This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2018.2883286, IEEE Transactions on Emerging Topics in Computing

Trading Computation for Communication: A Taxonomy of Data Recomputation Techniques

Ismail Akturk, Ulya R. Karpuzcu

Abstract—A critical challenge for modern system design is meeting the overwhelming performance, storage, and communication bandwidth demand of emerging applications within a tightly bound power budget. As both the time and power, hence the energy, spent in data communication by far exceeds the energy spent in actual data generation (i.e., computation), (re)computing data can easily become cheaper than storing and retrieving (pre)computed data. Therefore, trading computation for communication can improve energy efficiency by minimizing the energy overhead incurred by data storage, retrieval, and communication. This paper provides a taxonomy for the computation vs. communication trade-off accompanied by a quantitative characterization.

Index Terms—data recomputation; communication reduction; energy efficiency; amnesic execution; load value prediction.

1 Introduction

Addressing the energy problem of modern computing [1] is not possible without understanding *where the power goes*. Figure 1 demonstrates a generic template for the sequence of events accompanying each step of classic computing: Upon retrieval of the input operands from the memory hierarchy (① & ②), compute resources (be it general-purpose cores or specialized accelerators) derive the output data from the inputs (③), followed by storage (④ & ⑤) and retention (⑥) of the output data until the next update. Power goes to all of these events. The building blocks of classic processors, digital switches, consume dynamic power as they toggle, and – being far from ideal due to aggressive miniaturization – static power due to leakage when turned off.

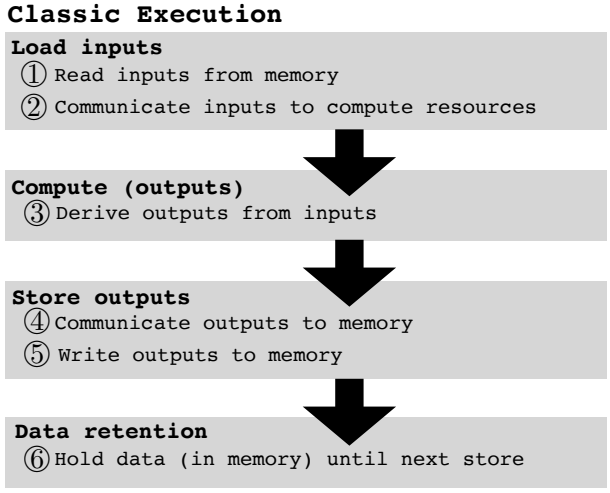

Fig. 1: Classic execution at each step of computation.

Both the breakdown of total power consumption across events and the ratio of dynamic to static power per event evolve as a function of the operating regime and technology. Unfortunately, emerging technology solutions are not mature enough to meet the growing performance, storage capacity, and communication bandwidth demand within the tightly bound power budget (mainly due to cooling and power delivery limitations). At the same time, imbalances between logic and memory technologies cause energy (time $\times$ power) consumption of data loads and stores (①, ②, ④ and ⑤) to significantly exceed the energy consumption of actual computation (③) [1], [2]. As a consequence, reproducing, i.e., *recomputing* data can become more energy efficient than storing and retrieving pre-computed data. This discrepancy is expected to become even more prevalent with technology scaling [3].

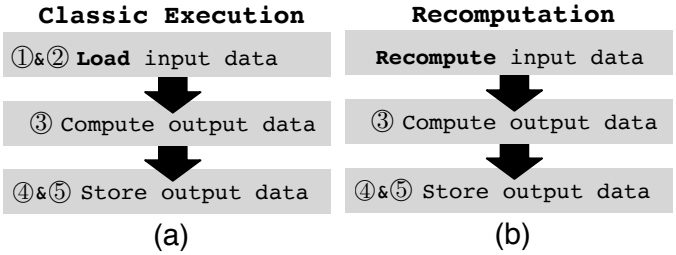

Fig. 2: Classic execution vs. Recomputation

Figure 2(a) summarizes the classic trajectory at each step of execution from Figure 1. Black arrows point to the direction of data flow. Figure 2(b), on the other hand, captures how the picture changes by adapting *recomputation*. The idea is swapping the load from step ① for the reproduction of the actual data values (which would otherwise be loaded from memory). Recomputation can reproduce such data values by brute-force **recalculation** [4][1], value **prediction** [5], [6], or **approximation** [7], [8] – spanning a three-way taxonomy. ① incurs the time and power overhead of the memory access to perform the load; ② , of the subsequent communication of

- *Ismail Akturk is with the Department of Electrical Engineering and Computer Science, University of Missouri, Columbia, MO, 65211. E-mail: akturki@missouri.edu*
- *Ulya R. Karpuzcu is with the Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, MN, 55455. E-mail: ukarpuzc@umn.edu*

Manuscript received February, 2018; revised September, 2018.

1. While Amnesiac [4] refers to *recomputation* as *recalculation*, we use *recomputation* in much broader sense in this paper: to refer to not only *recalculation*, but also *prediction* and *approximation*.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2018.2883286, IEEE Transactions on Emerging Topics in Computing

the respective data values, i.e., inputs to compute resources. Recomputation transforms the overhead of ① & ② to the overhead of the reproduction of the respective data values, which is similar to the overhead of ③. Therefore, recomputation can only improve energy efficiency if the cost of data reproduction remains less than the overhead of ① & ②. In other words, the overhead of ① & ② sets the budget for recomputation. Under recomputation, the workload becomes more compute-intensive to make a better use of classic processors optimized for compute performance, as opposed to energy efficiency.

We will next look closer into the 3-way taxonomy of data recomputation techniques, accompanied by a quantitative compare and contrast. In the following, Section 2 covers the motivation; Section 3 introduces the taxonomy; Sections 4 and 5 provide the evaluation; Section 6 discusses related work, and Section 7 summarizes our findings.

2 Motivation

While emerging technology solutions alter the breakdown of total energy among stages ① – ⑥ per Figure 1, the share of data communication and memory energy (i.e., ①, ②, ④, ⑤, ⑥) is projected to be predominantly higher [2]. At the same time, the inevitable quest for higher degrees of parallelism hurts data locality, therefore, increases communication energy further [1], [2].

2.1 Impact of Operating Regime & Process Technology

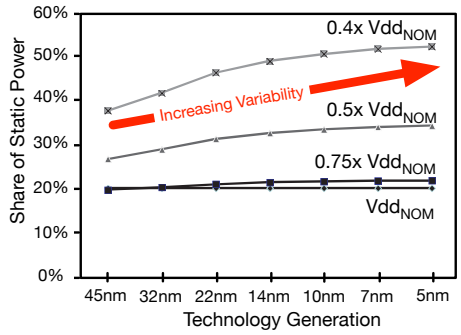

Fig. 3: Evolution of share of static power [9].

A promising way to boost energy efficiency is reducing the operating voltage, Vdd, aggressively to reach the switching threshold [10]. As Vdd decreases, both dynamic and static power reduce, however, not at the same pace: Static power reduces less, and the share of static power grows. Figure 3 depicts how the share of static power (y-axis) evolves with technology scaling (x-axis). Each trend-line corresponds to a different Vdd. For each technology generation, the share of static power is optimized not to exceed 20% under nominal conditions, when operating at nominal Vdd, Vdd_{NOM}. For each technology generation, as Vdd decreases from its nominal value Vdd_{NOM} by $0.75\times$, $0.5\times$, and $0.4\times$, the share of static power quickly increases. For example, at 22nm, as Vdd_{NOM} decreases by $0.4\times$, the share of static power approaches 50%. Aggravated by shrinking feature sizes, variability in design parameters intensifies this effect: Due to variability, for any given Vdd, the share of static power tends to increase over technology generations. On the other hand, the impact of variability increases with decreasing Vdd.

