario, optimizing execution time does not require a performance prediction: the optimal frequency is simply the highest frequency.

Therefore, of the four common efficiency metrics, only energy is suitable for our evaluation.

4.2. Simulation Methodology

4.2.1. Timing Model. We use a cycle-accurate simulator of an x86 superscalar out-of-order processor. The simulator models port contention, queuing effects, and bank conflicts throughout the cache hierarchy and includes a detailed DDR3 SDRAM model. Table 2 lists the baseline processor configuration.

Component	Parameter	Value	
		@1.5 GHz	@4.5 GHz
Chip	Static power (W)	9	28
	Peak dynamic power (W)	2	58
DRAM	Static power (W)	1	
	Precharge energy (pJ)	79	
	Activate energy (pJ)	46	
	Read energy (pJ)	1063	
	Write energy (pJ)	1071	
Other	Static power (W)	40	

Table 3: Power parameters

4.2.2. Power Model. We model three major system power components: chip power, DRAM power, and other power (fan, disk, etc.).

We model chip power using McPAT 0.8 [26] extended to support DVFS. Specifically, to generate power results for a specific chip frequency f, we:

1. run McPAT with a reference voltage V_0 and frequency f_0,
2. scale voltage using $V = \max(V_{\min}, \frac{f}{f_0} V_0)$,
3. scale reported dynamic power using $P = \frac{1}{2} C V^2 f$, and
4. scale reported static power linearly with voltage [3].

We model DRAM power using CACTI 6.5 [30] and use a constant static power as a proxy for the rest of system power.

Table 3 details the power parameters of the system.

4.2.3. DVFS Controller. Every 100K retired instructions, the DVFS controller chooses a chip frequency for the next 100K instructions.[4] Specifically, the controller chooses the frequency estimated to cause the least system energy consumption. To estimate energy consumption at a candidate frequency f while running at f_0, the controller:

1. obtains measurements of
   - execution time $T(f_0)$,
   - chip static power $P_{\text{chip static}}(f_0)$,
   - chip dynamic power $P_{\text{chip dynamic}}(f_0)$,
   - DRAM static power $P_{\text{DRAM static}}(f_0)$,
   - DRAM dynamic power $P_{\text{DRAM dynamic}}(f_0)$, and
   - other system power $P_{\text{other}}(f_0)$

[3] Energy and performance per watt are equivalent in the sense that in any execution interval, the same operating point is optimal for both metrics.

[4] We chose 100K instructions because it is the smallest quantum for which the time to change chip voltage (as low as tens of nanoseconds [21, 22], translating to less than 1K instructions) can be neglected.

for the previous 100K instructions from hardware performance counters and power sensors,

2. obtains a prediction of execution time $T(f)$ for the next 100K instructions from the performance predictor (either leading loads, stall time, or CRIT+BW),
3. calculates chip dynamic energy $E_{\text{chip dynamic}}(f_0)$ and DRAM dynamic energy $E_{\text{DRAM dynamic}}(f_0)$ for the previous interval using $E = PT$,
4. calculates $E_{\text{chip dynamic}}(f)$ by scaling $E_{\text{chip dynamic}}(f_0)$ using $E = \frac{1}{2}CV^2$,
5. calculates $P_{\text{chip static}}(f) = \frac{V}{V_0} P_{\text{chip static}}(f_0)$ as in [3],
6. and finally calculates total estimated system energy

$$\begin{aligned} E(f) &= E_{\text{chip}}(f) + E_{\text{DRAM}}(f) + E_{\text{other}}(f) \\ &= P_{\text{chip static}}(f) \times T(f) + E_{\text{chip dynamic}}(f) + \\ &\quad P_{\text{DRAM static}}(f_0) \times T(f) + E_{\text{DRAM dynamic}}(f) + \\ &\quad P_{\text{other}}(f_0) \times T(f). \end{aligned}$$

To isolate the effect of DVFS performance predictor accuracy on energy savings, we do not simulate delays associated with switching between frequencies. Accounting for these delays requires an additional prediction of whether the benefits of switching outweigh the cost. If the accuracy of that prediction is low, it could hide the benefits of high performance prediction accuracy, and vice versa.

4.3. Offline Policies

We model three offline DVFS controller policies: *dynamic optimal*, *static optimal*, and *perfect memoryless*.

