Execution Drafting: Energy Efficiency Through Computation Deduplication

Michael McKeown, Jonathan Balkind, David Wentzlaff
Princeton University
Princeton, NJ. USA
{*mmckeown,jbalkind,wentzlaf*}*@princeton.edu*

***Abstract*—Computation is increasingly moving to the data center. Thus, the energy used by CPUs in the data center is gaining importance. The centralization of computation in the data center has also led to much commonality between the applications running there. For example, there are many instances of similar or identical versions of the Apache web server running in a large data center. Many of these applications, such as bulk image resizing or video transcoding, favor increasing throughput over single stream performance. In this work, we propose Execution Drafting, an architectural technique for executing identical instructions from different programs or threads on the same multithreaded core, such that they flow down the pipe consecutively, or draft. Drafting reduces switching and removes the need to fetch and decode drafted instructions, thereby saving energy. Drafting can also reduce the energy of the execution and commit stages of a pipeline when drafted instructions have similar operands, such as when loading constants. We demonstrate Execution Drafting saving energy when executing the same application with different data, as well as different programs operating on different data, as is the case for different versions of the same program. We evaluate hardware techniques to identify when to draft and analyze the hardware overheads of Execution Drafting implemented in an OpenSPARC T1 core. We show that Execution Drafting can result in substantial performance per energy gains (up to 20%) in a data center without decreasing throughput or dramatically increasing latency.**

***Keywords*-Energy Efficient Computing; Multithreading; Data Center Computing; Cloud Computing; Microarchitecture; Computation Deduplication; Energy Efficiency**

I. Introduction

The computing industry has seen a huge shift of computation into large data centers and the Cloud [1], [2]. Reducing data center computers' energy use can not only save the data center operator money, but also reduce the environmental impact of computing and enable additional computation within the same power budget. Data center computing is typically different than desktop, mobile, or small-scale server computing in that applications are often optimized for throughput over absolute performance and the application mix in the data center can be more homogeneous. Optimizing processor energy is important because the processor consumes 20-30% of server power [3].

There are many examples of application commonality in a data center. Large websites typically scale out many exact replicas of a server configuration across a data center, utilizing only a modest number of applications which might include a load balancer, front end webserver, business logic, memory caching layer, and a database. In more data-intensive applications such as MapReduce [4] or Dryad [5], it is common for all of the Mappers to be executing the same map program and all of the Reducers to be using the same reduction program. There is also significant batch/bulk computation in the form of resizing of images or compressing of video. These bulk workloads have significant commonality and are not overly latency sensitive. Even in heterogeneous settings such as Infrastructure as a Service (IaaS) Clouds [6], [7] or server consolidation workloads, it is common for the platform (OS) and applications to be the same or similar, thereby exhibiting a large amount of commonality.

Much work has gone into reducing energy in computing and in particular in the data center. Techniques such as DVFS [8], [9], reducing energy use of idle machines [3], making computing systems energy proportional [10], [11], exploiting Dark Silicon [12]–[14], and energy aware processor design [12], [15]–[20] all have had significant impact.

In this work, we investigate a new way to reduce energy by deduplicating computation. In particular, we propose a technique called Execution Drafting (ExecD) which identifies and exploits commonality between applications executing on a multithreaded processor. Execution Drafting works by finding two or more applications which are executing the same or similar instruction sequences, but potentially have different data. A hardware synchronizer delays execution of one or more processes or threads to allow the execution points in the candidate programs to align. Once aligned, instructions flow down the processor pipeline such that the first instruction trailblazes while subsequent instructions can **draft** behind the first instruction, taking advantage of significant energy savings, as shown in Figure 1a. These energy savings can be achieved because the instruction, opcode, and register identifiers for drafted instructions are known to be the same as trailblazers, thereby saving energy for instruction fetch, instruction decode, and pipeline control. In addition to shutting down large portions of the front-end of a pipelined processor, Execution Drafting aligns programs which can potentially save energy in the register file, execution units, and commit portions of the pipeline when instructions that execute share the same operand values, such as when loading constants. If the control flow of the drafting programs diverge, an Execution Drafting processor simply reverts to

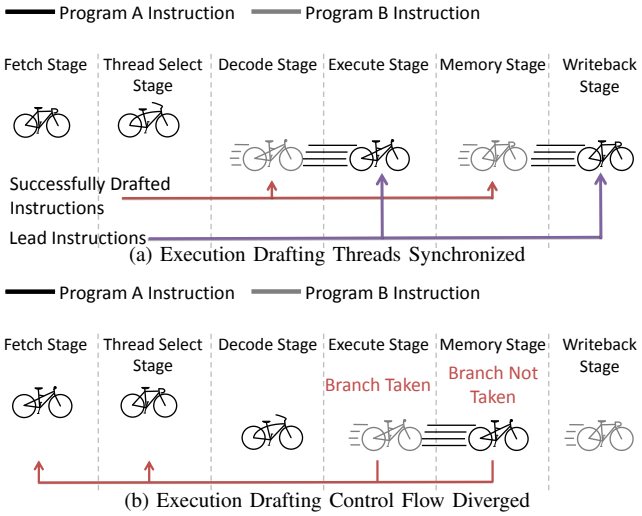
efficient time multiplexing of execution resources as is done in a multithreaded or time-shared processor.

Unlike many previous energy saving and parallelization techniques, Execution Drafting is able to exploit commonality across programs even if they do not contain the same control flow, data, or even code. For instance, Execution Drafting is capable of finding and exploiting commonality in order to save energy across applications within different Virtual Machines (VMs), across different versions of the same program, and even across completely different programs when they execute similar code, as in a common shared library.

The energy efficiency gains of Execution Drafting come at the expense of potential additional program latency. Execution Drafting will delay a program in order to synchronize execution. While this introduces latency, throughput is not drastically affected as the non-waiting thread/program will continue to execute. Although, throughput can be slightly positively or negatively affected due to memory system access reordering. We leverage an insight from previous research [21] by investigating how the introduction of a small and boundable amount of latency can lead to significantly improved energy efficiency in the data center. The addition of latency, without degradation of throughput, can be appropriate for batch applications or even in interactive jobs where the additional latency remains within the Service Level Agreement (SLA), as in many web serving applications where the network latency dominates.

Our work is closely related to previous work in exploiting identical instructions in a processor [22]–[26] (MMT, MIS, DRR, DIR, Thread Fusion) with some key differences. Unlike previous work, Execution Drafting exploits identical instructions among the same applications (or threads within the same application), but is broader and can also draft **different programs**. This enables Execution Drafting to synchronize code at different locations in the same program or completely different programs. Synchronization mechanisms invented in previous work rely solely on program counter (PC) matching, are not scalable, and are limited by the sizes of their structures. Execution Drafting includes synchronization mechanisms that overcome this. Unlike previous work, Execution Drafting does not track and match operand values to reuse instruction results. We are able to achieve the energy savings previous works achieve with operand matching without expensive RAM lookups, operand comparison, and load address checks. In contrast to Execution Drafting, which optimizes for energy efficiency, previous work trades extra energy to achieve performance gains. Last, we focus our study on small, in-order, low-power, throughput oriented cores, while previous work uses large out-of-order (OoO) cores. This is because we target data centers and our approach echoes the recent interest in utilizing ARM cores and Microservers in the data center.

The contributions of this work include the formulation



(a) Execution Drafting Threads Synchronized

(b) Execution Drafting Control Flow Diverged

Figure 1: High-level Execution Drafting Concept

of computational deduplication. We demonstrate this idea through the example of Execution Drafting. We present a detailed architecture of how to implement Execution Drafting in the OpenSPARC T1 multithreaded processor. We evaluate the energy benefit of Execution Drafting, different synchronization schemes, and the overheads. We show that Execution Drafting can achieve up to 20% $\frac{performance}{energy}$ gain on real world applications with a low latency increase and area overhead.

II. Computation Deduplication

Much work has been done in the area of data deduplication for storage, back-up, replication, and network transfer systems [27]–[42]. In the process of storing or transmitting data, unique chunks are identified and stored or transmitted accordingly. As subsequent chunks enter the system, if they are found to duplicate a previous chunk, the data in the chunk is not replicated but a pointer to the previous identical chunk is stored or transmitted instead. The key insight that makes this technique beneficial is that recurring chunks occur frequently. For example, with nightly back-ups, only a small portion of the data may change in any given day.

While data deduplication is used widely in storage and network systems, the analog in computation lacks attention. The idea behind computation deduplication is to identify identical or similar pieces of computation, and exploit their similarity to reduce the work required to perform it. While it is not a novel idea, we attempt to identify it as a general concept common to existing energy saving or performance enhancing techniques. In doing so, we hope to stimulate further research in this area, as cloud environments show great potential for computation deduplication.