As Vdd decreases, the total power consumption reduces, however, a progressively increasing fraction of this reduced consumption goes to static power. Thus, static-power-heavy stages (mainly ⑥) become relatively more power hungry than dynamic-power-heavy phases (mainly ③). Emerging non-volatile memory technologies such as PCM [11], [12] or STT-RAM [13], [14] can minimize data retention (⑥) power due to practically zero static power, but suffer from excessive write (⑤) energy [15]. Recomputation can still be beneficial in this case, since recomputation can help reduce all components of data communication and memory energy (i.e., ①, ②, ④, ⑤, ⑥), including writes and data retention.

While energy efficiency assumes its maximum in the vicinity of approximately $0.4\times$ or $0.5\times$ Vdd_{NOM} [15], operation at such ultra-low Vdd can easily hurt data locality. This is because only increasing concurrency can prevent performance degradation due to the sizable drop in operating speed (frequency) at low Vdd [10]. As a result, each core tends to spend both more time and power, therefore more energy, in data communication (i.e., ②, ④). Figure 3 does not consider this effect, which we will look closer into in Section 2.2.

3D stacking [16] or emerging photonics based interconnects [17] can render a lower data communication energy when compared to the state of the art, but would not alter the communication-centric nature of parallel processing: Orthogonal to the technology of the communication medium, higher levels of concurrency tend to hurt data locality, hence reduce the mean time to data communication. Accordingly, data communication is expected to remain as one of the most energy-hungry stages.

2.2 Concurrency vs. Data Locality

In a classic processor, cores communicate over the shared memory. Therefore, core-to-core communication translates into a sequence of core-to-memory (④ in Figure 1) and memory-to-core (② in Figure 1) communication. The magnitude and the frequency of data exchange depends on the data distribution among the cores.

With increasing number of cores, the problem can distribute data to cores following *strong* or *weak* [18] scaling. Under both scaling paradigms, n$\times$ more cores increase the throughput performance by n$\times$ in the best case, if we exclude the overhead of communication. Table 1 captures how the total and per core *problem size PS*, *execution time t*, and *throughput performance PS/t* evolve for an n-fold increase in core count.

TABLE 1: Strong vs. weak scaling for an n-fold increase in core count. Best case scenario, excluding communication overhead. PS: problem size.

Scaling	(Total) PS	PS per core	time (t)	PS/t	PS share (per core)
Strong	const.	/n	/n	$\times$n	/n
Weak	$\times$n	const.	const.	$\times$n	/n

Under strong (weak) scaling, overall *PS* remains constant (increases by n$\times$), hence, *PS per core* decreases by n$\times$ (remains constant). *t* is proportional to *PS per core*. As a result, *PS/t* increases by $n\times$. At the same time, *PS share* per core decreases, as tabulated in the last column, which represents the ratio of *PS per core* (column 3) over the total *PS* (column 2).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2018.2883286, IEEE Transactions on Emerging Topics in Computing

PS share serves as a proxy for data locality. This is because the total *PS* governs the total amount of data to be processed. Accordingly, *PS share* reflects the fraction of data processed by each core, which has to reside in close physical proximity to the core. Independent of the scaling paradigm, higher n – higher degrees of parallelism – tends to hurt data locality, hence, increase the likelihood of core-to-core communication. Factoring in the resulting data exchange overhead can easily wipe out the n-fold performance improvement. Indeed, time and power spent in data movement and the orchestration thereof is expected to dominate time and power spent in computation [1], [2].

3 Recomputation Taxonomy

The energy overhead of the load from Figure 2(a) determines the energy budget for recomputation. Unless the energy cost of reproducing data remains less than the energy cost of the respective load, recomputation cannot improve energy efficiency. Therefore, whether recomputation can improve energy efficiency or not tightly depends on where the data reside in the memory hierarchy – it is the location of the data in the memory hierarchy which determines the energy cost of the load. On the other hand, recomputation also incurs an energy cost due to the introduction of recomputing instructions to reproduce the respective data values.

Recomputation can reproduce such data values by brute-force **recalculation** [4], value **prediction** [5], [6], or **approximation** [7], [8], which gives rise to a 3-way taxonomy:

- Under brute-force **recalculation**, the recomputation effort goes to the *derivation of data values*, by re-executing the producer instructions (of the data values, which would otherwise be loaded from memory).
- Under **prediction**, the recomputation effort goes to the *estimation of data values* by exploiting *value locality* – the likelihood of the recurrence of data values [6] within the course of execution.
- Under **approximation**, the recomputation effort goes to the reproduction of the data values, however, *at reduced accuracy*. Generally, these techniques come in two flavors: (i) approximate recalculation, and (ii) approximate prediction. In (i), the recomputation effort goes to the actual *calculation of data values* – as it is the case for brute-force recalculation, however, *at reduced accuracy*. In this case, the compute resources may only partially execute the producer instructions (e.g., by dropping a subset), or perform recomputation at reduced precision. In (ii), the recomputation effort goes to to the *estimation of data values* by exploiting *value locality* – as it is the case for prediction, however, *approximately*.

Be it recalculation or prediction based, depending on the accuracy of approximate reproduction of the data values, **approximation** may degrade the accuracy of the end results. In this study we evaluate recomputation techniques at *iso-accuracy*, moreover, without comprimising accuracy. Hence, our analysis spans (full-accuracy) **recalculation** and **prediction**, and leaves **approximation** based recomputation to future work.

3.1 Recalculation Based Recomputation

Recalculation can be implemented in various ways. In the following, we will use a compiler-based proof-of-concept implementation similar to [4]: During code generation, the compiler replaces each energy-hungry load instruction with the sequence of (arithmetic/logic) producer instructions of the respective data values. To this end, the compiler recursively traces data dependencies. The sequence of producer instructions forms a backward slice, as depicted in Figure 4, which we will refer to as a *Recalculation Slice* or *RSlice*.

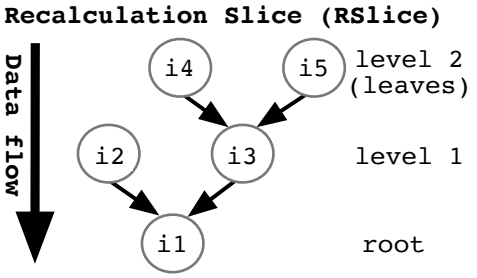

Fig. 4: Example Recalculation Slice (RSlice)

Figure 4 demonstrates an example RSlice. Each RSlice is an upside-down tree, with nodes representing producer instructions to be re-executed. Data flows from the leaves to the root. The node at the root corresponds to the immediate producer of the data value which would otherwise be loaded from memory. Nodes at level 1 correspond to the producers of the root. Nodes at level *l* correspond to the producers of nodes at level *l-1*. The number of incoming arrows at each node reflects the number of producers (of the node) to be re-executed. The leaf nodes either represent terminal instructions which do not have any producers, or instructions for which re-execution of their producers is not energy efficient. In the proof-of-concept implementation, the compiler is in charge of making sure that all input operands of producer instructions within an RSlice are available at the anticipated time of **recalculation**. Unless the compiler guarantees this constraint, an RSlice cannot replace its respective load in the binary.