The *dynamic optimal* policy places a lower bound on energy consumption. We compute this bound as follows:

1. run the benchmark under study at each chip frequency,
2. for each interval, find the minimum consumed energy across all frequencies,
3. total the per-interval minimum energies.

The *static optimal* policy chooses the chip frequency that minimizes energy consumed by the benchmark under study, subject to the constraint that frequency must remain the same throughout the run. The difference between dynamic and static optimal results yields potential energy savings due to benchmark phase behavior.

The *perfect memoryless* policy simulates a perfect *memoryless* performance predictor. We call a predictor *memoryless* if it assumes that for each chip frequency, performance during the next interval equals performance during the last interval. This assumption makes sense for predictors that do not "remember" any state (other than the measurements from the last interval); hence the name "memoryless." Note that all predictors discussed in this paper are memoryless. For each execution interval, the perfect memoryless policy chooses the chip frequency that would minimize energy consumption during the *previous* interval.

The perfect memoryless policy provides a quasi-optimal[5] bound on energy saved by memoryless predictors. A large difference between dynamic optimal and perfect memoryless results indicates that a memoryless predictor cannot handle the frequency of phase changes in the benchmark under study. Getting the most energy savings out of such benchmarks may require "memoryful" predictors that can detect and predict application phases.[6] We leave such predictors to future work.

4.4. Benchmarks

We simulate SPEC 2006 benchmarks compiled using the GNU Compiler Collection version 4.3.6 with the `-O3` option. We run each benchmark with the reference input set for 200M retired instructions selected using Pinpoints [32].

4.4.1. Benchmark Classification. To simplify the analysis of the results, we classify the benchmarks based on their memory intensity and the number of prefetch requests they trigger. We define a benchmark as *memory-intensive* if it generates more than 3 last level cache misses per thousand instructions (with no prefetching). We define a benchmark as *prefetch-heavy* if it triggers more than 5 prefetch requests per thousand instructions. The resulting benchmark classes are the same across all simulated frequencies.

5. Results

We show results for two configurations: with prefetching turned off and with a streaming prefetcher. In both cases, we show normalized energy reduction relative to the energy consumed at 3.7 GHz, the most energy-efficient static frequency across SPEC 2006 (which happens to be the same for both cases).

Before analyzing the results, we first explain their presentation using Figure 8 as an example. Note that, for each benchmark, the figure shows five bars within a wide box. The height of the box represents dynamic optimal energy reduction. Since no other DVFS policy can save more energy than dynamic optimal, we can use this box to bound the other five bars. The five bars inside the box represent energy reduction due to 1) leading loads, 2) stall time, 3) CRIT+BW, 4) optimal static DVFS policy, and 5) perfect memoryless DVFS policy. This plot design allows for easy comparisons of realized and potential gains for each benchmark and simplifies comparison of potential gains across benchmarks at the same time.

5.1. Realistic DRAM with No Prefetching

Figure 8 shows realized and potential energy savings across thirteen memory-intensive workloads. On average, CRIT+BW and stall time realize 5.5% and 5.1% out of potential 7.1% energy savings, whereas leading loads only realizes 3%. For completeness, Figure 9 shows energy savings for low memory intensity benchmarks (note the difference in scale).

The subpar energy savings by leading loads are due to its constant memory access latency approximation. As described in Section 2.2.1, leading loads accumulates the latency of the first load in each cluster of simultaneous memory requests to compute the memory component T_{memory} of total execution time T. It turns out that in such clusters, the leading load latency is usually *less* than that of the other requests. In fact, this is the case in all memory-intensive benchmarks except `libquantum` and `lbm`; in these eleven benchmarks the average leading load latency is only 74% of the average latency of the other memory requests. This discrepancy is due to the fact that the first memory request in a cluster is unlikely to contend with another request for a DRAM bank, whereas the later requests in the cluster

[5] We call this bound *quasi-optimal* because an imperfect memoryless predictor may actually save more energy than the perfect memoryless predictor if the optimal frequency for the previous interval does not remain optimal in the next interval.

[6] Section 6.3 describes related work on phase prediction.