In an Infrastructure as a Service (IaaS) setting, such as Amazon's Elastic Compute Cloud (EC2) [6], there are many VMs hosted within the same data center. However, many

customers utilize similar (or identical) versions of Linux or other commonly used software such as Apache [43] or MySQL [44]. In MapReduce [4] jobs, each Map and Reduce task, by definition, execute the same code. Many social networking websites store multiple resolutions of user images to prevent dynamic resizing when serving webpages [45]. Thus, when users upload images, they must be resized and stored. The resultant large amount of identical code is ripe for computation deduplication.

Commonality also exists between threads within a program. For example, many web servers launch a new thread for each client. Each thread performs the exact same computation, serving the same webpage, but to different clients. This type of parallelism also occurs in database servers.

Computation deduplication can enable performance enhancements, energy savings, or both. Generally, these benefits can be achieved by doing less work to execute identical copies of a computation. In this work, we focus on one instance of computation deduplication, Execution Drafting.

III. Execution Drafting Architecture

Execution Drafting (ExecD) is an example of computation deduplication which targets maximizing $\frac{performance}{energy}$ (similar to reducing energy-delay-product (EDP), but accounts for throughput). More specifically, it aims to reduce energy consumption while minimizing the performance impact. Note that this metric allows us to trade performance degradation for larger energy savings, resulting in a net gain in $\frac{performance}{energy}$. Performance can be either single-threaded latency or overall throughput. We optimize for both interpretations but emphasize throughput since many data center applications favor throughput over single-threaded performance [46]. ExecD does not considerably impact throughput, but does have the potential to increase single-threaded latency.

A. Processor Architecture

The high-level concept of ExecD is illustrated in Figure 1. In the figure, instructions from different threads are represented by different shades (dark, light) and different instruction types are represented by different types of bicycles (road, mountain, cruiser). ExecD attempts to synchronize similar programs or threads on a multithreaded core such that duplicate instructions are issued consecutively down the pipe, as shown in Figure 1a.

Duplicate Instructions: are defined as one of two categories: *Full duplicates* or *partial duplicates*. Full duplicates have identical machine code. Partial duplicates have the same opcode, but the full machine code does not match.

When duplicate instructions are executed consecutively, transistor switching is reduced. Decoding of a duplicate instruction is identical to the lead instruction; thus it only needs to be performed once. All control logic, i.e. ALU opcodes, multiplexer select signals, etc., will not switch. Only the data may differ between the execution of identical instructions. Thus, everything other than data buses remains static. In fact, it is possible that some data may be shared between threads. Immediate constants may be identical, threads may share memory, virtual memory addresses may be the same, and some data values (zero) are common.

This concept of reducing switching, thereby reducing energy, by executing duplicate instructions consecutively is called **drafting**, as indicated in Figure 1a. This term comes from the analog in racing, where one racer reduces drag by following closely behind another racer. We call the first instruction or thread in a set of drafted instructions the leader instruction or thread, and the instructions that follow behind, the follower instructions or threads (see Figure 1a).

Reducing switching is not the only energy saving component of ExecD. Knowing that identical threads are synchronized means that only one stream of instructions needs to be fetched. Thus, the fetch stage can be disabled for all follower threads. This is depicted in Figure 1a, where instructions are only fetched for the leader thread but are executed for the leader thread and each follower thread. We call this double-stepping or repeating the instruction.

Data dependent control flow may cause threads to diverge. These divergences require the fetch stage to be re-enabled for all diverged threads. This case is illustrated in Figure 1b, showing the cycle after Figure 1a. In the figure, mountain bikes represent branch instructions. The two successfully drafted branch instructions take different branch paths. ExecD must detect this case and notify the thread select and fetch stage. The thread select stage stops double-stepping instructions and the fetch stage begins fetching for the follower threads again. Note that interrupts and other control flow altering events also cause divergences.

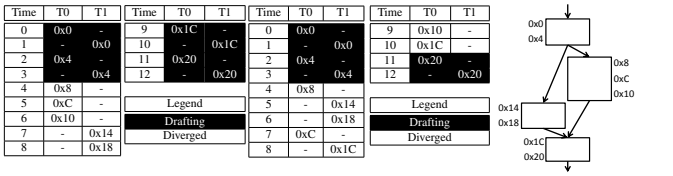
Data dependent control flow thus limits the energy savings ExecD is able to achieve. Consequently, the key to ExecD's energy savings is the amount of time the processor spends drafting. Thus, robust synchronization mechanisms are integral to ExecD's success.

B. Thread Synchronization Techniques

In this section we describe three thread synchronization methods (TSMs) that can be implemented in hardware: stall, random, and hybrid. The methods are simple heuristics, however, results show they perform well.

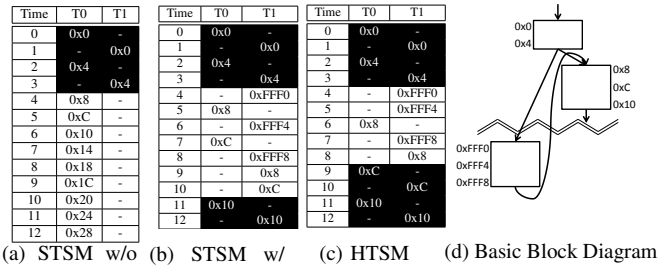
1) Stall Thread Synchronization Method: The stall thread synchronization method (STSM), relies on thread PCs for synchronization. When STSM encounters a control flow divergence, it compares threads' PCs. It selects the thread with the lowest PC and stalls the others in an attempt to accelerate the lagging thread to synchronize with the others.

This method works well on common code structures such as if-else and loop structures. Figure 2 shows the execution of an if/else structure, where the two threads take different paths. The thread that takes the path earlier in program order



(a) STSM (b) RTSM (c) Basic Block Diagram

Figure 2: If/Else Structure ExecD Example



(a) STSM w/o Threshold (b) STSM w/ Threshold (c) HTSM (d) Basic Block Diagram

Figure 3: Function Call ExecD Example

(T0 in Figure 2a) will be selected to execute and exit the if-else structure first, jumping to a larger PC than the other thread. STSM will now select the thread that took the other path (T1 in Figure 2a), as it now has the minimum PC, until it exits the if-else structure. The threads will synchronize at the end of the if-else structure. Similar logic explains why STSM also performs well on loops. If two threads execute different numbers of iterations of a loop, say 5 iterations and 10 iterations, the threads should stay synchronized for the first 5 iterations, after which one thread will exit the loop. This thread will now have a PC larger than the other thread, and STSM will select the other thread until it finishes its iterations. The threads will synchronize at the end of the loop structure.

There are caveats to STSM. Consider the case in Figure 3 where one thread takes a function call and the other does not. Assume the function code has a larger PC than any other threads' PC, i.e. it is a shared library or sits higher in memory, as shown in Figure 3d. Thus, STSM will stall the thread in the function call (T1 in Figure 3) until the other thread reaches a PC larger than it. Since the function code is placed at a large PC, this may not occur until the other threads terminate, depicted in Figure 3a. Thus, in order to handle this case, we set a threshold value for STSM. If the difference between PCs is outside this threshold, STSM resorts to alternating the issue of instructions from each thread. The threads will alternate until the thread that took the function call returns, in which case the PCs are within the threshold again and STSM works as described previously, as shown in Figure 3b. Choosing this threshold value is ad hoc, but empirically 100 works well and is used in this work.

The other caveat to STSM involves livelock. Consider two drafting threads trying to grab a spin lock that protects a critical section. One thread will acquire the lock and enter the critical section, and the other will continue to spin. STSM will always select the thread that is spinning, since its PC is lower, resulting in livelock since the lock holding thread is stalled. To avoid this, we implement a ratioed thread selection policy. If the minimum PC has been chosen so many times without synchronizing, select the maximum PC once and go back to selecting the minimum PC. This guarantees forward progress.

STSM breaks down when drafting different code. The same PC in two threads executing different code may not point to the same instruction. Thus, STSM may not be able to draft or disable fetch despite having matched up the PCs as we verify the instructions and the physical addresses match before drafting. This reliance on PCs is a shortcoming of STSM that other TSMs attempt to overcome.

2) Random Thread Synchronization Method: The random thread synchronization method (RTSM) is a simple mechanism. When RTSM encounters a divergence in the control flow of threads, it chooses a thread to execute at random until threads are synchronized again. This effectively varies thread execution points by sliding the threads by each other until a convergence point is found. The intuition behind this is to model random changes to code, for example between two different versions of the same program. RTSM attempts to fix the shortcoming that STSM has with drafting different programs or the same program with different versions and works well in practice. However, it does not necessarily perform optimally on common code structures such as if/else, evident from Figure 2b, and thus suffers when drafting the same program with different inputs.

3) Hybrid Thread Synchronization Method: The hybrid thread synchronization method (HTSM) attempts to inherit the positive properties from STSM and RTSM. HTSM is identical to STSM with two key differences. Recall the function call example in Figure 3 which motivated the addition of the threshold to STSM. In the case of being outside the threshold, HTSM chooses a random thread to issue instead of alternating the issue of instructions from each thread, as shown in Figure 3c. This hybridizes STSM and RTSM.

