Energy-Efficient Algorithms

Erik D. Demaine *
MIT CSAIL
32 Vassar Street
Cambridge, MA 02139
edemaine@mit.edu

Jayson Lynch*
MIT CSAIL
32 Vassar Street
Cambridge, MA 02139
jaysonl@mit.edu

Geronimo J. Mirano*
MIT CSAIL
32 Vassar Street
Cambridge, MA 02139
geronm@mit.edu

Nirvan Tyagi*
MIT CSAIL
32 Vassar Street
Cambridge, MA 02139
ntyagi@mit.edu

ABSTRACT

We initiate the systematic study of the *energy complexity* of algorithms (in addition to time and space complexity) based on Landauer's Principle in physics, which gives a lower bound on the amount of energy a system must dissipate if it destroys information. We propose energy-aware variations of three standard models of computation: circuit RAM, word RAM, and transdichotomous RAM. On top of these models, we build familiar high-level primitives such as control logic, memory allocation, and garbage collection with zero energy complexity and only constant-factor overheads in space and time complexity, enabling simple expression of energy-efficient algorithms. We analyze several classic algorithms in our models and develop low-energy variations: comparison sort, insertion sort, counting sort, breadth-first search, Bellman-Ford, Floyd-Warshall, matrix all-pairs shortest paths, AVL trees, binary heaps, and dynamic arrays. We explore the time/space/energy trade-off and develop several general techniques for analyzing algorithms and reducing their energy complexity. These results lay a theoretical foundation for a new field of semi-reversible computing and provide a new framework for the investigation of algorithms.

CCS Concepts

•Theory of computation → Models of computation; Design and analysis of algorithms;

Keywords: Reversible Computing; Landauer's Principle; Algorithms; Models of Computation

1. INTRODUCTION

Landauer limit. CPU power efficiency (number of computations per kilowatt hour of energy) has doubled every 1.57 years from 1946 to 2009 [14]. Within the next 15–60 years, however, this trend will hit a fundamental limit in physics, known as Landauer's Principle [19]. This principle states that discarding one bit of information (increasing the entropy of the environment by one bit) requires $kT \ln 2$ energy, where k is Boltzmann's constant and T is ambient temperature, which is about $2.8{\cdot}10^{-21}$ joules or $7.8{\cdot}10^{-28}$ kilowatt hours at room temperature (20°C). (Even at liquid nitrogen temperatures, this requirement goes down by less than a factor of 5.) Physics has proved this principle under a variety of different assumptions [19, 30, 17, 26, 18], and a recent *Nature* paper verified it experimentally [6]. Most CPUs discard many bits of information per clock cycle, as much as one per gate; for example, an AND gate with output 0 or an OR gate with output 1 "forgets" the exact values of its inputs. To see how this relates to Landauer's principle, consider the state-of-the-art 15-core Intel Xeon E7-4890 v2 2.8GHz CPU. In a 4-processor configuration, it achieves $1.2 \cdot 10^{12}$ computations per second at 620 watts, [1] for a ratio of $7.4 \cdot 10^{15}$ computations per kilowatt hour. At the pessimistic extreme, if every one of the $4.3 \cdot 10^{9}$ transistors discards a bit, then the product $3.2 \cdot 10^{25}$ is only three orders of magnitude greater than Landauer limit. If CPUs continue to double in energy efficiency every 1.57 years, this gap will close in less than 18 years. At the more optimistic extreme, if a 64-bit computation discards only 64 bits (to overwrite one register), the gap will close within 59 years. The truth is probably somewhere in between these extremes.

Reversible computing. The only way to circumvent the Landauer limit is to do logically *reversible* computations, whose inputs can be reconstructed from their outputs, using physically *adiabatic* circuits. According to current knowledge, such computations have no classical fundamental limitations on energy consumption. General-purpose CPUs with adiabatic circuits were constructed by Frank and Knight at MIT [10]. The design of reversible computers is still being actively studied, with papers on designs for adders [29], multipliers [28], ALUs [27], clocks [31], and processors [32] being published within the last five years. AMD's CPUs since Oct. 2012 (Piledriver) use "resonant clock mesh technology" (essentially, an adiabatic clock circuit) to reduce overall energy consumption by 24% [8]. Thus the ideas from reversible computing are already creating energy savings today.

But what can be done by reversible computation? Reversible computation is an old idea, with reversible Turing machines being proved universal by Lecerf in 1963 [21] and ten years later by Bennett [3]. Early complexity results showed that any computation can be made reversible, but with a quadratic space overhead [4] or an exponential time overhead [20, 37], in particular models of computation. More recent results give a trade-off with subquadratic space and subexponential time [7]. These general transforms are

*Supported in part by the MIT Energy Initiative and by MADALGO — Center for Massive Data Algorithmics — a Center of the Danish National Research Foundation.

ITCS'16, January 14–16, 2016, Cambridge, MA, USA.

ACM ACM 978-1-4503-4057-1/16/01...$15.00.
DOI: http://dx.doi.org/10.1145/2840728.2840756

[1] We follow Koomey et al.'s [14] definitions, using Cisco's measured SPECint_rate_base2006 of 2,320 to estimate millions of computations per second (MCPS).

too expensive; in particular, in a bounded-space system, consuming extra space to make computations reversible is just delaying the inevitable destruction of bits.

The relationship between thermodynamics and information theory is described by Zurek [40]. In a series of papers, Li, Tromp, and Vitanyi discuss irreversible operations as a useful metric for energy dissipation in computers and study the trade-off between time, space, and irreversible operations. An energy cost based on Kolmogorov complexity [24], a precise but uncomputable measure of the information content of a string, is introduced in [22] and further explored in [5, 36, 23]. These papers study algorithms for Bennett's pebble game as well as simulating Turing machines; however, they still focus on universal results, eshrew RAM models, and analyze problems more from a complexity than an algorithms perspective.

Irreversibility is just one source of energy consumption in current chips, and several other models of computation attempt to capture them individually: switching energy of VLSI circuits [12], dynamic and leakage power loss in CMOS circuits [15, 16], and I/O or memory access [11]. Albers [1] surveys many algorithmic techniques for reducing energy consumption of current computers, including techniques like sleep states and power-down mechanisms, dynamic speed scaling, temperature management, and energy-minimizing scheduling. Ultimately, however, we believe that irreversibility will become a critical energy cost shaping the future of computing, and a topic now ripe for algorithmic analysis.