The compiler swaps a load with its respective RSlice only if **recalculation** of the corresponding data value along the RSlice is more energy efficient than performing the load. To cross-validate the accuracy of the proof-of-concept optimizing compiler pass adapted from [4], we implement an Integer Linear Programming (ILP) based mathematical formulation (provided in the Appendix), which solves recalculation-enabled energy minimization as an optimization problem. We use Basic Block (BB) as the recalculation granularity (instead of an instruction) for the ILP formulation. A BB represents a super-instruction with a bounded number of input and output values. A BB cannot incorporate a branch or jump, by definition. For the ILP formulation, a finer granularity (i.e., instruction) incurs a higher overhead for dependency tracking from the computational complexity point of view. On the other hand, a coarser granularity increases the recalculation cost (due to a higher number of instructions to be reexecuted per eliminated load) and yields a more pessimistic solution. Basic block granularity provides a sweet spot in terms of computational complexity and accuracy for the ILP formulation. This ILP formulation suits itself well to compiler integration, as well.

3.2 Prediction Based Recomputation

Under **prediction**, the recomputation effort goes to the estimation of data values, instead of brute-force **recalculation**. Accurate estimation is only possible if data values (which otherwise would be loaded from memory) exhibit high

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2018.2883286, IEEE Transactions on Emerging Topics in Computing

TABLE 2: Benchmarks deployed

Suite	Benchmark	Input	Description
SPEC	429.mcf (mcf)	test	Combinatorial Optimization
SPEC	482.sphinx3 (sx)	test	Speech Recognition
NAS	is	A	Integer Sorting
PARSEC	canneal (ca)	simsmall	Routing Cost Minimization
PARSEC	facesim (fs)	simsmall	Motion Simulation
PARSEC	ferret (fe)	simsmall	Content Similarity Search
PARSEC	raytrace (rt)	simsmall	Real-time Raytracing
Rodinia	backpropagation (bp)	65536	Pattern Recognition
Rodinia	breath-first search (bfs)	graph1MW_6.txt	Graph Traversal
Rodinia	srad (sr)	100 0.5 502 458 1	Image Processing

value locality – i.e., a high likelihood of recurrence [6] within the course of execution. For example, if a data value exhibits excellent (100%) locality, just storing the value in a dedicated buffer and retrieving it from there may turn out to be more energy efficient than recalculating it (Section 3.1) or loading it from memory. Even if the value locality remains less than 100%, such buffered history of values can be used for **prediction**. Recent work has shown that emerging applications can oftentimes mask prediction incurred inaccuracy due to potential errors in estimation, as implied by the imperfect value locality [6].

Value retrieval from the history buffer constitutes the main cost of **prediction**. Under imperfect value locality, a prediction algorithm can help estimate the respective value by using the buffered history of previously observed values. In this case, the cost of executing the prediction algorithm should also be considered. Approximation aside, classic load value prediction features a repair mechanism to restore data values in case of a misprediction, as well. The overall cost of **prediction** including repair should fit into the recomputation budget, which in turn is set by the energy overhead of the respective load. **Prediction** based recomputation can only be beneficial if its energy cost remains less than the energy overhead of this load.

3.3 Recalculation + Prediction

Prediction based recomputation (Section 3.2) exploits locality of data values which would otherwise be loaded from memory. With respect to **recalculation** (Section 3.1), **prediction** targets the value to be produced by the root node of the RSlice. Input values of RSlice nodes may also exhibit significant value locality. Let us assume that such a node n resides at level l, and it is not a leaf. In this case, predicting n's inputs may turn out to be more energy efficient than re-executing n's producers residing at level l+1 of the RSlice. Hence, combining **recalculation** with **prediction** (i.e., **recalculation + prediction**) can result in pruned RSlices to harvest even more energy efficiency. Recall that, if retrieving input data of leaves requires energy-hungry memory accesses, recalculation along the RSlice cannot be of any use. Each intermediate node of the RSlice subject to **prediction** becomes practically a leaf, as re-execution past such nodes would no longer be necessary.

Recalculation + prediction can prune RSlices, however, even under pure **recalculation** (Section 3.1), RSlices can never grow excessively: the energy overhead of the respective load determines the budget for recomputation. The cost of **recalculation** increases with the number of levels, i.e., *height* of the RSlice, and the number of nodes residing at each level. The re-execution of each node instruction incurs an energy cost. At most, as many nodes can be re-executed (i.e., can reside in the RSlice) as can be fit into the recomputation budget. And **recalculation** can only improve energy efficiency if the cost of re-execution along the RSlice remains less than the recomputation budget, which is set by the energy overhead of the respective load. In this manner, the energy overhead of the load prevents excessive growth of the RSlice. Under **recalculation + prediction**, the cost of re-execution along the RSlice along with the cost of selective **prediction** constitute the cumulative cost of recomputation.

4 Evaluation Setup

We experiment with benchmarks from the SPEC2006 [19], PARSEC [20], NAS [21], and Rodinia [22] suites, which span emerging application domains (Table 2). In the evaluation, we only analyze the benchmarks which harvest sizable energy efficiency gain under recomputation. The analyzed mix contains both compute- and memory-intensive sequential or single-threaded applications. We use Sniper [23] for microarchitectural simulation. The simulated microarchitecture is modeled after an in-order single-core Intel Xeon Phi-like processor without loss of generality, which features an operating frequency of 1.09GHz at 22nm, an L1 instruction cache of 32KB (4-way, LRU), an L1 data cache of 32KB (8-way, LRU, WB), and an L2 cache of 512KB (8-way, LRU, WB).

We profile the native binaries (conforming to classic execution, hence excluding recomputation) of the benchmarks on Sniper: We record (i) value locality of instructions at runtime (to be exploited by **prediction** based recomputation); (ii) cache statistics, i.e., hit and miss rates, at runtime (to derive the probabilistic energy cost model of the compiler pass).

The energy per instruction (EPI) estimates per load, store, and non-memory instructions come from measured Xeon Phi data from [24], which for memory instructions, provides separate EPI estimates for each level *Li* in the memory hierarchy: EPI_{Li}. Using these EPI_{Li} and cache statistics from Sniper, we extract probabilistic EPI estimates for loads as follows: We derive Pr_{Li}, the probability of having the load serviced by level *Li*, using hit and miss statistics of *Li* from Sniper. Then, the sum of $\text{Pr}_{Li} \times \text{EPI}_{Li}$ over all levels i in the memory hierarchy gives the probabilistic energy cost per load. Using this energy cost per load, and the EPIs for non-memory instructions, the compiler pass swaps a load with its respective RSlice only if recalculation of the corresponding data value along the RSlice incurs a lower energy cost than performing the load.