The other key difference is that HTSM adds some flexibility to the PC based synchronization in STSM. HTSM looks for the PCs of the threads matching (STSM's goal), but the instructions and instruction physical addresses not matching. This indicates the threads are executing different code. HTSM detects this and tries to model changes to code, such as additions or removals. This is modeled using an offset that is added to all PC comparisons and differences. When HTSM encounters this case, it attempts to find the correct offset to synchronize to. It does this by alternating the execution of code from each thread, executing each thread at exponentially increasing rates. It executes thread 0 for 1 instruction, thread 1 for 2 instructions, thread 0 for 4 instructions, thread 1 for 8 instructions, etc. This effectively

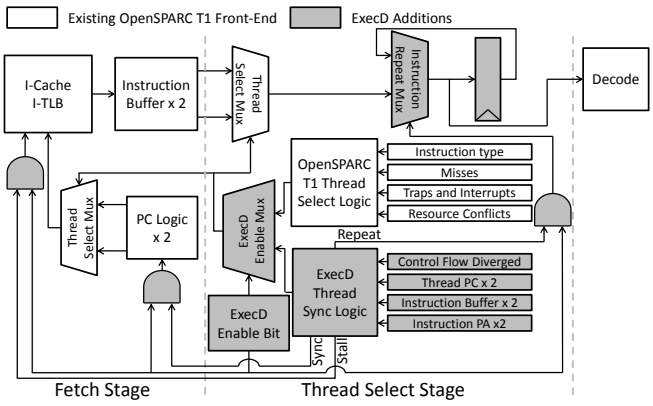


Figure 4: ExecD Changes to OpenSPARC T1 Front-End

slides the threads past each other, looking for the correct offset that matches the change to the code. HTSM continues to do this until it finds a synchronization point, at which it updates the offset with the difference in the PCs. All synchronization from that point on is based on synchronizing to that PC offset.

4) Note on Thread Synchronization Methods: While TSMs try to achieve high synchronization, energy savings can still be had when threads are not synchronized. When threads are diverged, duplicates still occur. In these cases, we prefer to draft duplicates over synchronizing (i.e. PCs don't match but is a duplicate), leading to energy savings.

IV. Execution Drafting Implementation

Figure 4 depicts modifications to the front-end of the OpenSPARC T1 [47] core for ExecD. The OpenSPARC T1 is an open source version of Sun's UltraSPARC™ T1 [48]–[50]. The core is reduced from 4 to 2 threads for this work. This core is used since it is multithreaded, making it amenable to ExecD. We modified the Verilog RTL of the core to implement ExecD and we plan to tape out a chip utilizing the modified core. The additions to the OpenSPARC T1 processor are small, however, we evaluate the overheads in our results.

The OpenSPARC T1 fetches two instructions per cycle and places them into the instruction buffer for their corresponding thread in the fetch stage. Instructions are always fetched from the thread that is issuing. In the thread select stage, one instruction from one thread is selected to issue. This decision is based on a weighted round-robin thread selection policy giving priority to the least recently executed thread. In addition, threads may become unavailable due to long-latency instructions, cache misses, traps, or resource conflicts [51].

Modifying the OpenSPARC T1 to implement ExecD requires replacing the round robin thread selection policy with a policy that attempts to synchronize the threads, discussed in Section III-B. As shown in Figure 4, a multiplexer (ExecD Enable Mux) is added to the thread selection decision path which selects between the default OpenSPARC T1 thread selection policy and the ExecD thread synchronization policy. This allows ExecD to control which threads get executed when. The multiplexer is controlled by an ExecD enable bit.

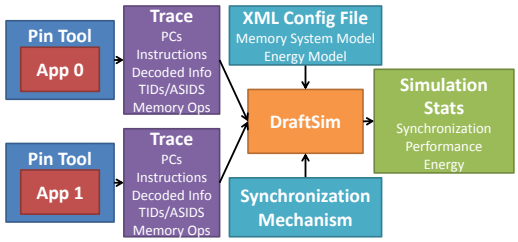


Figure 5: DraftSim Architecture

The ExecD Thread Synchronization Methods (TSMs) require multiple signals from different portions of the pipeline, as shown in the bottom right of Figure 4. The reason these signals are needed can be understood from the discussion in Section III-B. ExecD requires knowledge of when threads diverge in order to enable the fetching for follower threads and stop the double stepping of instructions. This signal includes branch divergences, interrupts, and hitting page boundaries, as we rely on threads executing from the same code page to determine threads are synchronized. It also requires the thread PCs for synchronization purposes. Thread instructions and the corresponding physical addresses are required to determine when threads are synchronized. Only when the instructions' physical addresses and the instructions themselves match, can ExecD declare being synchronized. This requires that the physical addresses be piped forward (not shown in Figure 4), as they are usually dropped after the tag comparison. Recall, ExecD does not require threads to be synchronized to achieve energy savings, as duplicate instructions that arise when diverged are also drafted, only the fetch stage cannot be disabled.

In order for ExecD to share the fetching of instructions for two synchronized threads and disable fetch for the follower thread, the physical code page must be shared. Physical code pages are shared in most operating systems when multiple instances of the same program are executing, in addition, most Virtual Machine Monitors deduplicate code pages between VMs enabling ExecD to draft threads between VMs. VMware ESX Server and KVM with Kernel Samepage Merging (KSM) enabled both perform this optimization.

The ExecD TSM logic outputs a few signals in order to control the pipeline when drafting: stall, sync, and repeat. The stall signal is wired to the fetch stage. This allows ExecD to prevent fetching for follower threads when drafting. The sync signal goes to the PC logic in order to keep the PCs of follower threads up to date when drafting. When stalling fetch for a follower thread, the PC of the leader thread gets copied into the PC of the stalling thread in fetch.

Table I: Core Model

DraftSim Parameter	Value
CPI (Non-Memory Instructions)	1
L1 Data Cache Size	8KB
L1 Data Cache Associativity	4
L1 Data Cache Line Size	32B
L1 Data Cache Access Latency	1 cycle
L2 Unified Cache Size	384KB
L2 Unified Cache Associativity	12
L2 Unified Cache Line Size	64B
L2 Unified Access Latency	22 cycles
DRAM Access Latency	52 cycles

Table II: Energy Model

Pipeline Stage	Energy per Cycle (nJ/cycle)
Fetch	0.60571875
Thread Select	0.03583125
Decode	0.020475
Execute	0.4675125
Memory	0.3241875
Writeback	0.252525
Whole Pipeline	1.70625

The PC of follower threads needs to stay up to date so that we fetch the correct instructions when re-enabling fetch after encountering a divergence. The repeat signal controls an instruction repeat multiplexer that is added to the datapath to support double stepping instructions. The instruction repeat multiplexer selects between the output of the thread select multiplexer and a flip-flop that latches the previously issued instruction. Thus, when double stepping, the leader instruction is chosen from the thread select multiplexer and follower instructions are selected from the flip-flop, only the thread identifier is changed.

V. Results

We first evaluate commonality in applications. This reference result represents an estimate for the amount of commonality Thread Synchronization Methods (TSMs) can find. Energy savings, performance overheads, $\frac{performance}{energy}$ gains, and area overheads of ExecD are evaluated, as well as the effect that different numbers of drafted threads has on energy savings.

A. Evaluation Methodology

To evaluate ExecD, we built a trace-based simulator, DraftSim, shown in Figure 5. DraftSim accepts traces generated by Pin [52], and models synchronization of the instruction traces using a specified TSM and reports synchronization results. It also accepts an XML configuration file that specifies a memory system model and energy model and simulates the execution of instructions to obtain performance and energy results. The core model assumes a clocks per instruction (CPI) of one for all non-memory instructions. We model the latency and occupancy of the L1 data cache, L2 cache, and memory. The parameters used in the core model for all results are shown in Table I and are modeled after the OpenSPARC T1 core.

We use a simple energy model to calculating energy consumption. The energy model is derived from the power breakdown of the OpenSPARC T1 core in [53] and the power measurements of the UltraSPARC™ T1 chip running at 1.2 Ghz / 1.2V in 90nm technology in [49]. The resulting energy model is presented in Table II. The energy model is used in the simulator as follows. If the threads are synchronized, an instruction is drafted behind another, and the instructions are full duplicates, the energy from fetch, thread select, and decode stages is not added for the duplicate instruction. If two partial duplicates are drafted, the energy from the decode stage is not added for the duplicate instruction. Otherwise, the energy for all stages is added. When the decode stage energy is not added, the register file energy is still added, as it is always accessed. The static power of unused stages is not modeled, however, this should be small and should not dramatically impact results. The simulator's energy model keeps track of whether threads are synchronized and when instructions are drafted. Applications from the SPECint CPU2006 [54], MediaBench I [55], and II [56] benchmarks are used throughout the evaluation. In addition, different version of the Apache web server [43] and GCC compiler [57] are used as benchmarks. In order to evaluate ExecD, we need multiple input sets for each benchmark. Unfortunately, not all applications within a benchmark suite have multiple input sets, so we restrict our evaluation to applications with multiple input sets.