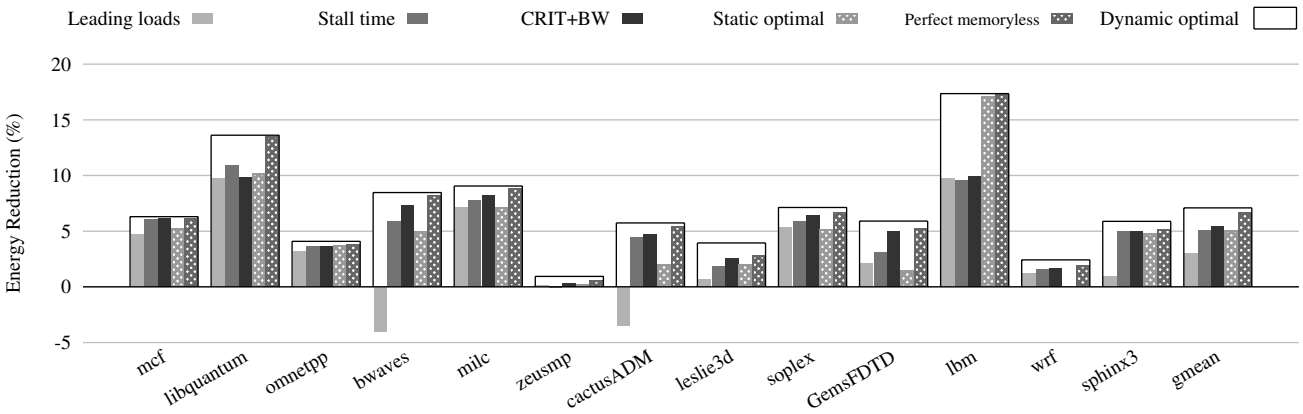

Figure 8: Realized and optimal energy savings for memory-intensive benchmarks (no prefetching)

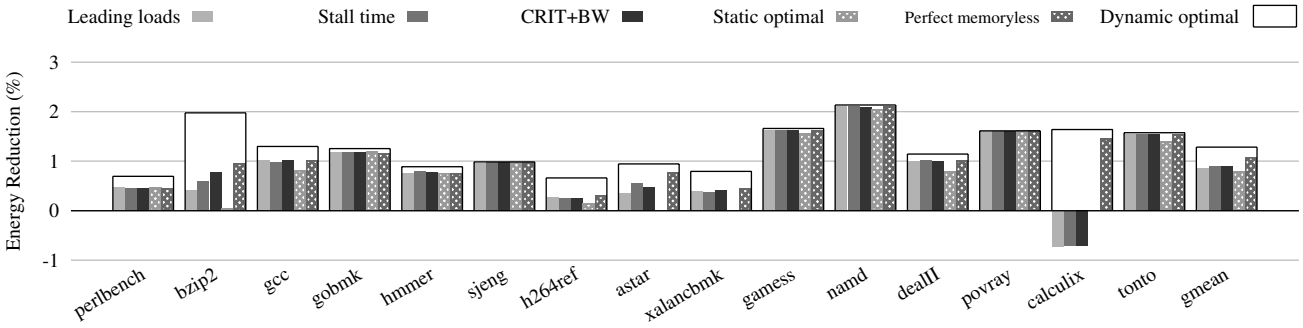

Figure 9: Realized and optimal energy savings for non-memory-intensive benchmarks (no prefetching)