Our results. This paper is the first to perform a thorough algorithmic study of partially reversible computing, and to analyze realistic time/space/energy trade-offs. We define the (Landauer/irreversibility) energy cost, and use it to explore reversible computing in a novel manner. Although there are many other sources of energy inefficiency in a computer we believe the Landauer energy cost is a fundamental and useful measure. A key perspective shift from most of the reversible computing literature (except [23]) is that we allow algorithms to destroy bits, and measure the number of destroyed bits as the energy cost. This approach enables the unification of classic time/space measures with a new energy measure. In particular, it enables us to require algorithms to properly clean up all additional space by the end of their execution, and data structures to be properly charged for their total space allocation.

We introduce three basic models for analyzing the energy cost of word-level operations, similar to the standard word models used in most algorithms today: the word RAM, the more general transdichotomous RAM, and the realistically grounded circuit RAM. Our models allow arbitrary computation to be performed, but define a spectrum of "irreversibility", from reversible (free) computation to completely destructive (expensive) computation. On top of these basic models (akin to assembly language), we build a high-level pseudocode for easy algorithm specification, by showing how to implement many of the familiar high-level programming structures, as well as some new structures, with zero energy overhead and only constant-factor overheads in time and space:

1. **Control logic:** If/then/else, for/while loops, jumps, function calls, stack-allocated variables.
2. **Memory allocation:** Dynamic allocation and deallocation of fixed-size or variable-size blocks, in particular implementing pointer-machine algorithms.
3. **Garbage collection:** Reference-counting and mark-and-sweep algorithms for finding no-longer-used memory blocks for automatic deallocation.
4. **Logging and unrolling:** Specific to energy-efficient computation, we describe a new programming-language feature that makes it easy to turn energy into space overhead, and later remove that space overhead by playing it backwards.

Primitive	Time (ops)	Space in Log (bits)	Energy (bits)	Thm.
Control Logic				
Paired Jump	$\Theta(1)$	1	0	3.1
Variable Jump	$\Theta(1)$	$1+w$	0	3.1
Protected If	$\Theta(1)$	0	0	3.2
General If	$\Theta(1)$	1	0	3.2
Simple For loop	$\Theta(l)$	0	0	3.3
Protected For loop	$\Theta(l)$	0	0	3.4
General For loop	$\Theta(l)$	$\lg l$	0	3.5
Function call	$\Theta(1)$	0	0	3.6
Memory Management				
Free lists	$\Theta(N)$	$\Theta(wN)$	0	3.7
Reference Counting	$\Theta(N)$	$\Theta(wN)$	0	3.8
Mark & Sweep	$\Theta(N)$	$\Theta(wN)$	0	3.9

Table 1: Summary of our reversible primitives analyses and results including control logic, memory management, and garbage collection. In this table, w is the word size, l is the number of loop iteration, and N represents number of memory objects.

These models open up an entire research field, which we call *energy-efficient algorithms*, to find the minimum energy required to solve a desired computational problem within given time and space bounds. We launch this field with several initial results about classic algorithmic problems, first analyzing the energy cost of existing algorithms, and then modifying or designing new algorithms to reduce the energy cost without significantly increasing the time and space costs. Table 2 summarizes these results.

Although there are many practical papers about minimizing energy in computation (favoring instructions that use somewhat less energy than others), the algorithms community has not made it a standard measure to complement time and space because, without the idea of reversibility, energy is simply within a constant-factor of time. By contrast, in our model, the energy cost can be anywhere between 0 (for reversible computation) and $t \cdot w$ where t is the running time (number of word operations) and w is the number of bits in a word.

Consequences. Reducing the energy consumption of many computations by several orders of magnitude (n) will have tremendous impact on practice. Computer servers alone constitute 23–31 gigawatts of power consumption, which translates to $14–18 billion annually and 1.1–1.5% of worldwide electricity use [13]; there are roughly 50 times as many PCs with an annual growth rate of 12% [38]; and there are about as many smartphones as PCs [9]. Improved energy efficiency would save both environmental impact and money. Reducing energy consumption would also improve the longevity of batteries in portable devices (laptops, phones, watches, etc.), or enable the use of smaller and lighter batteries for similar performance. Perhaps most interestingly, lower energy consumption would lead to faster CPUs, as cooling is the main bottleneck in increasing clock speeds; reducing the energy consumption by a factor of α, we expect to be able to run the CPU roughly α times faster. For example, the world record for CPU clock speed of 8.429 GHz was set by AMD with liquid nitrogen cooling [33].

Our approach is ambitious in that it requires rethinking both software (algorithms) and hardware. Our belief is that building a rich algorithmic theory for (partially) reversible computation, and showing the orders of magnitude in possible energy reduction for important problems, will prove to hardware makers that reversibility is a lucrative feature worth exploring intensely, even before it becomes inevitable by hitting the Landauer Limit.