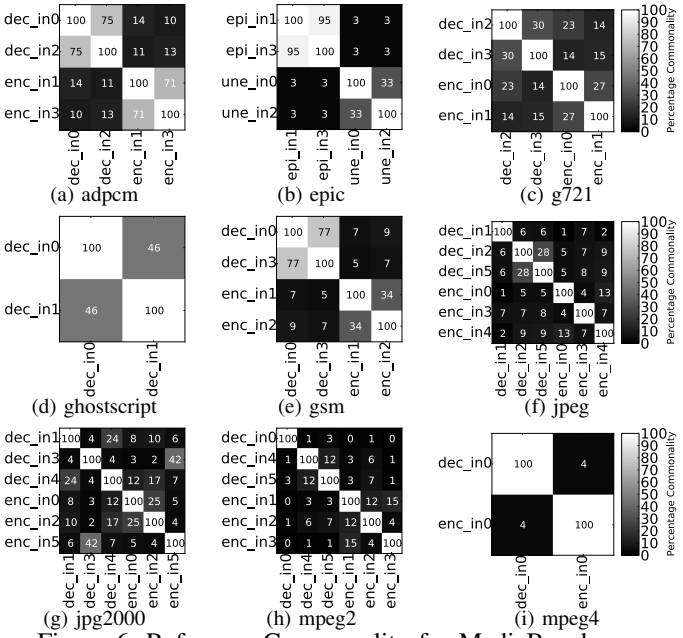


Figure 6: Reference Commonality for MediaBench

B. Peak Commonality Available in Applications

We start by evaluating how much commonality exists between applications, between the same program with different inputs as well as different versions of the same program with different inputs. We implemented a reference TSM in DraftSim to obtain an estimate of the available commonality. The reference TSM operates as follows: When a divergence is encountered, the TSM scans future instructions in the trace until it finds the earliest convergence point. It executes the instructions up until that point, after which the traces are synchronized again. While this greedy TSM is not practical to implement in real hardware, it provides a good estimate of the peak available commonality. We limit the number of

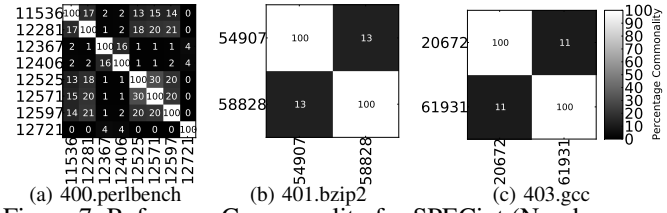


Figure 7: Reference Commonality for SPECint (Numbers on axes represent different inputs)

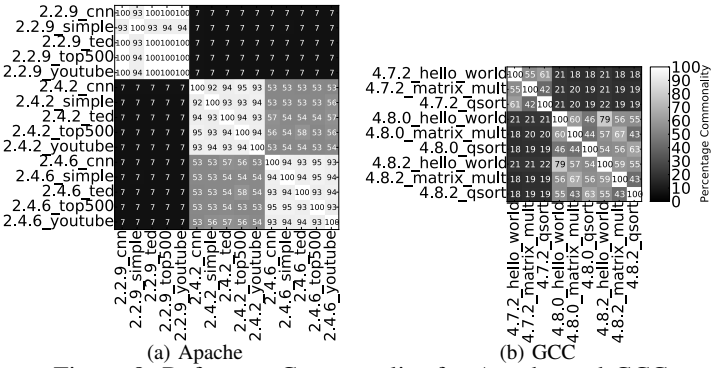


Figure 8: Reference Commonality for Apache and GCC

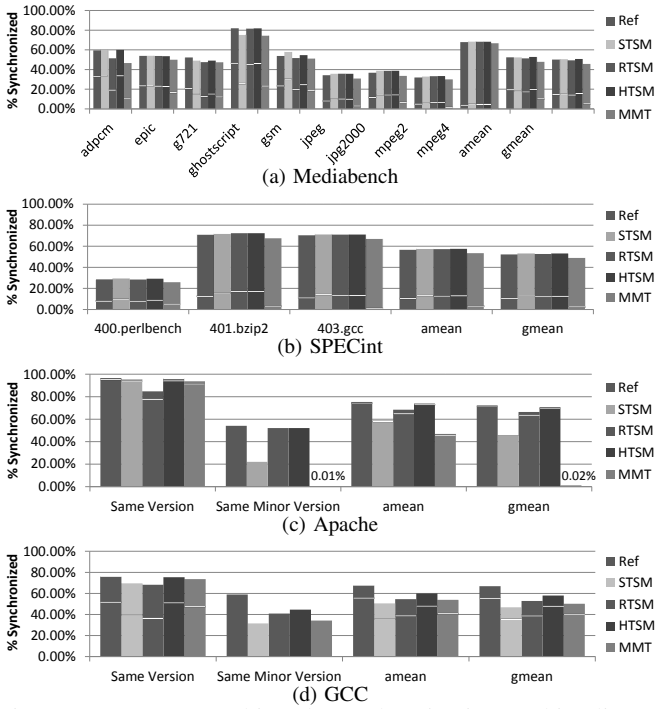


Figure 9: Average Achieved Synchronization. White lines indicate average excluding same program, same input (SPSI)

future instructions searched to 50 million instructions in the reference to bound the memory requirements.

1) Same Program, Different Inputs: Figures 6 and 7 show the reference commonality results for MediaBench I, II, and SPECint, respectively. These graphs show the percentage of execution which is drafted; lighter is better. The actual values are also annotated in each cell, rounded to the nearest integer. On each axis is a benchmark mode (for benchmarks with modes) and input combination. Each box represents the commonality found between drafting the two corresponding benchmark mode and input combinations. The numbers on the axes for SPECint represent different inputs. Some applications, such as adpcm, epic, ghostscript, and gsm, exhibit high commonality when executed in the same mode (i.e. encode/decode), even for different inputs.

However, other applications show sporadic or poor commonality. This is mainly due to irregular or high amounts of data dependent control flow. However, Section V-E shows that, despite the lack of energy benefits from drafting, the performance penalty from doing so is minuscule.

2) Different Program Versions, Different Inputs: We study the commonality in different versions of the same program with different inputs. Figure 8a compares different versions of Apache serving different pages. Figure 8b compares different versions of GCC compiling different applications.

All experiments show available high commonality between the same version with different inputs and versions with the same major and minor version but with different inputs. However, comparing versions across major or minor

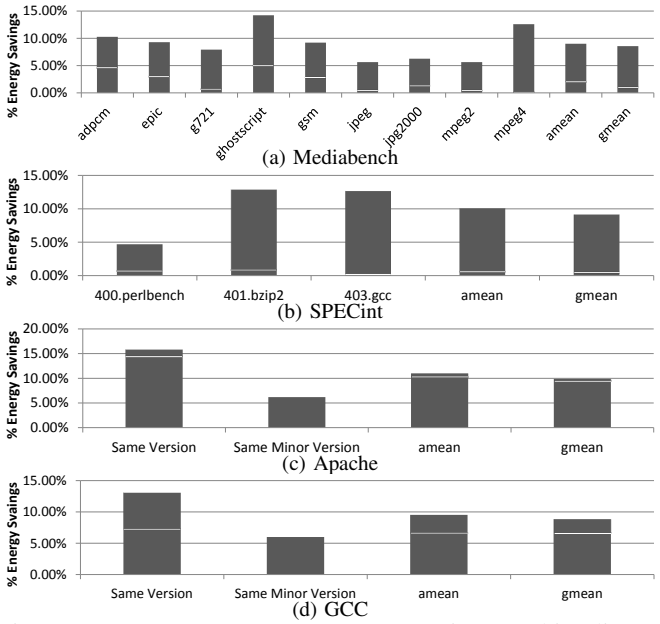


Figure 10: HTSM Average Energy Savings. White lines indicate average excluding SPSI

versions exhibits poor commonality. This is expected, as differing major and minor versions generally indicate large changes to the code, whereas different patch version numbers usually indicate a bug fix or small change [58].

C. Hardware Synchronization Method Performance

In this section, we determine how much commonality can be exploited by our Thread Synchronization Methods (TSMs) and the TSM presented in Minimal Multithreading (MMT) [22]. For a qualitative comparison to MMT, see Section VI. We ran experiments identical to the previous section, with each TSM. Figure 9 shows the average achieved synchronization across different input cross products for each TSM. The white lines indicate averages excluding the same program, same input (SPSI) combinations. Overall, the TSMs achieve comparable synchronization to the reference.