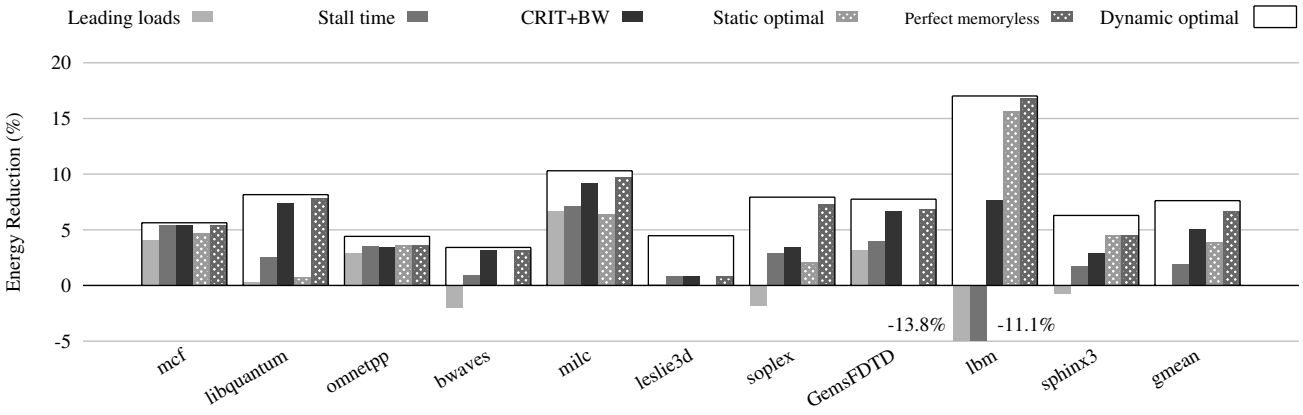

Figure 10: Realized and optimal energy savings for prefetch-heavy benchmarks

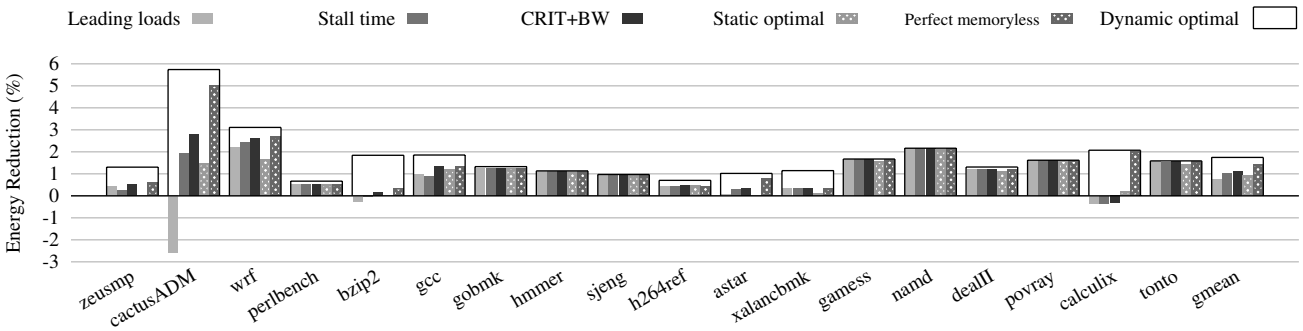

Figure 11: Realized and optimal energy savings for prefetch-light benchmarks

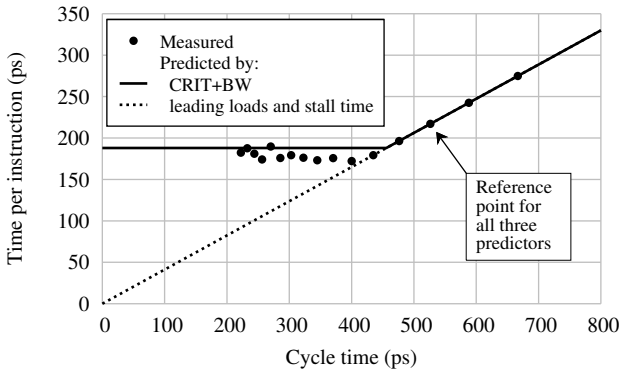

Figure 12: Measured and predicted TPI on `bwaves` with streaming prefetcher enabled

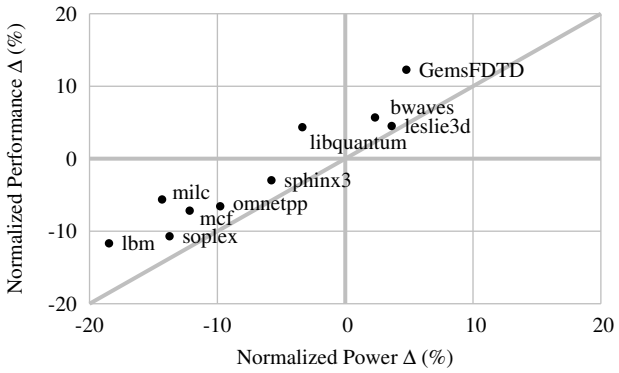

Figure 13: Performance delta versus power delta under DVFS with CRIT+BW for prefetch-heavy benchmarks

likely have to wait for the earlier ones to free up the DRAM banks. This underestimate of T_{memory} results in subpar energy savings, exemplified by `bwaves` and `cactusADM` on which leading loads actually consumes more energy than the baseline.

The fact that stall time beats leading loads supports our original argument that DVFS performance predictors must be designed for and evaluated with a realistic memory system. Both our experiments and prior work [12, 20] show that when evaluated with a constant access latency memory, leading loads saves more energy than stall time. Evaluation of the two predictors with a realistic DRAM system, however, shows this conclusion to be incorrect.