We implement the compiler pass from Section 3.1 in a Pin [25] based tool, which (by using the probabilistic energy cost model detailed above and by tracking data dependencies) swaps load instructions in the binary for the respective RSlices, only if recomputation incurs a lower energy consumption. At the same time, this tool adjusts the binary

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2018.2883286, IEEE Transactions on Emerging Topics in Computing

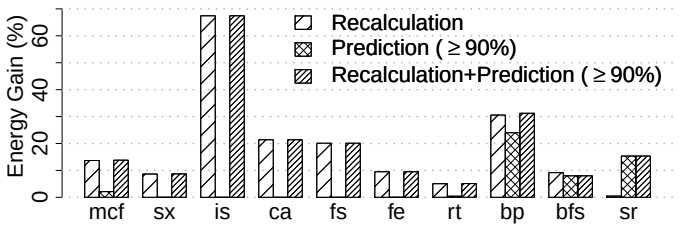

Fig. 5: Energy gain under recomputation.

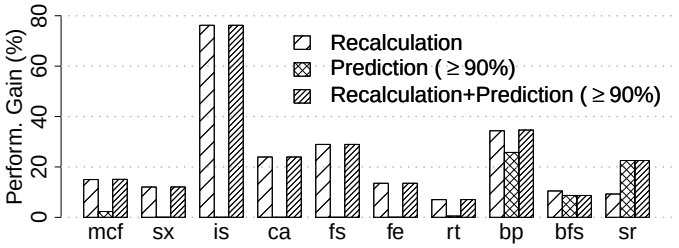

Fig. 6: Performance improvement under recomputation.

under **prediction** and **recalculation+prediction** following Sections 3.2 and 3.3. To identify its maximum potential, we restrict **prediction** with the prediction of the values which would otherwise be loaded (i.e., be produced by RSlice roots under **recalculation**). Under **recalculation+prediction**, on the other hand, prediction can target any RSlice instruction but the root. We deploy Sniper integrated with McPAT [26] to run these annotated binaries in order to collect performance and energy statistics under recomputation.

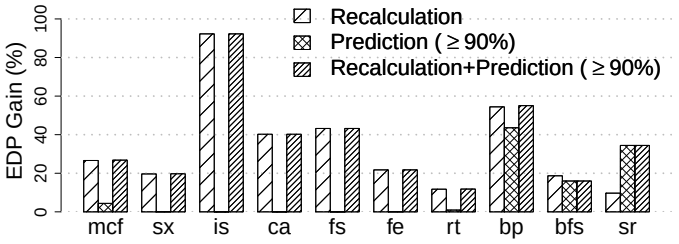

Fig. 7: EDP gain under recomputation.

5 Evaluation

We next quantify the energy efficiency under recomputation and analyze the implications for execution semantics.

5.1 Impact on Energy and Performance

Figure 5 compares the energy consumption under **recalculation**, **prediction**, and **recalculation+prediction** based recomputation. This analysis accounts for the overhead of recomputing producer instructions (along RSlices) under **recalculation** (Section 3.1), and history buffer accesses under **prediction** (Section 3.2). However, we assume that one history buffer access suffices for value prediction at 100% accuracy (i.e., we omit any potential overhead due to prediction algorithms or potential repair). For this experiment, we set the value locality threshold to enable prediction to 90%: prediction only applies to instructions which exhibit at least 90% value locality. **Prediction** targets only the values to be reproduced by *root* instructions of RSlices (all instructions along which are re-executed under **recalculation**). Under **recalculation+prediction**, on the other hand, prediction can target any RSlice instruction but the root (Section 3.3).

Figure 5 reports the energy gain with respect to native execution, which excludes recomputation. We observe that except *bp*, *bfs* and *sr*, the energy gain under **prediction** is insignificant. This is because only a small of number of instructions exhibit a higher value locality than 90%. Due to its wider applicability, **recalculation** unlocks higher energy gains, ranging from 5.06% to 67.43%, except *sr*. The **recalculation** cost for *sr* remains generally higher than the cost of the respective loads. An interesting observation is that *bfs* obtains lower energy gain under **prediction** and **recalculation+prediction** when compared to **recalculation** alone. The reason is that the RSlices of bfs are very short, rendering **recalculation** always cheaper than **prediction**. At the same time, our proof-of-concept implementation gives the priority to prediction if a value exceeds the locality threshold set for prediction (i.e., 90%) under **recalculation+prediction**: in other words, we omit recalculation for all values that exhibit a higher value locality than the threshold (90% in this case), even though recalculation turns out to be less energy hungry than the respective load. Therefore, the energy gain under **recalculation+prediction** cannot exceed the gain under **recalculation** for *bfs*. Overall, the energy gain due to **recalculation+prediction** remains limited for the majority of the benchmarks. The reason is twofold: the benchmarks either do not have enough value locality to exploit prediction (e.g. *mcf*, *sx*, *is*, *ca*, *fs*, *fe*, and *rt*), or recalculation is too costly (e.g. *sr*).

Figure 6 reports the corresponding improvement in performance (i.e., execution time) with respect to native execution. Generally, a similar trend to energy gain applies, except that the performance gain under **recalculation** for *sr* becomes more pronounced when compared to the energy gain.

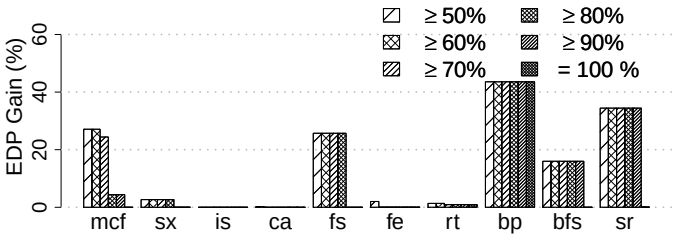

Fig. 8: EDP gain under **prediction** as a function of value locality threshold for prediction.

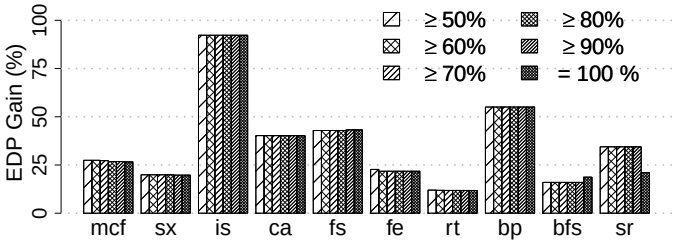

Fig. 9: EDP gain under **recalculation+prediction** as a function of value locality threshold for prediction.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2018.2883286, IEEE Transactions on Emerging Topics in Computing