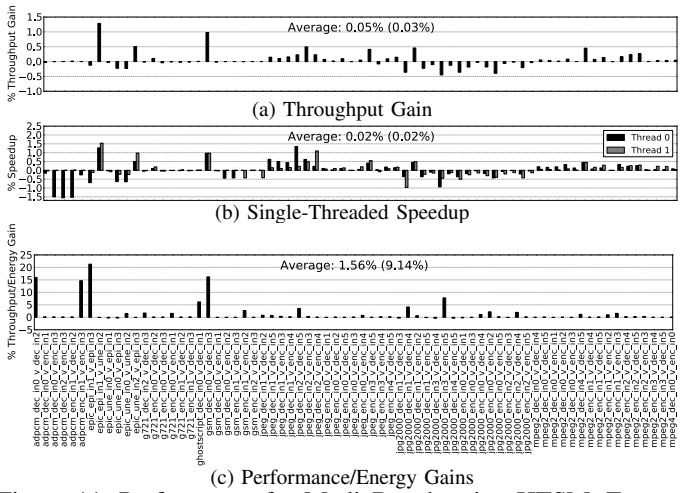
STSM achieves close to the same, if not more, synchronization versus the reference for the same program with different inputs (SPDI). However, it falls short on different versions of Apache or GCC. This is because STSM relies on PCs for synchronization, but in different versions of an application, the same PCs may not point to the same instruction. Nevertheless, it can still find a significant amount of commonality. For instance, drafting Apache 2.4.6 hosting cnn.com against top500.org achieves 94.84% commonality.

RTSM in part makes up for STSM's shortcomings. It achieves close to the same synchronization as the reference for the different program, different input (DPDI) experiments, but generally falls short with SPDI. This is due to RTSM's lack of reliance on PCs, which is beneficial for the DPDI case. However, it may not perform optimally on common code structures like if-else and loops, causing it to perform worse in general than STSM in the SPDI case.

HTSM is successful in hybridizing STSM and RTSM, achieving synchronization comparable to the reference in almost all cases, whether drafting different inputs for the same program or different programs. Note that HTSM does not perform better than other synchronization methods in all cases, such as gsm, where STSM performs better. Thus, HTSM is not the obvious choice in all cases.

The MMT simulations used a Fetch History Buffer (FHB) with 32 entries (as in the original work) and assumed only taken branch targets were recorded in the FHB. In addition, every instruction's PC was checked against the other thread's FHB for potential convergence points.

MMT performs similarly to STSM on the SPSI experiments. It performs well on common code structures like STSM, but may miss opportunities to synchronize short code snippets. In addition, MMT may not reconverge until shortly after optimal reconvergence points (the first common instruction after a divergence), while STSM achieves tighter synchronization. MMT performs relatively poorly in SPDI experiments, as it relies on PC matching (like STSM) and is limited by its FHB size. A larger FHB may increase MMT's ability to find synchronization points, especially for SPDI cases, where common code may be distanced further apart. STSM is not limited by the size of any structures, and therefore may perform better here.

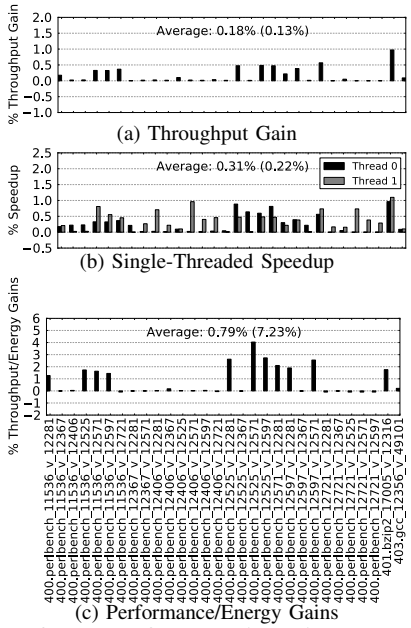


(a) Throughput Gain

(b) Single-Threaded Speedup

(c) Performance/Energy Gains

Figure 11: Performance for MediaBench using HTSM. Excludes SPSI for visibility. Averages: w/o SPSI (w/ SPSI)



(a) Throughput Gain

(b) Single-Threaded Speedup

(c) Performance/Energy Gains

Figure 12: Performance for SPECint using HTSM. Excludes SPSI for visibility. Averages: w/o SPSI (w/ SPSI)

D. Energy Savings

We present energy savings from ExecD in this section. To be brief, only HTSM will be used for the remainder of the results. However, energy savings are similar for STSM and RTSM. The average energy savings from HTSM for each benchmark are presented in Figure 10, where the white lines indicate averages without the SPSI combinations. Results are presented as a percentage saved versus the energy needed to execute the benchmarks using fine-grain multithreading (FGMT) - which cannot disable fetch, but may save energy by issuing coincidental identical instructions consecutively.

In general, the energy savings HTSM achieves is correlated to the amount of synchronization it achieves, between

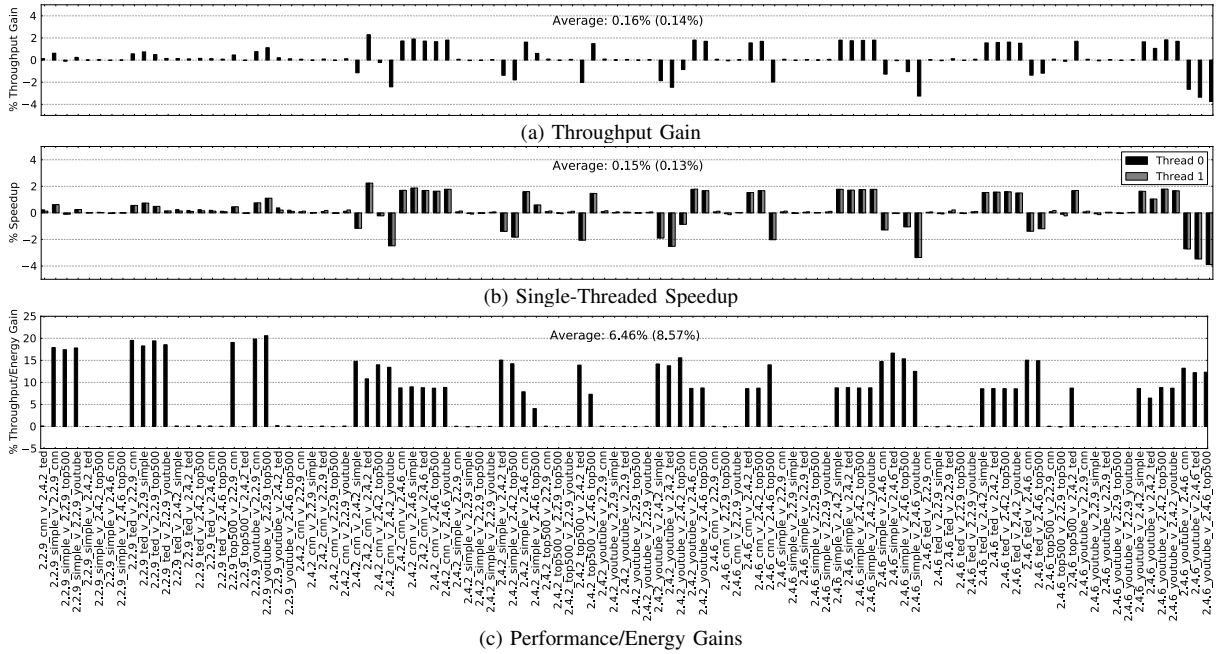


Figure 13: Performance for Apache using HTSM. Excludes SPSI for visibility. Averages: w/o SPSI (w/ SPSI)

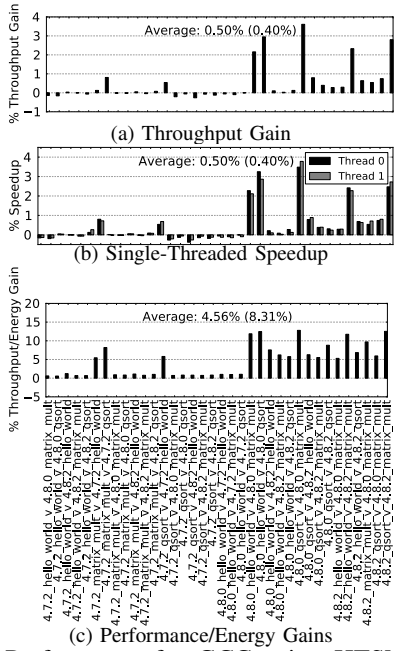


Figure 14: Performance for GCC using HTSM. Excludes SPSI for visibility. Averages: w/o SPSI (w/ SPSI)

5% and 20% energy savings for most benchmarks. This is evident when comparing Figures 9 and 10. Note that there can be exceptions: If many of the drafted instructions are partial duplicates, it will not save as much energy as with full duplicates. While there are achievable energy savings from drafting partial duplicates, they are comparably small as we model ExecD only saving energy for the decode stage, which does not contribute a large portion to the core energy (see Table II).