Note that CRIT+BW, the mechanism we derived from an abstract view of execution in Section 3, outperforms stall time, a mechanism based on a less precise view of execution, by a relatively small margin (5.5% vs. 5.1% energy saved). It is unclear, however, whether the approximations that make stall time work will hold in all configurations.

5.2. Realistic DRAM with Stream Prefetching

Figure 10 shows realized and potential energy reduction across ten prefetch-heavy benchmarks with a streaming prefetcher enabled. On average, CRIT+BW realizes 5% out of potential 7.6% energy savings, whereas stall time and leading loads only realize 1.8% and less than 0.1%, respectively. For completeness, Figure 11 shows energy savings for prefetch-light benchmarks (note the difference in scale).

5.2.1. Prediction Example. To provide insight into why CRIT+BW bests the competition on prefetch-heavy workloads, we analyze performance predictions generated by all three predictors for an interval of `bwaves`, the prefetch-heavy benchmark we use to motivate the limited bandwidth DVFS performance model in Section 3.2. Figure 12 contrasts the performance predictions generated by CRIT+BW, leading loads, and stall time. In particular, the figure shows:

1. sixteen thick dots representing measured time per instruction (TPI) at sixteen frequencies,
2. a dashed line showing TPI predicted by both leading loads and stall time while running at 1.9 GHz, and
3. a solid curve showing TPI predicted by CRIT+BW while running at 1.9 GHZ.

Note that all three predictions for low frequencies (right half of the figure) are identical. The reason lies in the highly streaming nature of `bwaves` that enables the prefetcher to eliminate *all* demand misses in the interval. Therefore, all three predictors estimate the memory component of execution time to be zero, predicting performance to scale proportionately with frequency.

On the other hand, predictions for higher frequencies diverge; CRIT+BW predicts that TPI saturates at 188 ps per instruction, whereas leading loads and stall time still predict performance to scale proportionally with frequency. Comparing the predictions to measured TPI demonstrates that accounting for limited memory bandwidth allows CRIT+BW to be more accurate than both leading loads and stall time.

Due to this prediction inaccuracy, a DVFS controller using either leading loads or stall time has to act on skewed estimates of energy consumption at high frequencies. Specifically, the controller may choose a high frequency and waste a lot of power for no performance benefit, losing out on potential energy savings.

5.2.2. Power and Performance Tradeoff. Figure 13 details how CRIT+BW trades off power and performance to reduce energy. The figure plots performance delta versus power delta (normalized to performance and power achieved at the baseline 3.7 GHz frequency). The diagonal line consists of points where performance and power deltas are equal, resulting in the same energy as the baseline.

CRIT+BW trades off power and performance differently across workloads. On `GemsFDTD`, `bwaves`, and `leslie`, CRIT+BW spends extra power for even more performance, while on `lbm`, `mcf`, `milc`, `omnetpp`, `soplex`, and `sphinx3` CRIT+BW allows performance to dip to save more power.

Note that CRIT+BW improves performance *and* saves power on `libquantum`. CRIT+BW does so by exploiting `libquantum`'s phase behavior. In some phases, CRIT+BW spends extra power for more performance; in others, it makes the opposite choice. On average, both performance and power consumption improve.

5.3. Analysis of `lbm`

With and without prefetching, `lbm` stands out due to its large potential energy savings which are not fully realized by CRIT+BW and the other predictors. The reasons, however, are different for each case.

Without prefetching, the reason lies in the peculiar nature of the benchmark. The majority (84%) of memory requests in `lbm` are stores and writebacks, which do not stall the processor. At high frequencies, however, the load memory requests are more likely to contend with these stores and writebacks for DRAM banks, taking more time to complete and thus violating the linear DVFS performance model assumption that T_{memory} stays the same across frequencies. This leads CRIT+BW (and the other predictors) to underestimate the performance effect of memory at high frequencies.

With prefetching, the reason lies in the details of memory request scheduling. At high frequencies, `lbm` floods the memory system with prefetches; this large number of memory requests allows the memory controller to make better scheduling decisions and reduce the number of row conflicts by up to 61%. The slack approach to estimating $T^{\text{min}}_{\text{memory}}$ does not take this effect into account, resulting in an overestimate of $T^{\text{min}}_{\text{memory}}$.

5.4. Summary

When evaluated on an out-of-order processor featuring a streaming prefetcher and a realistic DRAM system, CRIT+BW realizes 65% of dynamically optimal energy savings (75% of perfect memory-less energy savings) across all SPEC 2006 workloads, compared to only 34% (40%) for stall time and 12% (14%) for leading loads.