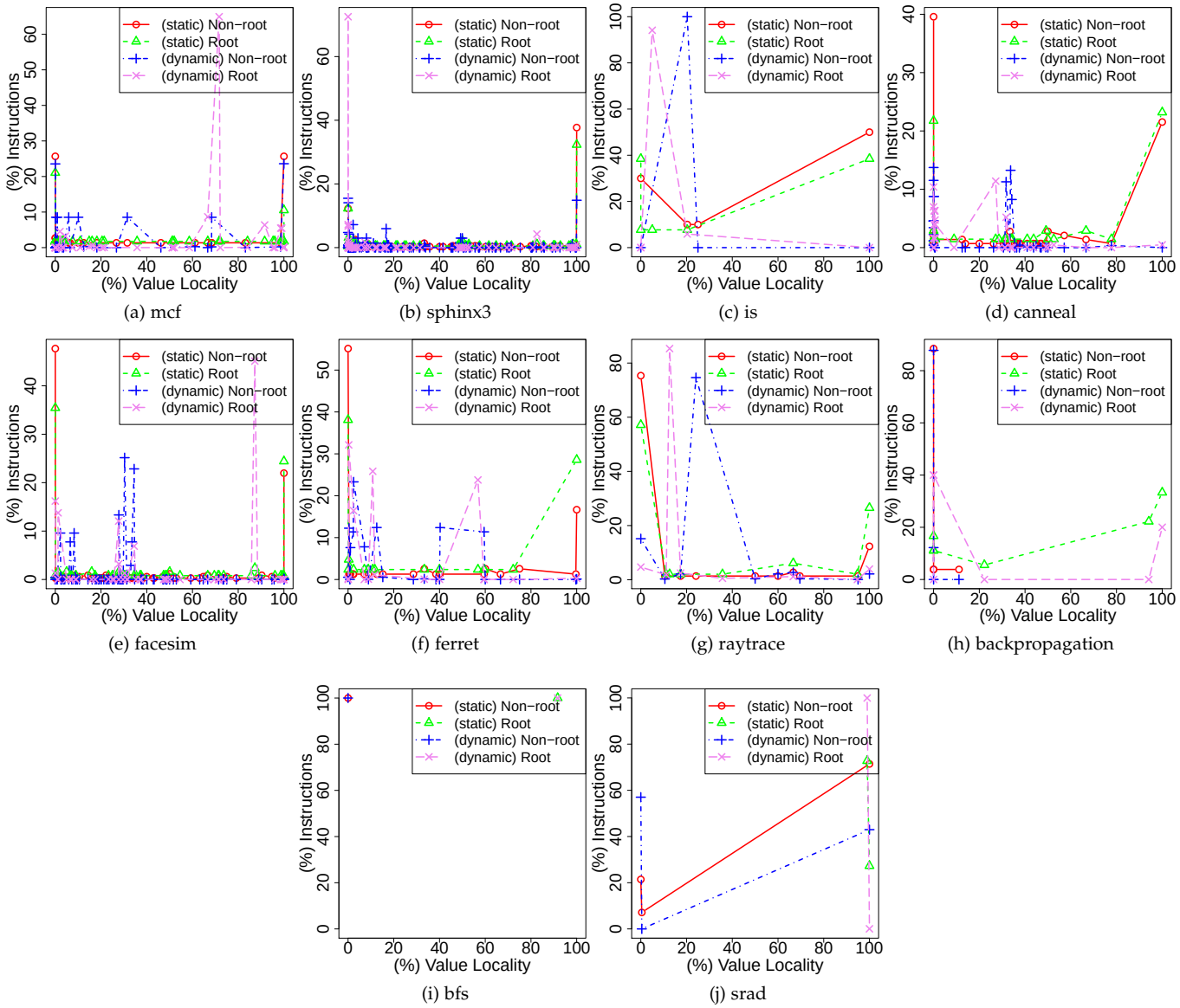

Fig. 10: Value locality of RSlice instructions.

Figure 7 summarizes the resulting gain in energy efficiency in terms of EDP (energy-delay product [27]), with respect to native execution. Overall, **recalculation+prediction** maximizes the EDP gain, and **recalculation** remains effective as well, except *sr* (as explained above). **Prediction** is beneficial for *bp*, *bfs*, and *sr* only – recall that even this gain under **prediction** is optimistic as we neglect any algorithmic or potential repair incurred overhead. Finally, **recalculation+prediction** results in 11.8% to 92.2% EDP gain across all benchmarks.

We next assess the sensitivity of EDP gain to the value locality threshold for prediction. Figure 8 reports the EDP gain under **prediction**; Figure 9, under **recalculation+prediction**, as we sweep the threshold between 50% and 100%. Each bar per benchmark represents a different value locality threshold from this range to enable prediction. Generally, as the threshold increases, the number of values exhibiting at least that much locality reduces – therefore, a lower number of predictions can be performed, and both the energy and performance gains drop accordingly. Among the benchmarks, *bp* exhibits the highest value locality, hence, it benefits most from **prediction**. *bfs* and *sr*, as well, benefit from **prediction** if the threshold remains lower than 100% – as very small number of loads swapped for RSlices feature 100% value locality for these benchmarks. On the other hand, *fs* and *mcf* harvest sizable EDP gain under **prediction** only if the threshold remains lower than 90% and 80%, respectively. The remaining benchmarks have a very small number of load instructions that exhibit $\geq$ 50% value locality, so only a negligible EDP gain applies under **prediction** (which already represents an upper limit for actual gains, as we neglect any algorithmic or repair related overhead). Therefore, **recalculation+prediction** can generally provide higher EDP gains when compared to **prediction**. As mentioned before, *bfs* has small RSlices, thus, the associated recalculation cost usually remains lower than the cost of prediction. Accordingly, *bfs* shows higher EDP gain for 100% threshold (at which a smaller number of values can be predicted, by definition, when compared to lower values of the threshold) under **recalculation+prediction**. Overall, we observe that

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2018.2883286, IEEE Transactions on Emerging Topics in Computing

our findings from Figure 7 generally apply over this wider range of threshold values. We can conclude that *recalculation has wider coverage for recomputation than prediction*. Next, we investigate why this is the case.

5.2 Impact on Execution Semantics

As explained in Sections 3.2 and 3.3, in the context of recomputation, prediction serves two purposes:
(i) to predict the values which would otherwise be loaded from memory (and which correspond to the values to be reproduced by RSlice roots under pure **recalculation**) under **prediction**;
(ii) to predict the input values of intermediate (non-root) RSlice nodes under **recalculation+prediction**.

Prediction can eliminate re-execution along an entire RSlice if the values to be reproduced by the RSlice root (i.e., the values which would otherwise be loaded from memory) exhibit sufficiently high locality. **Recalculation+prediction**, on the other hand, can prune any intermediate RSlice node (except the root) exhibiting sufficient (input) value locality to render a smaller RSlice, which in turn becomes less energy costly to execute.

For prediction based recomputation to work, the respective instructions should exhibit sufficiently high value locality. Figure 10 reports a histogram of % value locality (x-axis) for all instructions residing in RSlices. The y-axis reports the % share of instructions exhibiting a given value of locality on the x-axis. *Root* captures the output value locality of RSlice roots; *Non-root*, the input value locality of intermediate (non-root) RSlice nodes. Recall that the output value locality of RSlice roots corresponds to the value locality of the respective load instructions which are replaced by RSlices.

Notice the distinction between static and dynamic instructions (for both root and non-root, i.e., intermediate instructions). Static instructions are the ones that are embedded in the binary by the compiler. Dynamic instructions are the ones that are actually executed at runtime. A static instruction may have multiple dynamic instances executed at runtime, or may not be executed at all. This distinction helps us to explain why, for instance, we do not obtain much benefit from **prediction** although a great fraction of static instructions have high value locality for *is* (Figure 10c): 38.46% of (static) root instructions of *is* have 100% value locality, but *is* does not benefit much from **prediction** (Figure 8). This is because, at runtime, the root instructions having 100% value locality are not executed as many times as other root instructions that have lower value locality. In fact, less than 1% of dynamic root instructions executed have 100% value locality for *is*, as shown in Figure 10c. The previous section revealed that bp benefits from **prediction** the most (Figure 8). Therefore, we expect a larger fraction of roots to have very high value locality for this benchmark. Figure 10h reveals that 20% of dynamic root instructions of bp have 100% value locality indeed. A similar trend holds for non-root instructions under **recalculation+prediction**. For **recalculation+prediction**, prediction of non-root instructions can provide sizable gains only if the dynamic share of non-root instructions exhibiting high value locality is large.