Algorithm	Time	Space (words)	Energy (bits)	Thm.
Sorting Algorithms				
Comparison Sort	$\Theta(n \lg n)$	$\Theta(n)$	$\Theta(n \lg n)$	6.2
Reversible Comparison Sort	$\Theta(n \lg n)$	$\Theta(n)$	0	6.3, 6.4
Reversible Insertion Sort	$\Theta(n^2)$	$\Theta(n)$	0	6.5
Counting Sort	$\Theta(n+k)$	$\Theta(n+k)$	$\Theta(n+k)$	6.6
Reversible Counting Sort	$\Theta(n+k)$	$\Theta(n+k)$	0	6.8
Graph Algorithms				
Breadth-first Search	$\Theta(V+E)$	$\Theta(V+E)$	$\Theta(wV+E)$	6.9
Reversible BFS[10]	$\Theta(V+E)$	$\Theta(V+E)$	0	6.10
Bellman-Ford	$\Theta(VE)$	$\Theta(V)$	$\Theta(VEw)$	6.12
Reversible Bellman-Ford	$\Theta(VE)$	$\Theta(VE)$	0	6.13
Floyd-Warshall	$\Theta(V^3)$	$\Theta(V^2)$	$\Theta(V^3 w)$	6.14
Reversible Floyd-Warshall [10]	$\Theta(V^3)$	$\Theta(V^3)$	0	6.15
Matrix APSP	$\Theta(V^3 \lg V)$	$\Theta(V^2)$	$\Theta(wV^3 \lg V)$	6.17
Reversible Matrix APSP [10]	$\Theta(V^3 \lg V)$	$\Theta(V^2 \lg V)$	0	6.16
Semi-reversible Matrix APSP	$\Theta(V^3 \lg V)$	$\Theta(V^2)$	$wV^2 \lg V$	6.16
Data Structures				
Standard AVL Trees (build)	$O(n \lg n)$	$O(n)$	$O(w \cdot n \lg n)$	
(search)	$O(\lg n)$	$O(1)$	$O(\lg n)$	5.4
(insert)	$O(\lg n)$	$O(1)$	$O(w \lg n)$	5.5
(k deletes)	$O(k \lg n)$	$O(1)$	$O(w \lg n)$	5.6
Reversible AVL Trees (build)	$O(n \lg n)$	$O(n)$	0	
(search)	$O(\lg n)$	$O(1)$	0	5.7
(insert)	$O(\lg n)$	$O(1)$	0	5.8
(k deletes)	$O(k \lg n)$	$O(k)$	0	5.9
Standard Binary Heap (insert)	$O(\lg n)$	$O(1)$	$O(\lg n)$	5.10
(delete max)	$O(\lg n)$	$O(\lg n)$	$O(w \lg n)$	5.11
Reversible Binary Heap (insert)	$O(\lg n)$	$O(1)$	0	5.10
(delete max)	$O(\lg n)$	$O(\lg n)$	0	5.12
Dynamic Array (build)	$O(n)$	$O(n)$	0	
(query)	$O(1)$	$O(1)$	0	5.3
(add)	$O(1)$	$O(1)$	0	5.3
(delete)	$O(1)$	$O(1)$	0	5.3

Table 2: Summary of our algorithmic analyses and results. In this table, n is the problem size or number of elements in the data-structure, w is the word size, lg is $\log_2$, and in graph algorithms, V is the number of vertices, and E is the number of edges.

Guide. This paper has several sections and does not necessarily need to be read in order or in full, depending on the reader's interest. We recommend reading Sections 2.2 and 2.4 before continuing onto later parts of the paper, to set up the model which is used extensively in the rest of the paper. The remainder of Section 2 further explores our energy models and useful variations. The remaining sections of the paper can be read in any preferred order. Parts of Sections 3–6 use results from previous sections, but these should remain understandable without having seen the prior proofs. Section 3 constructs and analyzes basic control logic and memory management, to enable high-level pseudocode for algorithm specification. Section 4 provides some general techniques we have developed for constructing (semi-)reversible algorithms. Sections 5 and 6 analyze several classic algorithms and data structures, and construct new algorithms and data structures that are more energy efficient. Section 7 poses open problems.

2. ENERGY MODELS

In the following sections we present three different models of computation which define an energy complexity that attempts to capture the energy loss from Landauer's Principle. We begin with a circuit model due to its intuitiveness and similarity to early work done on reversible logic and computation. We then build up RAM models which bear far more similarity to those used for the analysis of algorithms.

2.1 Energy Circuit Model

At the lowest level we will consider logical gates. Every gate is a Boolean function $g : x \to y$. The energy cost of a gate is defined as the log of the size ratio of the input space, X, to the output space, $Y = g(X)$. Thus, energy $E = \lg\left(\frac{X}{Y}\right)$, whose units are bits. The energy cost cannot be negative because a given input cannot map to more than one output. Here we forbid randomized computation. Alternatively, one could allow the creation of b random bits at an energy cost of b. Also, the energy cost is zero exactly when the function is bijective in which case we call the gate *reversible*.

2.2 Energy Word RAM Model

The Energy Word RAM model allows any contiguous segment of memory of size w to be accessed in constant time and defines a fixed set of operations that can take in $O(1)$ word sized inputs in constant time. We also assume memory allocation is handled in a reversible manner. This will become a more reasonable assumption later, when we show linked-lists and stacks can be implemented reversibly. The program and operations have the following restrictions. First, we restrict ourselves to the operations typically found

in high-level languages as well as their reversible analogues. Second, the operation's energy costs should be calculated based off of what can be constructed in the circuit model. Third, all reversible operations must come paired with their inverse operation. Finally, all Energy Word RAM programs must return the machine to its original state, with the exception of a copy of the output living somewhere in memory. This can be done simply but expensively by irreversibly zeroing out every bit and paying the associated energy cost.

The reversible operations we allow include in-place addition and subtraction (e.g., $a \mathrel{+}= b$), increment and decrement (e.g., $a \mathrel{+}= 1$), swapping two variables, testing for equality or less-than relation, copying a variable into an initially empty variable

$$(\textsc{Copy}(a, \underline{b}) \equiv b \mathrel{+}= a)$$

and destroying a known copy of a variable

$$(\textsc{DestroyCopy}(a, b) \equiv b \mathrel{-}= a)$$

We have introduced here a useful notation, that of underlining variables whose values are empty, which shall serve us in writing pseudocode as well. The irreversible operations we allow include overwriting one variable with another, and computing the bitwise AND or OR of two variables.

In this model, we intend that our lowest level pseudocode correspond to an assembly-like language. For simplicity we will continue to work with variables and locations in memory as though they are all stored in RAM, rather than deal with registers, paging, and other complications that may arise depending on the computer architecture. At this level we also explicitly number every line of our program and grant the code access to the program counter, PC, which is the location in memory of the current instruction. At every instruction the PC is incremented, but it can also be manipulated manually, allowing jumps among other operations. It is very easy to make code irreversible by manipulating the PC, as this is implicitly adding control logic to the program. The instruction set we will be using in this paper is the same as the one with which we defined our Word RAM model. For an instruction set for a reversible computer that has been built see Appendix B of Frank's Thesis [10] or [32].

2.3 Energy Transdichotomous RAM Model

The Energy Transdichotomous RAM model is computationally the most powerful and flexible. As with the Word RAM model, we allow access to memory segments of size w in constant time and assume memory allocation is done reversibly. Generally we will assume that $w = \Omega(\lg n)$, making the word size capable of indexing the entire input of the problem. We also allow any operations on $O(1)$ words to be performed in constant time; however, every algorithm can only use a constant number of different operations. The energy cost of an operation is simply the log of the ratio of the input space to the output space, as in the circuit model. Note that this is a lower bound on the energy cost of the operation in the circuit model, and thus a lower bound in the Energy Word RAM model. Finally, we still need to leave the computer in its initial state, except for a copy of the output.