The energy results from DraftSim are conservative. DraftSim does not take into account energy savings from the execute, memory or writeback stages if data happens to be shared between drafted instructions. In addition, energy saved from the control logic, i.e. ALU opcode, multiplexer control signals, etc., is not considered. However, we do not model the energy penalty from our TSM logic, but do take into account energy from the thread select stage with the OpenSPARC T1 thread select logic. Our TSM logic should not consume more power than the OpenSPARC T1 thread select logic (based on the area results in Section V-G), thus it is conservative to use the thread select logic energy as a substitute for the TSM logic energy. As such, the actual energy savings may be higher than what is presented.

E. Performance Results

This section examines throughput and single stream performance, but we emphasize that throughput is more significant for data center applications with QoS bounds. Only HTSM results are presented; STSM and RTSM are similar.

1) Throughput: Throughput gains as a percentage of FGMT throughput for each benchmark using HTSM are shown in Figures 11a, 12a, 13a, and 14a along with averages. Two averages are presented, excluding SPSI (first average), and including SPSI (second average in parentheses). The data for SPSI is excluded from the plots for visibility, but is trivial to infer since the execution is the exact same when drafting SPSI.

Throughput gains are in the range of [-5%, 4%] across all benchmarks. We were surprised to find that ExecD can achieve throughput gains, even for the majority of drafting combinations within a certain benchmark suite, as shown in Figure 12a and 14a. Changes in throughput are mainly due to cache performance, as for both ExecD (regardless of the TSM) and FGMT, an instruction is selected to be executed every cycle if ready. ExecD TSMs will never stall both threads on a given cycle (unless stalling for a different reason). However, in some cases, ExecD TSMs align instructions such that cache performance is improved and, in other cases, they do not. Nevertheless, throughput is not largely affected by ExecD TSMs. Thus, we can be aggressive in our attempts to draft applications since, even if drafting does not achieve any energy savings, the performance will not be significantly degraded.

An interesting future technique to explore is to prefetch data from drafted threads based on cache misses of duplicate instructions. This could lead to improved cache performance, giving ExecD a throughput advantage over FGMT. This idea is related to previous work in Helper Threads or pre-execution [59]–[63].

2) Single-Threaded Performance: The single-threaded performance results are presented as a percentage speedup compared to FGMT. They are shown for each benchmark using HTSM in Figures 11b, 12b, 13b, and 14b, with the same averages and caveats as the throughput figures. Results for both drafted threads are shown. Our TSMs can negatively affect single-threaded performance as they stall threads to keep them synchronized. This increases latency over FGMT.

Single-threaded performance speedup is in the range of [-5%, 4%] across all benchmarks. Again, we were surprised to see ExecD achieve speedups, even for the majority of drafting combinations within a certain benchmark suite, as shown in Figure 12b and 14b. This could be a result of cache performance improvements or the way HTSM schedules instructions. HTSM and STSM could stall one thread for many cycles, while executing instructions from the other thread. This means the thread that was not stalled has a lower latency. RTSM shows single-threaded latency closer to that of FGMT, as it chooses instructions to execute at random. Thus, it averages out to FGMT in the long term.

The take away: single-threaded performance is not dramatically impacted by ExecD. As with throughput, this provides some freedom in scheduling threads to be drafted.

F. Performance/Energy Results

$\frac{Performance}{Energy}$ is the key metric that ExecD optimizes for. In this section we interpret performance as throughput, since it is more pertinent to data center applications. Figures 11c, 12c, 13c, and 14c show $\frac{throughput}{energy}$ gains for each benchmark using HTSM. $\frac{Throughput}{Energy}$ gains are up to 20% for drafting threads that exhibit high commonality. For threads that do not exhibit high commonality, there is nominal gain or loss, meaning it is not detrimental to draft these types of programs. For example, drafting Apache instances within the same major and minor versions achieves high $\frac{throughput}{energy}$ gains, however, drafting instances with different minor versions leads to nominal $\frac{throughput}{energy}$ gain.

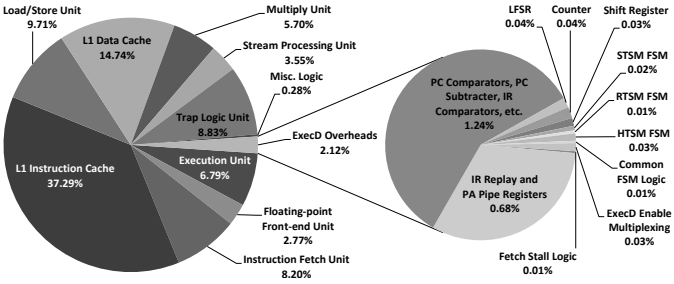


Figure 15: ExecD Area Overhead in 2-threaded OpenSPARC T1 Core. Excludes L2 Cache.

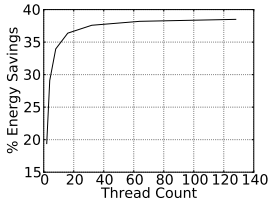


Figure 16: Energy Savings as a Function of Thread Count Executing 100 Instructions per Thread

G. Execution Drafting Area Overheads

We synthesized the Verilog RTL for a 2-threaded OpenSPARC T1 core with ExecD modifications with Synopsys Design Compiler using the IBM 32nm SOI process at 1.2Ghz/0.9V. The SRAM area for caches was estimated using the IBM RAM compilers. Note, the L2 cache area is excluded. In addition, the area of the register files, TLBs, and store buffer are excluded as they are either highly ported or use CAMs (making it difficult to estimate the area). The area overheads for ExecD are small, 2.12% of the core area as shown in Figure 15. Most of the area overheads are due to comparators for PCs and instructions, a subtractor for the PCs, and pipe registers for physical addresses. Note, this implementation includes all three TSMs, which allows for many of the hardware resources to be shared between them, e.g. HTSM and RTSM can share a linear feedback shift register (LFSR). The area result correlates well with the power model we used, lending credence to its validity.

H. Execution Drafting with Higher Thread Counts

Thus far, we have only presented results for drafting two threads. However, it is possible to draft higher thread counts. Figure 16 shows the energy savings from ExecD as a function of drafted thread count. This was generated by drafting 100 instructions from each thread and comparing against the energy from executing the instructions from each

thread sequentially. For two threads, the energy savings is ~20%, which is our experimental upper bound. Yet, greater energy savings could be had by drafting more threads. Energy savings increases dramatically until 32 threads, after which we see diminishing returns. This is because there is more opportunity to find commonality and greater energy savings can be harvested with larger blocks of drafted threads. The benefits diminish since we only save front-end energy. We leave developing scalable TSMs and evaluating the energy savings to future work.

VI. Related Work

Minimal Multithreading (MMT) [22] is the work most related to ExecD, but has many key differences. Unlike MMT, ExecD not only drafts the same applications, but is broader and can draft different programs as we presented. The MMT synchronization scheme relies only on PC matching, while ExecD is able to synchronize common instruction sequences at different PCs. This enables ExecD to synchronize code at different locations in the same program or completely different programs, which MMT is unable to do.

Our design is simpler and more scalable than MMT. As we show in Figure 16, drafting larger numbers of threads leads to larger energy savings. When drafting 64 threads, MMT's synchronization method requires each thread CAM match (comparing) against 63 fetch history buffers (FHBs), which is not scalable. Our approach does not require all-to-all comparisons of PC histories. MMT is limited by the size of the FHBs, which also limits scalability. In fact, MMT's synchronization approach is similar to our reference.

Our approach is different from MMT. Unlike MMT, we do not match operand values to reuse (memoize) instruction results. Instead, we always execute the instruction for each thread. If the operand values are the same, wires do not toggle in the wake of the leader thread, saving energy. Thus, we achieve the energy savings MMT achieves without expensive RAM lookups, operand comparisons, and load address checks. In contrast to ExecD, MMT trades energy for performance while ExecD optimizes for energy efficiency. The MMT synchronization mechanism, which uses a FHB to CAM thread PCs against PCs previously fetched from other threads, is not robust. MMT only records branch targets in the FHB, which means that reconvergence can only occur at branch targets. This prevents MMT from reconverging in short if/else code hammocks. ExecD handles these cases.

Last, we focus our study on small, low-power, throughput oriented cores, while MMT uses large OoO cores. We intentionally chose lean cores like those used in current data center architectures such as Applied Micro, Calxeda [64], Tilera [65], [66], and the new AMD ARM chip for ExecD since we target data centers.

Multithreaded Instruction Sharing (MIS) [23] is also related to ExecD and takes a similar approach to MMT. MIS uses small, throughput oriented cores, like ExecD, but does not perform active synchronization of the threads. MIS relies on small control flow divergences and finding duplicate instructions within a limited window of instructions. Thus, MIS can try to deduplicate code from different programs, but will not be very successful in doing so.