6. Related Work

To our knowledge, this paper is the first to propose a DVFS performance predictor designed to work with a realistic DRAM system. Specifically, our predictor addresses two characteristics of realistic DRAM systems which make DVFS performance prediction difficult: varying memory request latencies and prefetching, neither of which are considered by the state-of-the-art [12, 20, 34].

We have already compared our predictor to leading loads and stall time. Here we briefly discuss three major areas of related work: performance and power prediction for DVFS, analytical performance models, and phase prediction.

6.1. Performance and Power Prediction for DVFS

Most prior papers on DVFS performance and power prediction [5–8, 10, 25, 29] address the problem above the microarchitectural level and do not explore hardware modification. Hence, these approaches can only use already existing hardware performance counters as inputs to their performance and power models. These counters were not designed to predict the performance impact of DVFS and thus do not work well for that purpose. Hence, these papers resort to statistical [5–8, 25] and machine learning [10, 29] techniques.

In contrast, we design new hardware counters with the explicit goal of aiding DVFS performance prediction. This approach was introduced by leading loads [12, 20, 34] and stall time [12, 20] proposals and extended to power prediction by Spiliopoulos et al. [38].

The tradeoff between these two general approaches is as follows: statistical and machine learning techniques are easier to apply to complex prediction scenarios (e.g., per-core DVFS); however, our approach of designing new hardware counters builds on an understanding of the underlying microarchitectural effects that ensures robust predictions even for applications never seen before.

6.2. Analytical Performance Models

Traditional analytical performance models [4, 11, 13, 14, 19, 27, 31, 37, 39] have a different purpose than the commonly used linear DVFS performance model and our limited bandwidth DVFS performance model. Specifically, traditional analytical models are used to gain insight into the performance bottlenecks of modeled architectures and drive design space exploration. These models are evaluated off-line and target only a first order performance estimate. A DVFS performance model, on the other hand, is an analytical performance model evaluated at runtime by the operating system or the hardware power management unit and has to be accurate to be useful.

6.3. Phase Prediction

Phase detection and prediction mechanisms [17, 18, 36, 42] can help improve DVFS performance prediction accuracy and hence the overall utility of DVFS. Specifically, a DVFS mechanism can benefit from phase prediction by triggering re-training of the DVFS performance predictor in the beginning of each phase, and switching to the predicted optimal operating point for the rest of the phase.

7. Conclusions

We have shown that a DVFS performance predictor must be designed with an accurate model of the memory system in mind.

We demonstrated quantitatively that previously proposed DVFS performance predictors, designed with an over-simplified view of the memory system (e.g., assuming a constant access latency or disregarding prefetching), generate inaccurate performance predictions and lose out on potential energy savings. In particular, we have shown that the commonly used linear DVFS performance model breaks down in the presence of prefetching because it does not account for finite memory bandwidth.

To address these problems, we have 1) developed the limited bandwidth DVFS performance model that takes memory bandwidth into account, and 2) proposed CRIT+BW, a low cost mechanism that accurately predicts the performance impact of frequency scaling in the presence of a realistic memory system, realizing 65% of the potential energy savings.

Acknowledgments

We thank Onur Mutlu, members of the HPS research group, our shepherd Lieven Eeckhout, and the anonymous reviewers for their comments and suggestions. We thank Rafael Ubal and other developers of Multi2Sim [41], from which we adapted the x86 functional model that drives our performance simulator. We gratefully acknowledge the support of the Cockrell Foundation and Intel Corporation.

References

[1] T. D. Burd and R. W. Brodersen, "Energy efficient CMOS microprocessor design," in *Proc. 28th Hawaii Int. Conf. Syst. Sci. (HICCS-28)*, vol. 1, Jan. 1995, pp. 288–297.

[2] M. Butler, L. Barnes, D. D. Sarma, and B. Gelinas, "Bulldozer: An approach to multithreaded compute performance," *IEEE Micro*, vol. 31, no. 2, pp. 6–15, Mar. 2011.

[3] J. A. Butts and G. S. Sohi, "A static power model for architects," in *Proc. 33rd ACM/IEEE Int. Symp. Microarchitecture (MICRO-33)*, Dec. 2000, pp. 191–201.