This model is convenient to work in because it is relatively easy to calculate the energy cost of many operations and the flexibility of choosing operations allows us to exploit information in the system without having to work out the details of how it would be implemented. For example, when dividing an integer by four would generally incur two bits of energy loss or two bits of garbage; however, if we happen to know that the number is even, there is really only a single bit of information being lost. Instead of having to worry about how to perform shifts and additions to save this bit, the Transdichotomous RAM model allows us to have a 'divide by four when evenly divisible by 2' operation with the restriction that it only takes even inputs.

We now develop some conventions for writing programs in the Transdichotomous RAM Model. All lines are of the form *TUPLE* $=$ *TUPLE*. Both tuples must contain the same number of elements, and the number of elements must be $O(1)$. The left tuple is a list of all of the values in memory which are used in the computation being performed on this line, including those simply being overwritten. The right tuple contains expressions representing the values that will be in the corresponding variables on the left. These expressions must contain no more than $O(1)$ constant time operations. One interesting convention about this language is every variable implicitly serves two purposes depending on its location. On the left, all variables refer to the memory location where they are stored, and on the right they refer to the values being represented at those memory locations.

As we did above, here we shall annotate variables whose value is known to be zero (often new, unassigned variables) with an underline. This information is often critical to the energy cost of an expression. For example, $(a, b, c) = (a, b, a + b)$ would cost w units of energy because we are erasing every bit in c before replacing it with the value $a + b$. However, $(a, b, \underline{c}) = (a, b, a + b)$ has no energy cost because the input has the value of c assumed to be zero, thus reducing the input space by a factor of 2^w and making the number of inputs and outputs the same.

The following are some examples of common operations written in the format. All operations are assumed to be integer operations with reasonable overflow and rounding conventions. The following examples cost zero energy:

- COPY: $(a, \underline{b}) = (a, a)$
- DESTROYCOPY: $(a, b) = (a, b - a)$
- ADD: $(a, b) = (a + b, b)$
- LESSTHAN: $(a, b, \underline{c}) = (a, b, a < b)$

2.4 High-level Pseudocode

Although the previous section provides a nice, clean way to analyze the energy, space, and time complexity of an algorithm; we may want a more concise and C-like language. Past research on reversible programming languages has focused on fully reversible programming languages and architectures. The first high-level reversible programming languages developed were Janus [25][39] and R [10]. The first reversible architecture, Pendulum, was developed by Vieri [35][34]. Along with Pendulum, Vieri introduced a reversible low-level instruction set, PISA, which is used as a basic reversible instruction set for many future works. Most recently, this architecture has been further improved with the development of Bob [32] using a slightly modified version of PISA known as BobISA, providing more efficient branch handling and address calculation. Axelsen [2] presented the first compilation techniques to translate high-level Janus to low-level PISA, two independently developed reversible languages, and showed that his techniques can be extended for use in any high-level reversible language.

We modeled our pseudocode off of these previous high and low level reversible languages while also adding a few new commands to allow for partial reversibility. We now allow lines of the form *VARIABLE* $=$ *EXPRESSION* as well as for loops, while loops, if/else statements, and subroutine calls. We also introduce log blocks

$x = x + y + z$	**high**
$(x, y, z) = (x + y + z, x, z)$	**intermediate**
101 $\underline{tempx} = x$ 102 x += y 103 y −= x 104 x += z 105 y += *tempx* 106 y += *tempx* 107 *tempx* −= y	**low**

Figure 1: Simple example of code in high, intermediate, and low-level pseudocode.

$\underline{a} = x > y$ *counter* += a	**log**: $\underline{a} = x > y$ *counter* += a **unroll**	$\underline{a} = x > y$ *counter* += a a −= $x > y$ **dealloc**(a)
(a) garbage data not unrolled	(b) logged high-level	(c) logged low-level / automatic unroll

Figure 2: Three examples detailing the mechanics of logged code.

and unroll statements in Section 2.4.1. On lines where we are assigning a variable, we assume that every input in the expression will remain unchanged in its memory location after the computer performs the operation and that the variable on the left-hand side will have its value replaced by the value of the expression. If this is a reversible operation, the variable will merely be changed as appropriate; if it is an irreversible operation, then the variable will be changed and an additional energy cost will be incurred based on the model being used.

Figure 1 gives some simple examples of equivalent code in the three different levels of pseudocode conventions we've developed (high, intermediate, low). The high level is our C-like language. The intermediate language converts high level control logic to jumps and labels. The low level breaks it down further to an assembly-like language. Future sections will use one or more conventions as needed for clarity.

2.4.1 Logging and Unrolling

Dealing with garbage data tends to become tedious when writing reversible computer code. For example, suppose that we were comparing two variables, a and b, and that we wanted to use the result of this comparison to increase some counter; see Figure 2a.

In a normal computer, by the function's end, a would be garbage-collected automatically; however, in our reversible computer a naive garbage collection algorithm would destroy the information stored in a, clearing whatever value it held and costing a word of energy. Thus, the reversible algorithm programmer must handle the task of deallocating a manually.

We call the process of using a series of commands to directly reverse some portion of the code *unrolling*. Manually writing all such commands can be tedious and is prone to error. To expedite the process, we introduce the high-level keywords **log** and **unroll**:

In Figure 2b, the line $\underline{a} = x > y$ is included inside the **log** indentation block, and so is to be reversed at the call to **unroll**. For much longer programs, this extra syntax can save the programmer a great deal of effort that would otherwise be spent writing reverse code. Note that the **log** and **unroll** commands only exist in the highest-level language, and are translated into their manual equivalent at compile-time. The above program, therefore, would compile to the low-level program seen in Figure 2c.

The rules for unrolling are straightforward. Reversible commands can be unrolled simply by including their inverse commands in reverse calling order. Unrolling reversible control logic is discussed in Section 3.1.