MIS performs operand comparisons in order to reuse (memoize) instruction results from different threads, like MMT. However, MIS claims performance benefits, not energy savings. In fact, they claim energy losses. This is at odds with MMT's findings, which claims both performance and energy benefits. Considering MIS and MMT take the same approach and have similar mechanisms, this is a strange result, but may have to do with MMT using OoO cores and MIS using small, throughput oriented cores.

Thread Fusion [26], unlike drafting, "fuses" threads, meaning the core fetches and decodes only a single copy of an instruction (requiring full duplicates) but issues it to two functional units in the same cycle when possible. Dynamic value reuse may enable using a single functional unit, deduplicating computation and thus saving energy. This requires comparators and tracking in register files and tracking in scoreboards, but enables energy savings in the presented workloads. Thread Fusion uses synchronization points (SPs), added manually, by the compiler, or by OpenMP. The authors also suggest an algorithm for finding new SPs. Thread Fusion relies on PCs, so it is unlikely to work well on different programs or versions.

Molina et al. [24] and Sodani et al. [25] also exploit identical instructions in order to reuse (memoize) instruction results, but only within a single thread in a superscalar. They claim performance benefits, but do not study energy.

Our thread synchronization methods are similar to thread-management techniques [67] like thread delaying (to save energy) and thread balancing (to improve performance). In a controlled environment like the one those techniques utilize, they could be complementary to our own. Since ExecD targets different programs, it can save energy without tracking all threads' progress.

There are many other areas of work related to ExecD. Variations on multithreading architectures, such as the vector-thread (VT) architecture [68], are similar to ExecD, combining independent execution with deduplication when possible. However, automatically finding commonality in data center workloads and factoring out redundant work is not a focus of architectures like VT. In addition, VT requires software changes, which ExecD does not. Temporal Single Instruction Multiple Thread (T-SIMT) [69] is another example in this class which employs *syncwarp* instructions to bring divergent execution paths back together. We considered a similar solution, but found it to achieve similar results to our hardware synchronization mechanisms, without software changes.

Warp reconvergence [70], [71] is an active area of GPGPU research as warp divergence is bad for both energy and per-

formance. These works aim to reconverge warps as early as possible, making them similar to ExecD. However, GPGPU requires a different programming model, and thus requires software changes, making it less general than ExecD.

There are, of course, many previous works in reducing energy consumption of data center servers. This includes, DVFS [8], [9], PowerNap [3], VM consolidation [72]–[75], energy proportional computing [10], [11], MapReduce energy efficiency [76]–[81], [81], configurable accelerators exploiting Dark Silicon [14] such as Conservation Cores [13], and general energy-aware processor design techniques [12], [15]–[20]. Most of these methods are complimentary to ExecD and do not attempt to deduplicate computation.

VII. Conclusion

Execution Drafting is an architectural method for reducing energy consumption and optimizing $\frac{performance}{energy}$ in data center servers. ExecD synchronizes duplicate instructions from different programs or threads on the same multithreaded core, such that they flow down the pipe consecutively, or draft. ExecD is an instance of a broader class of work in computation deduplication, in which common code is exploited to reduce the work required to execute it.

We evaluated the commonality available in the same applications with different inputs and the same applications with different version numbers, the effectiveness of thread synchronization methods, and the energy-performance trade-off. Results indicate ExecD can achieve up to 20% $\frac{performance}{energy}$ gain over fine-grain multithreading with small area overheads and the potential for larger gains in drafting larger thread counts.

Acknowledgments

This work was partially supported by the NSF under Grant No. CCF-1217553 and DARPA under Grants No. N66001-14-1-4040 and HR0011-13-2-0005. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] R. Buyya, C. S. Yeo, and S. Venugopa, "Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities," in *Proceedings of the IEEE International Conference on High Performance Computing and Communications 2008*.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, Feb 2009.

[3] D. Meisner, B. T. Gold, and T. F. Wenisch, "Powernap: eliminating server idle power," in *Proceedings of the international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS 2009, pp. 205–216.

[4] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the Symposium on Opearting Systems Design & Implementation*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, pp. 10–10.

[5] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 59–72, 2007.

[6] "Amazon elastic compute cloud (amazon ec2)," 2013. [Online]. Available: http://aws.amazon.com/ec2

[7] "Microsoft azure," 2013, http://www.microsoft.com/azure. [Online]. Available: http://www.microsoft.com/azure

[8] S. Herbert and D. Marculescu, "Analysis of dynamic voltage/frequency scaling in chip-multiprocessors," in *Low Power Electronics and Design (ISLPED), 2007 ACM/IEEE International Symposium on*, pp. 38–43.

[9] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott, "Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling," in *High-Performance Computer Architecture, 2002. Proceedings. International Symposium on*. IEEE, pp. 29–40.

[10] X. Fan, W.-D. Weber, and L. A. Barroso, "Power provisioning for a warehouse-sized computer," in *Proceedings of the international symposium on Computer architecture*, ser. ISCA '07, pp. 13–23.

[11] K. T. Malladi, B. C. Lee, F. A. Nothaft, C. Kozyrakis, K. Periyathambi, and M. Horowitz, "Towards energy-proportional datacenter memory with mobile dram," in *Proceedings of the International Symposium on Computer Architecture*, ser. ISCA '12. IEEE, pp. 37–48.

[12] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke, "Composite cores: Pushing heterogeneity into a core," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '12. IEEE, pp. 317–328.

[13] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. Taylor, "Conservation cores: reducing the energy of mature computations," in *Proceedings of Architectural support for programming languages and operating systems*, ser. ASPLOS '10, pp. 205–218.

[14] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the international symposium on Computer architecture*, ser. ISCA '11, pp. 365–376.

[15] J. H. Tseng and K. Asanović, "Ringscalar: A complexity-effective out-of-order superscalar microarchitecture," MIT-CSAIL-TR-2006-066, MIT CSAIL Technical Report, September 2006.

[16] S. Das and W. J. Dally, "Static cache way allocation and prediction," Concurrent VLSI Architectures Group, Stanford University, Tech. Rep. 132, August 2012.

[17] R. C. Harting, V. Parikh, and W. J. Dally, "Energy and performance benefits of active messages," Concurrent VLSI Architectures Group, Stanford University, Tech. Rep. 131, February 2012.

[18] V. Parikh, R. C. Harting, and W. J. Dally, "Configurable memory hierarchies for energy efficiency in a many core processor," Concurrent VLSI Architectures Group, Stanford University, Tech. Rep. 130, February 2012.

[19] E. Witchel, S. Larsen, C. S. Ananian, and K. Asanović, "Direct addressed caches for reduced power consumption," in *Proceedings of the ACM/IEEE international symposium on Microarchitecture*, ser. MICRO '01, pp. 124–133.

[20] K. Asanović, "Energy-exposed instruction set architectures," in *In Progress Session, International Symposium on High Performance Computer Architecture*, January 2000.

[21] D. Meisner and T. F. Wenisch, "Dreamweaver: Architectural support for deep sleep," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2012, pp. 313–324.

[22] G. Long, D. Franklin, S. Biswas, P. Ortiz, J. Oberg, D. Fan, and F. T. Chong, "Minimal multi-threading: Finding and removing redundant instructions in multi-threaded processors," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '10. IEEE Computer Society, pp. 337–348.

[23] M. Dechene, E. Forbes, and E. Rotenberg, "Multithreaded instruction sharing," Department of Electrical and Computer Engineering, North Carolina State University, Tech. Rep., December 2010.

[24] C. Molina, A. González, and J. Tubella, "Dynamic removal of redundant computations," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '99. ACM, pp. 474–481.

[25] A. Sodani and G. S. Sohi, "Dynamic instruction reuse," in *Proceedings of the International Symposium on Computer Architecture*, ser. ISCA '97. ACM, pp. 194–205.

[26] R. Rakvic, J. González, Q. Cai, P. Chaparro, G. Magklis, and A. González, "Energy efficiency via thread fusion and value reuse," *IET Computers & Digital Techniques*, vol. 4, no. 2, pp. 114–125, 2010.

[27] J. C. Mogul, "A trace-based analysis of duplicate suppression in HTTP," Compaq Computer Corporation Western Research Labratory, Tech. Rep. 99/2, November 1999.

[28] N. T. Spring and D. Wetherall, "A protocol-independent technique for eliminating redundant network traffic," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '00. ACM, pp. 87–95.

[29] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," in *Proceedings of the ACM Symposium on Operating Systems Principles*, ser. SOSP '01. ACM, pp. 174–187.

[30] S. Quinlan and S. Dorward, "Venti: A new approach to archival data storage," in *Proceedings of the USENIX Conference on File and Storage Technologies*, ser. FAST '02, Berkeley, CA, USA.