[4] X. E. Chen and T. M. Aamodt, "Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs," in *Proc. 41st ACM/IEEE Int. Symp. Microarchitecture (MICRO-41)*, Nov. 2008, pp. 59–70.

[5] K. Choi, R. Soma, and M. Pedram, "Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times," in *Proc. Conf. Design, Automation, and Test in Europe (DATE 2004)*, vol. 1, Feb. 2004, pp. 4–9.

[6] G. Contreras and M. Martonosi, "Power prediction for Intel XScale processors using performance monitoring unit events," in *Proc. 2005 Int. Symp. Low Power Electron. and Design (ISLPED'05)*, Aug. 2005, pp. 221–226.

[7] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos, "Online power-performance adaptation of multithreaded programs using hardware event-based prediction," in *Proc. 20th Int. Conf. Supercomputing (ICS'06)*, Cairns, Queensland, Australia, Jun. 2006, pp. 157–166.

[8] M. Curtis-Maury, A. Shah, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz, "Prediction models for multi-dimensional power-performance optimization on many cores," in *Proc. 17th Int. Conf. Parallel Arch. and Compilation Techniques (PACT'08)*, Oct. 2008, pp. 250–259.

[9] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu, "Memory power management via dynamic voltage/frequency scaling," in *Proc. 8th ACM Int. Conf. Autonomic Computing (ICAC 2011)*, Jun. 2011, pp. 31–40.

[10] G. Dhiman and T. S. Rosing, "Dynamic voltage frequency scaling for multi-tasking systems using online learning," in *Proc. 2007 Int. Symp. Low Power Electron. and Design (ISLPED'07)*, Aug. 2007, pp. 207–212.

[11] P. G. Emma and E. S. Davidson, "Characterization of branch and data dependencies in programs for evaluating pipeline performance," *IEEE Trans. Comput. (TOC)*, vol. C-36, no. 7, pp. 859–875, Jul. 1987.

[12] S. Eyerman and L. Eeckhout, "A counter architecture for online DVFS profitability estimation," *IEEE Trans. Comput. (TOC)*, vol. 59, pp. 1576–1583, Nov. 2010.

[13] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A mechanistic performance model for superscalar out-of-order processors," *ACM Trans. Comput. Syst. (TOCS)*, vol. 27, pp. 3:1–3:37, May 2009.

[14] A. Hartstein and T. R. Puzak, "The optimum pipeline depth considering both power and performance," *ACM Trans. Arch. and Code Optimiz. (TACO)*, vol. 1, pp. 369–388, Dec. 2004.

[15] M. Horowitz, T. Indermaur, and R. Gonzalez, "Low-power digital design," in *IEEE Symp. Low Power Electron. (ISLPE'94) Digest of Tech. Papers*, Oct. 1994, pp. 8–11.

[16] *Intel 64 and IA-32 Architectures Optimization Reference Manual Version 026*, Intel Corporation, April 2012.

[17] C. Isci, G. Contreras, and M. Martonosi, "Live, runtime phase monitoring and prediction on real systems with application to dynamic power management," in *Proc. 39th ACM/IEEE Int. Symp. Microarchitecture (MICRO-39)*, Dec. 2006, pp. 359–370.

[18] C. Isci and M. Martonosi, "Phase characterization for power: Evaluating control-flow-based and event-counter-based techniques," in *Proc. 12th IEEE Int. Symp. High Perf. Comput. Arch. (HPCA-12)*, Feb. 2006, pp. 121–132.

[19] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *Proc. 31th Int. Symp. Comput. Arch. (ISCA 2004)*, Jun. 2004, pp. 338–349.

[20] G. Keramidas, V. Spiliopoulos, and S. Kaxiras, "Interval-based models for run-time DVFS orchestration in superscalar processors," in *Proc. ACM Int. Conf. Computing Frontiers (CF'10)*, May 2010, pp. 287–296.

[21] W. Kim, D. Brooks, and G.-Y. Wei, "A fully-integrated 3-level DC-DC converter for nanosecond-scale DVFS," *IEEE J. Solid-State Circuits (JSSC)*, vol. 47, no. 1, pp. 206–219, Jan. 2012.

[22] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks, "System level analysis of fast, per-core DVFS using on-chip switching regulators," in *Proc. 14th IEEE Int. Symp. High Perf. Comput. Arch. (HPCA-14)*, Feb. 2008, pp. 123–134.