To allow our model to unroll semi-reversible programs, which may include irreversible commands, we introduce the *log stack*, a data structure onto which the program can push extra bits of information to be used later to invert the otherwise-irreversible operations. We keep track of our position in the log stack with the log pointer, lp. In the Transdichotomous model, every operation must have its inverse and the process for logging that operation explicitly specified. Furthermore, we assume that this garbage is encoded as efficiently as possible and thus only requires as many bits of space as are needed to distinguish the input space from the output space. Once again, when we log lines with operations that were previously irreversible, we are implicitly defining new operations and should take appropriate precautions. In our Word RAM model, these operations, their inverses, and operations capable of interfacing with lp and memory must be specified.

2.4.2 Promise Notation

We introduce another notational convention that will assist in writing low-energy pseudocode for the Transdichotomous RAM model. At the end of a standard line of code, one may add a comma, the keyword "assert", and then a claimed Boolean expression restricting the values of the involved variables. Some useful examples include:

IsTrue $= 0$, **assert** $0 \leq$ *IsTrue* ≤ 1
$x = x/y$, **assert** $6 \mid x$

Here *IsTrue* may have been the result of a comparison and is known to be either 0 or 1. Thus the energy cost of destroying it is only 1 bit instead of w bits. In the second example, we might know that the problem being computed has some symmetries that a compiler might not see which restrict the values x can take on. Asserts allow us to implicitly define functions which have a restricted input space and thus reduce energy costs. Given the convenience of defining functions in this manner, we must be very careful that we are still using only $O(1)$ different operations in our algorithm.

3. REVERSIBLE PRIMITIVES

In this section, we develop many high-level primitives commonly used throughout algorithms, but which need special care to be done in an energy-efficient manner. Before proceeding, we should discuss in slightly more detail the architecture of our theoretical semi-reversible computer. Our computer only has a single mode of operation, always incrementing the program counter with every instruction. Reversing operations comes from writing the inverse operation in a later section of code, rather than having a separate reversal mode which travels backward along the program counter, inverting those operations. This gives us more flexibility in how to

handle irreversible sections of code, and the manner in which we reverse operations which are not dependent upon each other. However, it comes at the cost that we cannot recover the value of the program counter. Thus this design will have to incur an energy cost of w every time the computer is reset. Because we can run many programs between restarts, we do not consider this to be of major consequence.

3.1 Control Logic

Following Frank [10], we can make branching logic reversible with constant space overhead using paired branching with the destination of a branch being a branch that points back. Thus, we have symmetry, and when running backward, we can just follow the branch we arrived on. However, because we are not working in a fully reversible model, there are some caveats we must pay attention to: all reversible control logic in this section depends on all of the code within the control sequence being reversible. If this is not the case, we can make no guarantees about the correctness when irreversible operations are being performed within some control logic, especially if they are manipulating the variables the control logic depends on.

In Section 2.2 we noted that we can do comparisons reversibly with a single bit of extra space. In this section, we look at jumps, branches, conditionals, for loops, and function calls.

Here we consider the most basic building blocks of control logic, alterations to the program counter in the form of jumps and branches (conditional jumps). Jumps can be performed by a reversible addition to the program counter, we use notation **goto**, **gotoifeq**, and **gotoifneq**. However, if the program counter is allowed to change, we can no longer assume every line was reached by an increment to the program counter, thus creating an irreversible situation. To deal with this, all program counter jumps must be paired with a **comefrom**, **comefromifeq**, or **comefromifneq** statement. In our pseudocode, we allow for **goto** to direct an absoloute or relative jump and note that a compiler can transform absolute to relative, as is used in most reversible architectures.

THEOREM 3.1. *Jumps can be implemented reversibly with constant factor increases in time and space and up to an additive extra word of space per jump.*

PROOF. All jumps must be paired with **comefrom** statements. In the case of a regular **comefrom** statement, the program knows that it reached this location via a jump and thus will jump back in the reverse. However, it is rare that an unconditional jump such as this exist. In the more general case, the program must decide whether the comefrom was reached via a jump or from the line above by an increment in the program counter. To address this, we log two things upon jumping: (1) the length of the jump and (2) a bit indicating that a jump occurred. We then use a **comefromif** to check this bit upon reversing. At the corresponding reverse **comefromif** we'll pop the value off the log stack and use it to either change the program counter or not depending on whether the code jumped to that location.

We can make an additional optimization if a **comefrom** statement only has one corresponding **goto**. Because we know the jump location corresponding with the **comefrom**, this can be implemented reversibly by noting the jump length directly in the source code and not logging. In this case we only have a single bit of storage for logging whether the jump was taken. □

3.1.1 Conditional Statements

We distinguish between two different types of conditionals, a *protected* **if** statement and a general **if** statement. A protected **if** statement is one in which the conditional is not modified within the **if** statement.

THEOREM 3.2. *Protected If statements can be implemented reversibly with constant-factor increases in time and space, and general If statements with an extra bit of overhead in space.*

We now examine a special case of **for** loops. A *simple for loop* is one in which a variable i iterates over the values 1 through k, each time executing some piece of code which does not alter i or k.

THEOREM 3.3. *Simple for loops can be implemented reversibly with constant-factor increases in time and space.*

We consider an extension of the simple for loop. A for loop has *internal conditions* if all variables used in the condition of the for loop only exist within the scope of the for loop. That value is *protected* if it is never changed irreversibly. We define a *protected* for loop as a for loop with protected, internal conditions.

THEOREM 3.4. *A protected for loop can be performed reversibly with constant factor overhead in time and space.*

THEOREM 3.5. *A general for loop can be performed reversibly with constant factor overhead in time and an extra word of space representing the number of loop executions.*

THEOREM 3.6. *Function calls can be implemented reversibly with constant-factor increases in time and space.*

THEOREM 3.7. *Memory allocation using free lists can be done reversibly with constant-factor overheads in time and space.*

Garbage collection often uses a technique known as reference counting. Reference counting keeps track of the number of references to an object or resource and deallocates the space when it is no longer referenced. In our analysis, we do not charge the cost of freeing or destroying the objects to the algorithm. Since this destruction would need to happen regardless of what sort of garbage collection, if any, was performed we believe the energy costs involved are not a fair representation of the work done by the garbage collector itself.