Figure 11 shows how the node count of RSlices change as the locality threshold to enable prediction increases from 50% to 100% under **recalculation+prediction** – *none* reflects no prediction, i.e., pure **recalculation**. The figure reports a histogram of node count of RSlice (x-axis). The y-axis reports the % share of RSlices having a given node count on the x-axis. A lower threshold enables more predictions, hence more producer instructions can get pruned, and the node count shrinks more. We observe that prediction at a value locality threshold of 50% can reduce the node count of RSlices up to 56%.

6 Related Work

Kandemir et al. proposed recalculation to reduce off-chip memory area in embedded processors [28]. Koc et al. investigated how the recalculation of data residing in memory banks in low-power states can reduce the energy consumption by preventing frequent switching of the corresponding banks to high-power states for data retrieval [29]. Koc et al. further devised recalculation-aware compiler optimizations for scratchpad memories [30]. The compiler strategies from [29] and [30] are confined to array variables. Amnesiac [4], on the other hand, replaces energy-hungry loads with a sequence of low-energy arithmetic/logic instructions to recalculate the respective data values, i.e., values which would otherwise be loaded from memory. The goal is saving energy. Amnesiac is not limited to embedded processors or specific data structures. Therefore we use a similar technique to Amnesiac as a more generic representative for brute-force **recalculation** throughout this paper.

Processing in/near memory (PIM/PNM) [31], [32], [33], [34], [35] can bridge the gap between logic and memory speeds by embedding compute capability in/near memory. Processing in memory can minimize energy-hungry data transfers, as well, and is orthogonal to recomputation. Memoization [14], [36] – the dual of recomputation – replaces (mainly frequent and expensive) computation with table look-ups for pre-computed data. Similar to processing in memory and recomputation, memoization can mitigate the communication overhead (as long as table look-ups remain cheaper than long-distance data retrieval). Memoization and recomputation can complement each other in boosting energy efficiency. Similar to memoization, computation reuse [37], [38], [39], [40] tries to reduce the amount of computation to be performed. The idea of computation reuse is based on the observation that data-intensive applications are run over again and again with either identical or very similar inputs. In such cases, there is a considerable redundant computation (as the input remains almost the same with the previous input), and computation is needed only for the inputs that are changed (which are very limited). Although the main motivation behind computation reuse is to boost performance, it can also improve energy-efficiency as long as the input similarity check and copying the previously computed result remain cheaper than computing the result from scratch. Compared to computation reuse, recomputation is suitable for wider range of applications since it does not rely on input similarity (i.e., recomputation can be used in applications that exhibit no input similarity, as well).

The inefficiency of traditional CPU-centric processing motivated similar a body of work for large-scale data analytics, as well: near-data processing/in-situ analysis strategies that take computation to the storage (rather than data to the processor). These approaches can minimize larger scale energy-hungry data transfers and, being of the same spirit as PIM/PNM solutions, are orthogonal to recomputation. For example, for data sets that cannot fit into main memory, Cho et al. [41] proposed an active SSD architecture which supports basic data processing functions (such as

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2018.2883286, IEEE Transactions on Emerging Topics in Computing

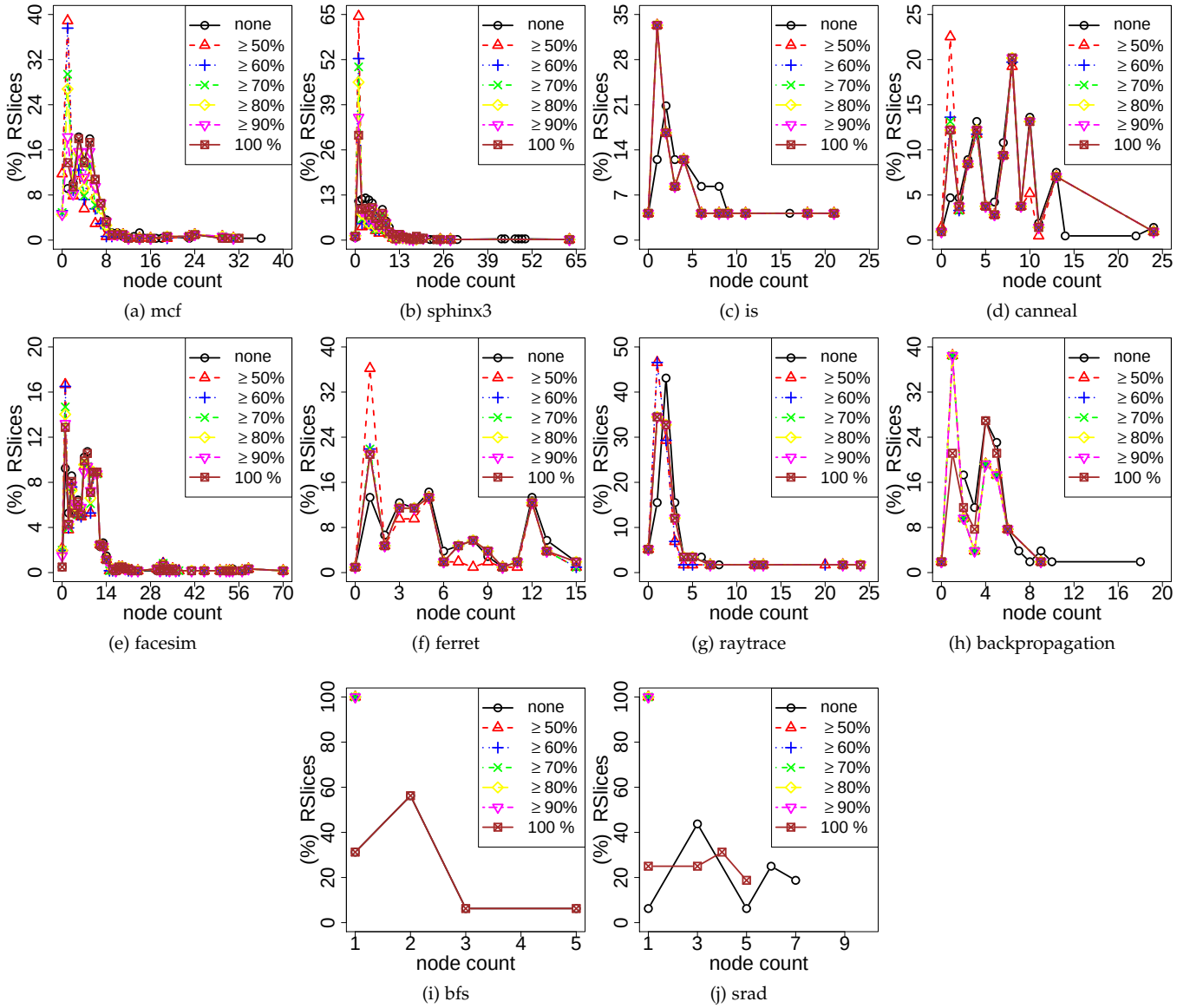

Fig. 11: Node count of RSlices before (**recalculation**) and after pruning (**recalculation+prediction**).

filtering and aggregation) that can be performed directly on flash memory controllers, relying on the high SSD-internal bandwidth and the embedded CPU in the SSD controller to avoid costly data transfers. Tiwari et al., on the other hand, exploited the idle cycles of the SSD controller to process SSD-resident data [42]. Compared to Cho et al. [41], their approach does not require any hardware changes to the SSD controller. Gu et al. took the idea of SSD-based near-data processing further and provided a user-programmable framework that features high-level language support, dynamic load balancing, and multi-core support [43].