[23] R. Kumar and G. Hinton, "A family of 45nm IA processors," in *2009 IEEE Int. Solid-State Circuits Conf. (ISSCC 2009) Digest Tech. Papers*, Feb. 2009, pp. 58–59.

[24] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden, "IBM POWER6 microarchitecture," *IBM J. of Research and Develop.*, vol. 51, no. 6, pp. 639–662, Nov. 2007.

[25] S. J. Lee, H.-K. Lee, and P.-C. Yew, "Runtime performance projection model for dynamic power management," in *Advances in Comput. Syst. Arch. 12th Asia-Pacific Conf. (ACSAC 2007) Proc.*, ser. Lecture Notes in Computer Science, Aug. 2007, vol. 4697, pp. 186–197.

[26] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. 42nd ACM/IEEE Int. Symp. Microarchitecture (MICRO-42)*, Dec. 2009, pp. 469–480.

[27] P. Michaud, A. Seznec, and S. Jourdan, "Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors," in *Proc. 1999 Int. Conf. Parallel Arch. and Compilation Techniques (PACT'99)*, Oct. 1999, pp. 2–10.

[28] *MT41J512M4 DDR3 SDRAM Datasheet Rev. K*, Micron Technology, Inc., Apr. 2010, http://download.micron.com/pdf/datasheets/dram/ddr3/2Gb_DDR3_SDRAM.pdf.

[29] M. Moeng and R. Melhem, "Applying statistical machine learning to multicore voltage and frequency scaling," in *Proc. ACM Int. Conf. Computing Frontiers (CF'10)*, May 2010, pp. 277–286.

[30] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," HP Laboratories, Tech. Rep. HPL-2009-85, Apr. 2009.

[31] D. B. Noonburg and J. P. Shen, "Theoretical modeling of superscalar processor performance," in *Proc. 27th ACM/IEEE Int. Symp. Microarchitecture (MICRO-27)*, Nov. 1994, pp. 52–62.

[32] H. Patil, R. S. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation," in *Proc. 37th ACM/IEEE Int. Symp. Microarchitecture (MICRO-37)*, Dec. 2004, pp. 81–92.

[33] S. Rixner, W. J. Dally, U. J. Kapasi, P. R. Mattson, and J. D. Owens, "Memory access scheduling," in *Proc. 27th Int. Symp. Comput. Arch. (ISCA 2000)*, Jun. 2000, pp. 128–138.

[34] B. Rountree, D. K. Lowenthal, M. Schulz, and B. R. de Supinski, "Practical performance prediction under dynamic voltage frequency scaling," in *2011 Int. Green Computing Conf. and Workshops (IGCC'11)*, Jul. 2011.

[35] Y. Sazeides, R. Kumar, D. M. Tullsen, and T. Constantinou, "The danger of interval-based power efficiency metrics: When worst is best," *Comp. Arch. Lett. (CAL)*, vol. 4, no. 1, Jan. 2005.

[36] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *Proc. 30th Int. Symp. Comput. Arch. (ISCA 2003)*, San Diego, California, Jun. 2003, pp. 336–347.

[37] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vemon, and D. A. Wood, "Analytic evaluation of shared-memory systems with ILP processors," in *Proc. 25th Int. Symp. Comput. Arch. (ISCA 1998)*, Jun. 1998, pp. 380–391.

[38] V. Spiliopoulos, S. Kaxiras, and G. Keramidas, "Green governors: A framework for continuously adaptive DVFS," in *2011 Int. Green Computing Conf. and Workshops (IGCC'11)*, Jul. 2011.

[39] E. Sprangle and D. Carmean, "Increasing processor performance by implementing deeper pipelines," in *Proc. 29th Int. Symp. Comput. Arch. (ISCA 2002)*, Jun. 2002, pp. 25–34.

[40] J. Tendler, J. S. Dodson, J. S. Fields Jr., L. Hung, and B. Sinharoy, "POWER4 system microarchitecture," *IBM J. of Research and Develop.*, vol. 46, pp. 5–25, Oct. 2001.

[41] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: A simulation framework for CPU-GPU computing," in *Proc. 21st Int. Conf. Parallel Arch. and Compilation Techniques (PACT'12)*, Sep. 2012, pp. 335–344.

[42] F. Vandeputte, L. Eeckhout, and K. De Bosschere, "A detailed study on phase predictors," in *Proc. 11th Int. Euro-Par Conf. Parallel Process. (Euro-Par 2005)*, Aug. 2005, pp. 571–581.