THEOREM 3.8. *Reference Counting can be done reversibly with constant-factor overhead in space and time.*

THEOREM 3.9. *Mark and Sweep can be done reversibly with constant-factor overhead in space and time.*

4. ENERGY REDUCTION TECHNIQUES

This section overviews some general techniques that have been helpful in constructing reversible algorithms and proves some general theorems about algorithms sharing certain properties.

4.1 Complete Logging

One very simple, yet surprisingly useful technique is to simply log every step of an algorithm. This incurs a space cost of $O(t(n))$ words where $t(n)$ is the runtime of the algorithm. Although this seems wasteful, the prevalence of linear time algorithms or linear time sub-routines in algorithms makes this important to remember.

4.2 Reversible Subroutine

Earlier we saw that function calls can be implemented reversibly. We now give a stronger result for being able to use some reversible subroutines efficiently.

THEOREM 4.1. *If we have a fully reversible subroutine whose only effect on the program is through its return value, one need only store the inputs and outputs to this subroutine to later unroll it with only a constant-factor overhead in time.*

PROOF. To do this, we use a slightly more complicated, two-step unrolling process. First, after the subroutine has initially run, we copy out the output and immediately unroll the subroutine. This copy of the output looks no different to the rest of the program from what would normally be computed, and we've already stipulated that the subroutine cannot alter the program through any other means. When it comes time to unroll the subroutine, we may have lost important logged information needed to take us from the output back to the input. At this point, we run the subroutine forward, recovering all of that needed information. Next we delete the copy of the output and unroll the subroutine normally. □

4.3 Data Structure Rebuilding

When attempting to implement data structures which support insert and delete operations reversibly, we run into a new challenge. Often the insertion or deletion operation will create some amount of garbage data which is necessary to reverse it in the future. We also need the result of the operation to remain in place, so we cannot immediately reverse the operation. Thus over the life of the data structure, its size will depend on the total number of insert and delete operations, rather than just the number of elements in the data structure. To circumvent this we can use a technique we call *periodic unwinding*. Note, this technique depends exceedingly on what is considered the data-structure and what is an algorithm that uses the data-structure. This is discussed more below and in Section 5.

THEOREM 4.2. *If a data structure which allows reversible insertions, deletions, and traversals can be constructed reversibly from k insertions in $O(k)$ time and space, and its operations can be performed reversibly with constant-factor overhead in time and space, then it can be maintained reversibly in amortized time with constant-factor overheads in space and time via* periodic unwinding.

PROOF. If there are only $O(n)$ deletions, then we can simply log and unroll all of the operations. If not, we need to keep track of the number of insertions and deletions that have occurred because the last rebuilding. We will keep track of these counts and increment them as part of the insertion or deletion routine. We also track the number of elements in the data structure. Whenever a delete is called, we then check to see whether the number of deletions is more than twice the number of nodes in the data structure. If this is true, we will proceeded to rebuild the data structure. We perform a reversible traversal of the tree, with the addition that we make an extra copy of the inserted data at every node. Now that we have this copy, we can proceed to unroll the data structure, clearing the log. Once this is done we construct a new data structure with the same values as before the reversing, but with none of the accumulated garbage. Construction of the new data structure takes $O(n)$ time. To trigger a rebuilding, we must have called delete a larger number of times than the size of the data structure we are building. We charge the amortized constant cost per element being added to the new data structure to the number of deletes performed, giving us constant amortized time. Our counters and copies of the data all require $O(n)$ space, meaning we never use more than a constant-factor overhead in space. □

With this method, after rebuilding and clearing the log, the data structure can no longer provide any information about past items which were deleted from the data structure. This is covered by the assumption that the algorithm interacting with the data structure makes copies of all of its inputs and if it needs them to reverse itself, it is responsible for maintaining that information. Thus, depending on how the data structure is being used, this technique can be superfluous or very powerful.

5. DATA STRUCTURES

Data structures are meant to be used in the context of an algorithm. In the standard model for algorithms, we can draw a nice abstraction between these two and analyze their properties separately. We also wish to do so in the energy complexity model; however, we need to be more careful about the responsibilities of the data structure and the algorithm for maintaining information and reversibility. First, we assume the data structure is only accessed through the prescribed operations; we don't want the algorithm irreversibly altering stored elements or manually altering the data structure in an unknown way. Second, if it is a reversible data structure, every operation must have a reverse operation. Third, when inserting, the algorithm gives a copy of the data to the data structure and is responsible for maintaining its own reversibility after an insert has been reversed. Fourth, the algorithm will handle zeroing of the bits of elements removed from the data structure. For this purpose, the common delete operation will be replaced with *Extract*(x) which removes an element from a data structure and returns it; however, we will generally still call this operation delete. Fifth, for a reversible algorithm, it is responsible for making the correct calls to reverse functions to reset the data structure.

This is certainly not the only way to treat this interface. We could just as well require the data structure to remember all of the calls performed on it so it could reverse itself upon command. Similarly, we could imagine that a deleted item cannot be handed back to the algorithm, but must in some way be removed by the data structure, most likely when unrolling. We've chosen our conventions because it more closely matches our idea of how subroutines should work and because it is clearer to us how to analyze such cases.

5.1 Stacks and Linked Lists

THEOREM 5.1. *Doubly-linked lists can be implemented reversibly with constant-factor overheads in time and space.*

COROLLARY 5.2. *Stacks, queues, and dequeues can be implemented reversibly with constant-factor overheads in time and space.*

5.2 Dynamic Arrays

THEOREM 5.3. *Dynamic arrays can be implemented reversibly with constant time and space overhead with an extra bit of space for* ADD/DELETE *operations. Size of the structure grows with the number of* ADD/DELETE *operations.*

PROOF. We now consider how to handle these operations reversibly. Because both ADD/DELETE operations work at the end of the array, we check the *length* attribute to find where to perform the reverse operation. For REVERSE-ADD, we remove the element

and decrement *length*. And for REVERSE-DELETE, we must log the deleted element to add it back and increment *length*. We must also consider how to handle table doubling. On an ADD/DELETE operation, table doubling (or halving) occurs based on the result of a single comparison of *length* and *size*. We can log a single bit representing the result of this comparison for each ADD/DELETE operation that will indicate whether a table doubling (or halving) needs to be reversed.