7 Conclusion

Recomputation can minimize, if not eliminate, the prevalent power and performance (hence, energy) overhead incurred by data storage, retrieval, and communication, thus, render more energy efficient execution. This paper provided a quantitative proof-of-concept analysis for the computation vs. communication trade-off, along with a taxonomy. Recomputation replaces data load(s) from memory with the reproduction of the respective data. Unless the energy cost of reproducing data remains less than the energy cost of retrieving it from memory, recomputation cannot improve energy efficiency.

In this study, we explored (interactions between) two broad classes of recomputation techniques: brute-force **recalculation** and **prediction**. Under **recalculation**, the recomputation effort goes to the derivation of the data values (which would otherwise be loaded from memory), by re-executing the producer instruction(s) of the eliminated load(s). Under **prediction**, the recomputation effort goes to the estimation of the data values by exploiting value locality – the likelihood of the recurrence of values (which would otherwise be loaded from memory) within the course of execution. We find that **recalculation** has wider coverage for recomputation than **prediction**, mainly because **prediction** cannot be effective under limited value locality as opposed to **recalculation**.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2018.2883286, IEEE Transactions on Emerging Topics in Computing

APPENDIX

Mathematical Formulation

We adopt an integer linear programming (ILP) based formulation for energy minimization under brute-force recalculation. Table 3 lists input and output parameters along with the objective.

Optimizer Inputs	E_{rd}	(average) energy per memory read
	E_{wr}	(average) energy per memory write
	E_{inst}	(average) energy per non-memory instruction
	BB_i	basic block *i* in dynamic control flow
	$\#_{rd,i}$	number of memory read instructions in BB_i
	$\#_{wr,i}$	number of memory write instructions in BB_i
	$\#_{inst,i}$	number of non-memory instructions in BB_i
	$\#_{consumer,i}$	total number of consumers of BB_i
	$\#_{producer,i}$	total number of *immediate* producers of BB_i
	$\#_{BB}$	number of basic blocks in dynamic control flow
Objective	E_i	(minimize) total energy to execute BB_i
Optimizer Outputs	RD_i	$= \begin{cases} 1, & \text{if } BB_i \text{ reads from memory} \\ 0, & \text{otherwise} \end{cases}$
	WR_i	$= \begin{cases} 1, & \text{if } BB_i \text{ writes to memory} \\ 0, & \text{otherwise} \end{cases}$
	C_i	$= \begin{cases} 1, & \text{if } BB_i \text{ is computed} \\ 0, & \text{otherwise} \end{cases}$
	RC_i	$= \begin{cases} 1, & \text{if } BB_i \text{ is recomputed} \\ 0, & \text{otherwise} \end{cases}$

TABLE 3: The lingua franca for ILP-based formulation.

Objective: The energy E_i consumed to execute the basic block (BB) i from the dynamic control flow, BB_i can be formulated as

$$E_i = E_{\text{non-memory instructions}} + E_{\text{writes}} + E_{\text{reads}}$$
$$E_{\text{non-memory instructions}} = C_i \times (E_{inst} \times \#_{inst,i})$$
$$E_{\text{writes}} = WR_i \times (E_{wr} \times \#_{wr,i})$$
$$E_{\text{reads}} = RD_i \times (E_{rd} \times \#_{rd,i})$$

which represents a running sum of products of *energy per instruction*, EPI, and *number of instructions* in the basic block. A different EPI applies for non-memory (e.g. arithmetic), read (load), and write (store) instructions. Indicator variables C, RD, and WR capture whether the BB is computed, reads from the memory, or writes to the memory, respectively. As the optimizer works at BB granularity, RD and WR apply to all reads and writes within the BB. BB_i may be subject to recomputation, possibly multiple times. Under recomputation, E_i evolves to

$$E_i = E_{\text{non-memory instr.}} + E_{\text{writes}} + E_{\text{reads}} + E_{\text{recomputations}}$$
$$E_{\text{non-memory instructions}} = C_i \times (E_{inst} \times \#_{inst,i})$$
$$E_{\text{writes}} = WR_i \times (E_{wr} \times \#_{wr,i})$$
$$E_{\text{reads}} = \sum_{k=1}^{\#_{consumer,i}} RD_{i,k} \times (E_{rd} \times \#_{rd,i})$$
$$E_{\text{recomputations}} = \sum_{k=1}^{\#_{consumer,i}} RC_{i,k} \times (E_{inst} \times \#_{inst,i})$$

When compared to the previous formulation, recomputation not only incurs the additional energy $E_{\text{recomputations}}$, but also changes the read energy E_{reads}. This is because, to be able to recompute BB_i, we need BB_i's inputs. BB_i may need to read inputs from memory, or have them recomputed by its producers. $\#_{consumer,i}$ denotes total number of consumers of BB_i. Theoretically, BB_i can be recomputed as many times as its number of consumers (if each consumer demands input recomputation). Accordingly, decision variables RC_i (which indicates whether a consumer of BB_i demanded BB_i's recomputation) and RD_i (which indicates whether BB_i is reading from memory in preparation to be recomputed per one of its consumer's request) occur $\#_{consumer,i}$ times in the energy calculation. The energy consumption of the entire execution can be derived from the energy cost of component basic blocks from

$$E = \sum_{i=1}^{\#_{BB}} E_i$$

where $\#_{BB}$ is the number of basic blocks in the dynamic control flow of the program, and E_i denotes the energy consumed to execute the (dynamic) basic block BB_i.

Constraints: The optimizer minimizes E(nergy) subject to the following constraints: The first constraint states that each basic block from the dynamic control flow must be computed at least once

$$C_i = 1, \quad for \quad i = 1, 2, ...\#_{BB}$$

The next set of constraints stem from dependencies between dynamic basic blocks. If BB_i is computed (i.e., $C_i = 1$), BB_i may have either read its inputs from memory (i.e., $RD_i = 1$), or demanded recomputation of its inputs by its *immediate* producers (i.e., for each immediate producer BB_p of BB_i, $RC_p = 1$). Accordingly,

$$C_i - (RD_{i,p} + RC_p) = 0$$

applies, where $p = 1, 2, ..., \#_{producer,i}$ points to the p^{th} immediate producer of BB_i. $RD_{i,p}$ indicates that BB_i read data produced by its p^{th} immediate producer from memory. RC_p indicates that BB_i got its input by having its p^{th} immediate producer recomputed. BB_i's immediate producers have immediate producers themselves. These non-immediate producers of BB_i may get transitively recomputed to generate BB_i's inputs, as BB_i's immediate producers are recomputed. The recomputation of BB_i's non-immediate producers gives rise to a further set of constraints in a recursive fashion. For each immediate producer BB_p of BB_i

$$RC_p - (RD_{p,pp} + RC_{pp}) = 0$$

applies, where $pp = 1, 2, ..., \#_{producer,p}$ points to the pp^{th} immediate producer of BB_p. Finally, a basic block should not be recomputed, if all of its consumers read their input data from memory. In other words, the basic block can be recomputed only if at least one of its consumers does not read the input data from memory:

$$RD_{i,p} + RC_p <= 1$$

for all immediate producers BB_p of BB_i.