[31] P. Kulkarni, F. Douglis, J. LaVoie, and J. M. Tracey, "Redundancy elimination within large collections of files," in *Proceedings of the USENIX Annual Technical Conference*, ser. ATEC '04, Berkeley, CA, USA, pp. 5–5.

[32] N. Jain, M. Dahlin, and R. Tewari, "Taper: Tiered approach for eliminating redundancy in replica synchronization," in *Proceedings of the USENIX Conference on File and Storage Technologies*, ser. FAST'05, Berkeley, CA, USA, pp. 21–21.

[33] D. R. Bobbarjung, S. Jagannathan, and C. Dubnicki, "Improving duplicate elimination in storage systems," *Trans. Storage*, vol. 2, no. 4, pp. 424–448, Nov. 2006.

[34] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proceedings of the USENIX Conference on File and Storage Technologies*, ser. FAST'08, Berkeley, CA, pp. 18:1–18:14.

[35] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble, "Sparse indexing: Large scale, inline deduplication using sampling and locality," in *Proccedings of the Conference on File and Storage Technologies*, ser. FAST '09. Berkeley, CA, USA: USENIX Association, pp. 111–123.

[36] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein, "The design of a similarity based deduplication system," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. ACM, pp. 6:1–6:14.

[37] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge, "Extreme binning: Scalable, parallel deduplication for chunk-based file backup," in *Modeling, Analysis Simulation of Computer and Telecommunication Systems. IEEE International Symposium on*, ser. MASCOTS '09, pp. 1–9.

[38] B. Debnath, S. Sengupta, and J. Li, "Chunkstash: Speeding up inline storage deduplication using flash memory," in *Proceedings of the USENIX Annual Technical Conference*, ser. USENIXATC'10, Berkeley, CA, USA, pp. 16–16.

[39] F. Guo and P. Efstathopoulos, "Building a high-performance deduplication system," in *Proceedings of the USENIX Annual Technical Conference*, ser. USENIXATC'11, Berkeley, CA, USA, pp. 25–25.

[40] J. Min, D. Yoon, and Y. Won, "Efficient deduplication techniques for modern backup operation," *Computers, IEEE Transactions on*, vol. 60, no. 6, pp. 824–840, 2011.

[41] W. Xia, H. Jiang, D. Feng, and Y. Hua, "Silo: A similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput," in *Proceedings of the USENIX Annual Technical Conference*, ser. USENIXATC'11, Berkeley, CA, USA, pp. 26–28.

[42] P. Shilane, M. Huang, G. Wallace, and W. Hsu, "Wan-optimized replication of backup datasets using stream-informed delta compression," *Trans. Storage*, vol. 8, no. 4, pp. 13:1–13:26, Dec. 2012.

[43] "The apache http server project," 2012. [Online]. Available: http://httpd.apache.org/

[44] "Oracle mysql," 2013. [Online]. Available: http://www.mysql.com/

[45] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in haystack: Facebook's photo storage," in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10, Berkeley, CA, USA, pp. 1–8.

[46] L. A. Barroso, J. Dean, and U. Hölzle, "Web search for a planet: The google cluster architecture," *IEEE Micro*, vol. 23, no. 2, pp. 22–28, Mar. 2003.

[47] "OpenSPARC T1." [Online]. Available: http://www.oracle.com/technetwork/systems/opensparc/opensparc-t1-page-1444609.html

[48] A. Leon, J. Shin, K. Tam, W. Bryg, F. Schumacher, P. Kongetira, D. Weisner, and A. Strong, "A power-efficient high-throughput 32-thread sparc processor," in *International Solid-State Circuits Conference. Digest of Technical Papers*, 2006, pp. 295–304.

[49] A. Leon, B. Langley, and J. Shin, "The ultrasparc t1 processor: Cmt reliability," in *Custom Integrated Circuits Conference*, 2006, pp. 555–562.

[50] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: a 32-way multithreaded sparc processor," *Micro, IEEE*, vol. 25, no. 2, pp. 21–29, 2005.

[51] *OpenSPARC T1 Microarchitecture Specification*, Santa Clara, CA, 2006.

[52] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. ACM, pp. 190–200.

[53] J. Sartori, B. Ahrens, and R. Kumar, "Power balanced pipelines," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pp. 1–12.

[54] S. P. E. Corporation, 2013, http://www.spec.org/.

[55] C. Lee, M. Potkonjak, and W. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communications systems," in *Microarchitecture. Proceedings., IEEE/ACM International Symposium on*, 1997, pp. 330–335.

[56] "Mediabench ii," 2006. [Online]. Available: http://euler.slu.edu/~fritts/mediabench/

[57] "Gcc, the gnu compiler collection," October 2013. [Online]. Available: http://gcc.gnu.org/

[58] "Semantic versioning 2.0.0." [Online]. Available: http://semver.org/

[59] D. Kim, S. S.-w. Liao, P. H. Wang, J. d. Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. P. Shen, "Physical experimentation with prefetching helper threads on intel's hyper-threaded processors," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. IEEE Computer Society, pp. 27–.

[60] D. Kim and D. Yeung, "Design and evaluation of compiler algorithms for pre-execution," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2002. ACM, pp. 159–170.

[61] M. Kamruzzaman, S. Swanson, and D. M. Tullsen, "Inter-core prefetching for multicore processors using migrating helper threads," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '11. ACM, pp. 393–404.

[62] S. S. Liao, P. H. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. P. Shen, "Post-pass binary adaptation for software-based speculative precomputation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 02. ACM, pp. 117–128.

[63] K. Sundaramoorthy, Z. Purser, and E. Rotenburg, "Slipstream processors: Improving both performance and fault tolerance," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2000. ACM, pp. 257–268.

[64] R. Courtland, "The high stakes of low power," *Spectrum, IEEE*, vol. 49, no. 5, pp. 11–12, May 2012.

[65] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook, "Tile64 - processor: A 64-core soc with mesh interconnect," in *Solid-State Circuits Conference. Digest of Technical Papers. IEEE International*, Feb 2008, pp. 88–598.

[66] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele, "Many-core key-value store," in *Green Computing Conference and Workshops (IGCC), International*, July 2011, pp. 1–8.

[67] R. Rakvic, Q. Cai, J. González, G. Magklis, P. Chaparro, and A. González, "Thread-management techniques to maximize efficiency in multicore and simultaneous multithreaded microprocessors," *ACM Trans. Archit. Code Optim.*, vol. 7, no. 2, pp. 9:1–9:25, Oct. 2010.

[68] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanović, "The vector-thread architecture," in *Computer Architecture, 2004. Proceedings. International Symposium on*, pp. 52–63.

[69] Y. Lee, R. Krashinsky, V. Grover, S. Keckler, and K. Asanović, "Convergence and scalarization for data-parallel architectures," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2013, pp. 1–11.

[70] G. Diamos, B. Ashbaugh, S. Maiyuran, A. Kerr, H. Wu, and S. Yalamanchili, "Simd re-convergence at thread frontiers," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44 '11. ACM, pp. 477–488.

[71] S. Collange, "Stack-less SIMT reconvergence at low cost," Tech. Rep. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00622654

[72] K. Ye, D. Huang, X. Jiang, H. Chen, and S. Wu, "Virtual machine based energy-efficient data center architecture for cloud computing: A performance perspective," in *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, 2010, pp. 171–178.

[73] A. Beloglazov and R. Buyya, "Energy efficient resource management in virtualized cloud data centers," in *Proceedings of the IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, ser. CCGRID '10. IEEE Computer Society, pp. 826–831.

[74] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu, "No "power" struggles: Coordinated multi-level power management for the data center," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2008. ACM, pp. 48–59.

[75] A. Beloglazov, J. Abawajy, and R. Buyya, "Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing," *Future Gener. Comput. Syst.*, vol. 28, no. 5, pp. 755–768, May 2012.

[76] Y. Chen, S. Alspaugh, D. Borthakur, and R. Katz, "Energy efficiency for large-scale mapreduce workloads with significant interactive analysis," in *Proceedings of the ACM european conference on Computer Systems*, ser. EuroSys '12, pp. 43–56.

[77] Y. Chen, L. Keys, and R. H. Katz, "Towards energy efficient mapreduce," EECS Department, University of California, Berkeley, Tech. Rep., Aug 2009.

[78] W. Lang and J. M. Patel, "Energy management for mapreduce clusters," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 129–139, Sep. 2010.

[79] T. Wirtz and R. Ge, "Improving mapreduce energy efficiency for computation intensive workloads," in *Green Computing Conference and Workshops (IGCC), 2011 International*, pp. 1–8.

[80] J. Leverich and C. Kozyrakis, "On the energy (in)efficiency of hadoop clusters," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 1, pp. 61–65, Mar. 2010.

[81] Y. Chen, A. S. Ganapathi, A. Fox, R. H. Katz, and D. A. Patterson, "Statistical workloads for energy efficient mapreduce," EECS Department, University of California, Berkeley, Tech. Rep., Jan 2010.