This bit is necessary in order to undo table doubling because we can not determine whether a table doubling operation occurred just by looking at the resulting *length* and *size* attributes. For example, consider a table with length n and size $2n$ where the last operation was an ADD. This state could have been reached in two ways. (1) length $n-1$, size n incurring a table double ; (2) length $n-1$, size $2n$.

We maintain the dynamic array to preserve the order and length of its elements in the reverse direction, thus a REVERSE-QUERY operation can be run in the same way as a QUERY operation by simply making the same query again. Periodic rebuilding of this data structure follows from Theorem 4.2 because all operations are reversible with constant factor overhead and rebuild can be done in linear time. □

5.3 AVL Trees

Using and maintaining standard AVL trees incurs an energy cost proportional to the time of the associated operations.

THEOREM 5.4. SEARCH(x) *can be performed on standard AVL trees in* $\Theta(\lg n)$ *time,* $O(1)$ *auxiliary space and* $O(\lg n)$ *energy.*

THEOREM 5.5. INSERT(x) *can be performed on standard AVL trees in* $\Theta(\lg n)$ *time,* $O(1)$ *auxiliary space and* $O(w \lg n)$ *energy.*

THEOREM 5.6. DELETE(x) *can be performed on standard AVL trees in* $\Theta(\lg n)$ *time,* $O(1)$ *auxiliary space and* $O(w \lg n)$ *energy.*

We will show that, provided only SEARCH and INSERT operations are invoked, reversible AVL trees can be maintained with only constant-factor auxiliary space consumption. If DELETE is to be invoked, then the structure will accumulate an extra $\Theta(k)$ words of space for k DELETE operations invoked over the lifetime of the tree. Such space consumption can still be made reasonable within the context of a larger algorithm, provided that runs of INSERT and DELETE form a small part of the algorithm, and are unwound periodically to refresh log space.

Since these algorithms employ only conditional branches which do not modify their conditions (for example, in SEARCH when comparing a value against a node of the tree to choose a branch, we leave the value of the comparison intact post-search), they are completely reversible with no logging penalty.

THEOREM 5.7. SEARCH(x) *can be performed on reversible AVL trees in* $\Theta(\lg n)$ *time,* $O(1)$ *auxiliary space and 0 energy.*

PROOF. Provided that our reversible AVL tree is constructed using two-way nodes, performing SEARCH(x) reversibly is straightforward. Upon reaching a node v in the tree, we compare its value with x and use the resulting bit to determine whether to jump left or right. After jumping, as we have maintained in memory a pointer back to v, we can compare x to v again to destroy this bit and free the space gain. Once the final node is reached and our answer is found or determined to be absent, we can log our result somewhere and reverse our computation to destroy any remaining garbage bits. This procedure uses constant auxiliary space, and produces our answer reversibly in $O(\lg n)$ time.

□

INSERT is the next operation we address. It includes the task of reversibly rebalancing the tree, a slightly more complicated task than that of SEARCH.

From a given tree, there may be multiple legal trees that underwent a different rotation to produce it. Thus, if we didn't store any auxiliary information about the AVL rotations as we performed them, it would not be possible to immediately reverse a tree's configuration. A key insight into the space consumption of this process is to note that, for a tree containing n unique elements, each of those elements must occupy at least $\Omega(\lg n)$ bits of space each on average (in the word model in particular, each element takes a constant $w = \Theta(\lg n)$ bits of space). Thus, we must store a $\Theta(\lg n)$-sized entry to a rotation log for each inserted element. The space cost of this logging is absorbed into the space cost of the element's value in the tree itself. This is the premise of the following theorem:

THEOREM 5.8. INSERT(x) *for x not yet present in the reversible AVL tree can be performed in* $\Theta(\lg n)$ *time, preserving the* $\Theta(n)$ *space cost of the tree and using 0 energy.*

PROOF. Insertion consists of traversing the reversible AVL tree, adding the new element to the tree, and making any rotations that are necessary to balance the tree.

Traversing the tree can be done reversibly as in SEARCH(x), and we refer to its proof for reversibility. Once we know where x is to be added into the tree, we create a new node for it and proceed to rebalance the tree.

During the balancing step, rotations begin at the lowest level and proceed upward. To perform our operations reversibly, we will keep a log of every rotation performed at each of the $\lg n$ levels of the tree. For each level, we will store 01 for a right-rotation, 10 for a left-rotation, and 00 for no rotation. By keeping this log, we have enough information to go in the reverse direction, proceeding from the top of the tree to its bottom and checking x's value against those of the encountered nodes to progress. In this way, we keep our INSERT(x) action reversible.

Each log entry need only store a number of bits proportional in size to the maximum height of the tree. Because there are n unique entries in the tree at any given time, each call to INSERT incurs only $O(\lg n)$ bits of space cost. Thus our log, which stores $\Theta(\lg n)$ additional bits per element, results in only a constant-factor increase in the space consumption of the tree. This holds even if deletions from the tree have also been performed, as inserting into a tree will always grow the tree's space consumption asymptotically by $\Omega(\lg n)$ bits per insertion, while the log size will grow by $O(\lg n)$ bits per insertion. The space consumption of the tree is thus preserved to within constant factors.

□

THEOREM 5.9. DELETE(x) *can be incorporated into reversible AVL trees, taking* $O(\lg n)$ *time and incurring an additional* $\Theta(k \lg n) = O(kw)$ *bits or* $O(k)$ *words of space for k delete operations.*

5.4 Binary Heaps

THEOREM 5.10. *Binary Heaps can have items inserted irreversibly with* $\Theta(\lg n)$ *time,* $\Theta(1)$ *space, and* $\Theta(\lg n)$ *energy; or reversibly with* $\Theta(\lg n)$ *time,* $\Theta(1)$ *space, and* 0 *energy.*

THEOREM 5.11. *Binary Heaps can have the root node deleted irreversibly with* $\Theta(\lg n)$ *time,* $\Theta(\lg n)$ *space, and* $\Theta(w \lg n)$ *energy.*

THEOREM 5.12. *Binary Heaps can have the root node deleted reversibly with* $\Theta(\lg n)$ *time,* $\Theta(\lg n)$ *space, and* 0 *energy.*

As with AVL trees, binary heaps subject to k insertions and deletions will accumulate an extra $O(k)$ space to be maintained. In some cases this can be resolved by periodic unwinding.