REFERENCES

[1] M. Horowitz, "Computing's Energy Problem (and what we can do about it)," *Keynote at ISSCC*, 2014.

[2] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, and S. Karp, "Exascale computing study: Technology challenges in achieving exascale systems," *DARPA Information Processing Techniques Office (IPTO) sponsored study*, 2008.

[3] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the Future of Parallel Computing," *IEEE Micro*, vol. 31, no. 5, 2011.

[4] I. Akturk and U. R. Karpuzcu, "AMNESIAC: Amnesic Automatic Computer - Trading Computation for Communication for Energy Efficiency," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[5] J. Huang and D. Lilja, "Exploiting Basic Block Value Locality With Block Reuse," in *International Symposium on High Performance Computer Architecture*, 1999.

[6] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value Locality and Load Value Prediction," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[7] K. J. Lin, S. Natarajan, and J. W. S. Liu, "Imprecise Results: Utilizing Partial Computations in Real-Time Systems," in *Real-Time Systems Symposium*, 1987.

[8] J. S. Miguel, M. Badr, and N. E. Jerger, "Load Value Approximation," in *International Symposium on Microarchitecture*, 2014.

[9] H. Kaul, M. Anders, S. Hsu, A. Agarwal, R. Krishnamurthy, and S. Borkar, "Near-threshold Voltage (NTV) Design: Opportunities and Challenges," in *Design Automation Conference*, 2012.

[10] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge, "Near-Threshold Computing: Reclaiming Moore's Law Through Energy Efficient Integrated Circuits," *Proceedings of the IEEE*, vol. 98, no. 2, 2010.

[11] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *International Symposium on Computer Architecture*, 2009.

[12] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *International Symposium on Computer Architecture*, 2009.

[13] R. Desikan, C. R. Lefurgy, S. W. Keckler, and D. Burger, "On-chip mram as a high-bandwidth, low-latency replacement for dram physical memories," *IBM Austin Center for Advanced Studies Conference*, 2003.

[14] X. Guo, E. Ipek, and T. Soyata, "Resistive computation: avoiding the power wall with low-leakage, STT-MRAM based computing," in *International Symposium on Computer Architecture*, 2010.

[15] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie, "Hybrid cache architecture with disparate memory technologies," in *International Symposium on Computer Architecture*, 2009.

[16] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb, "Die stacking (3d) microarchitecture," in *International Symposium on Microarchitecture*, 2006.

[17] D. Vantrease, R. Schreiber, M. Monchiero, M. McLaren, N. P. Jouppi, M. Fiorentino, A. Davis, N. Binkert, R. G. Beausoleil, and J. H. Ahn, "Corona: System implications of emerging nanophotonic technology," in *International Symposium on Computer Architecture*, 2008.

[18] J. L. Gustafson, "Reevaluating Amdahl's Law," *Communications of the ACM*, vol. 31, no. 5, 1988.

[19] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Computer Architecture News*, vol. 34, no. 4, Sep. 2006.

[20] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," Princeton University, Tech. Rep. TR-811-08, January 2008.

[21] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS parallel benchmarks–summary and preliminary results," in *Conference on High Performance Computing Networking, Storage and Analysis*, 1991, pp. 158–165.

[22] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IEEE International Symposium on Workload Characterization*, 2009.

[23] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *SC*, November 2011.

[24] Y. Shao and D. Brooks, "Energy characterization and instruction-level energy model of Intel's Xeon Phi processor," in *International Symposium on Low-Power Electronics and Design*, September 2013.

[25] 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 *Programming Language Design and Implementation*, 2005.

[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 *International Symposium on Microarchitecture*, 2009.

[27] R. Gonzalez and M. Horowitz, "Energy dissipation in general purpose microprocessors," *JSSC*, vol. 31, no. 9, pp. 1277–1284, 1996.

[28] M. Kandemir, F. Li, G. Chen, G. Chen, and O. Ozturk, "Studying storage-recomputation tradeoffs in memory-constrained embedded processing," in *Design, Automation and Test in Europe*, 2005.

[29] H. Koc, O. Ozturk, M. Kandemir, and E. Ercanli, "Minimizing Energy Consumption of Banked Memories Using Data Recomputation," in *International Symposium on Low-Power Electronics and Design*, 2006.

[30] H. Koc, M. Kandemir, E. Ercanli, and O. Ozturk, "Reducing Off-Chip Memory Access Costs Using Data Recomputation in Embedded Chip Multi-processors," in *Design Automation Conference*, 2007.

[31] H. S. Stone, "A logic-in-memory computer," *IEEE Transactions on Computers*, vol. C-19, no. 1, Jan 1970.

[32] P. M. Kogge, "The EXECUBE approach to massively parallel processing," in *International Conference on Parallel Processing*, 1994.

[33] D. Patterson *et al.*, "A case for intelligent RAM," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, 1997.

[34] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, "Flexram: toward an advanced intelligent memory system," in *International Conference on Computer Design*, 1999.

[35] P. Kogge, S. Bass, J. Brockman, D. Chen, and E. Sha, "Pursuing a petaflop: point designs for 100 TF computers using PIM technologies," in *Frontiers of Massively Parallel Computing*, 1996.

[36] A. Sodani and G. S. Sohi, "Dynamic Instruction Reuse," in *International Symposium on Computer Architecture*, 1997.

[37] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[38] D. Tiwari and Y. Solihin, "MapReuse: Reusing Computation in an In-Memory MapReduce System," in *International Parallel and Distributed Processing Symposium*, May 2014, pp. 61–71.

[39] D. A. Connors and W.-m. W. Hwu, "Compiler-directed Dynamic Computation Reuse: Rationale and Initial Results," in *International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1999, pp. 158–169.

[40] A. Sodani and G. S. Sohi, "Understanding the differences between value prediction and instruction reuse," in *International Symposium on Microarchitecture*, Dec. 1998, pp. 205–215.

[41] S. Cho, C. Park, H. Oh, S. Kim, Y. Yi, and G. R. Ganger, "Active Disk Meets Flash: A Case for Intelligent SSDs," in *International Conference on Supercomputing*. New York, NY, USA: ACM, 2013, pp. 91–102.

[42] D. Tiwari, S. Boboila, S. S. Vazhkudai, Y. Kim, X. Ma, P. J. Desnoyers, and Y. Solihin, "Active Flash: Towards Energy-efficient, In-situ Data Analytics on Extreme-scale Machines," in *USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2013, pp. 119–132.

[43] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang, "Biscuit: A Framework for Near-data Processing of Big Data Workloads," in *International Symposium on Computer Architecture*. Piscataway, NJ, USA: IEEE Press, 2016, pp. 153–165.

Ismail Akturk is an assistant professor of Electrical Engineering and Computer Science at University of Missouri, Columbia. He received his Ph.D. from the University of Minnesota, Twin Cities. His research interests include improving energy efficiency, scalability and fault tolerance of computing systems, emerging and non-conventional computing paradigms.

Ulya R. Karpuzcu is an associate professor of Electrical and Computer Engineering in University of Minnesota, Twin-Cities. She holds an M.S. and Ph.D. in Electrical and Computer Engineering from University of Illinois, Urbana- Champaign. Her research interests span all aspects of energy-efficient computing including application-domain specialized architectures, approximate computing, and physics of computing.