6. ALGORITHMS

This section includes the analysis for the time, space, and energy complexity of several standard algorithms in our model. We also give a number of improved algorithms. Some of our results for algorithms with zero energy complexity are similar to results claimed or proved in [10] about reversible algorithms. However, we prove these results within our own model, which differers slightly from [10].

6.1 Sorting

Sorting is among most fundamental and well understood algorithmic problems. In this section we give reversible algorithms for comparison and counting sorts which match the time and space complexities of know irreversible algorithms. It is especially interesting to see this is achievable despite the known entropy change during comparison sorts which give us a lower bound on their time complexity.

6.1.1 Comparison Sort

THEOREM 6.1. *A comparison sort destroying its input must consume* $\Omega(\lg n!)$ *energy.*

We can achieve this energy bound with Merge Sort. Merge Sort takes in an array of numbers, recursively calls itself on half of the array until it reaches the sorted array of size 1. It then merges the returned arrays by iteratively comparing the smallest values in each array and moving it to the beginning of a new sorted array.

THEOREM 6.2. *Comparison sort destroying its input can be done in* $\Theta(n \lg n)$ *time,* $\Theta(n)$ *space, and* $\Theta(n \lg n)$ *energy.*

If we do not destroy the input and are careful with our algorithm, we can do better:

THEOREM 6.3. *Comparison sort, not destroying its input, if performed on an array of* n *w-bit elements with* $\lg n = O(w)$*, can be done reversibly in* $\Theta(n \lg n)$ *time,* $\Theta(n)$ *auxiliary space, and 0 energy.*

PROOF. This algorithm is a modification of Merge Sort. In summary, we will we augment each element of the array with its index in the array, and so-equipped shall reversibly merge sort the elements in $\Theta(n \lg n)$ time. After we remove these indices from the sorted array, the output of the algorithm will be the original array L and a sorted copy of the array L_{sorted}.

During a traditional Merge Sort, there are three main steps: dividing the array in two, recursing on each half of the array, and merging the two resultant arrays into a complete, sorted array.

The first step, dividing an array in two, is reversed trivially: Given the two resultant lists of such an operation $L[r : s]$ and $L[s + 1 : t]$, the original subarray $L[r : t]$ is their concatenation.

The second step, recursing, will be reversible if our entire algorithm is reversible. We know the base case of sorting a size-1 array is reversible, and thus if steps 1 and 3 of our algorithm are also reversible, then this recursive step will be as well.

The third step, in contrast to the first two, presents us with some difficulties. Given a resultant, fully-merged subarray $L_{sorted}[r : t]$, it is not at all obvious how to go backwards, i.e. how to reproduce the input subarrays $L_{sorted}[r : s]$ and $L_{sorted}[s + 1 : t]$. Information will be lost in this merge step, and to allow our algorithm to be done reversibly, we must find some way to preserve it.

Our augmentation makes this step possible. Before sorting, we transform L into a new array L' of twice the size, which consists of the elements of L each augmented with their index in L. Thus, the elements of our array are 2-tuples (v, i) of each element's value and original location in L. The above transformation is sufficient to make the merge step reversible. To see why, consider any step in the algorithm in which we are trying to merge two sorted subarrays $L[r : s]$ and $L[s + 1 : t]$. Denote the merge subroutine that we are trying to compute as $M_{s+\frac{1}{2}}(L[r : s], L[s + 1 : t])$; that is, we are merging around a pivot $s + \frac{1}{2}$, determined by which step of the algorithm we are presently carrying out. All elements (v, i) with $i < s + \frac{1}{2}$ must have come from $L[r : s]$, and elements with $i > s + \frac{1}{2}$ from $L[s + 1 : t]$. Given a resultant array $L_{sorted}[r : t] = M_{s+\frac{1}{2}}(L_{sorted}[r : s], L_{sorted}[s + 1 : t])$, we can reverse the merge operation step-by-step simply by checking each element's index against the pivot $s + \frac{1}{2}$ to determine where it came from. This enables us to construct two-way branches that perform the merge in a way that is instantaneously reversible. Because the pivot is fixed for each step in the algorithm, no information is lost in computing and decomputing it, and thus this step of the algorithm may be implemented reversibly with only constant additional auxiliary space.

The output of the above algorithm is a list L'_{sorted} of (v, i) tuples sorted in the v keys (whereas our original list L' was "sorted" in the i keys).

Now we need only remove the auxiliary indices from the elements (v, i) to produce our unaugmented sorted list L_{sorted}. This step must be handled with care to ensure that every step is reversible. We begin by reproducing the original array L via a single pass over L'_{sorted}, a simple operation that has not yet destroyed any data. Next, we copy out L_{sorted}, the final, sorted array that we care about. What remains is to dismantle L'_{sorted}, and here we shall employ a special trick: We will perform a single pass over the *original* array L, and for every value v encountered we will perform a logged binary search for this element in the *unaugmented* sorted list L_{sorted}. When the element is found, we will know the value v, index i, and the location of the element (v, i) in the augmented sorted array L'_{sorted}. This is sufficient to destroy this element, setting its entry to zero before unrolling the log of our binary search. The complete dismantling operation uses only $\Theta(\lg n)$ additional logging space total, and only takes time $\Theta(n \lg n)$, so our runtime and space consumption are preserved.

This algorithm is instantaneously reversible at every step, and could be implemented using only simple for loops and two-way conditional branches. Thus, the algorithm is completely reversible under our model. Given an array of size $n = O(2^w)$ which occupies nw space in memory, we can reversibly comparison-sort the array using $\Theta(nw)$ bits of auxiliary space in $\Theta(n \lg n)$ time, matching the best irreversible algorithm to within constant-factors of space and time. □

THEOREM 6.4. *Comparison sort, not destroying its input, can be done reversibly on an array of* n *d-bit elements which require* nd *space in* $\Theta(n \lg n)$ *time,* $\Theta(nd)$ *auxiliary space, and 0 energy.*

PROOF. As we saw in the preceding theorem, reversible comparison sort is straightforward to perform if we first augment each element in the array with a number corresponding to its index in the original list. When the size of the values d is $\Omega(\lg n)$, then we

attain the optimum space bound as the $\lg n$-sized indices get absorbed into the space cost of the d-bit elements. However, we are faced with a conundrum when d is $o(\lg n)$.

To handle this case, we shall employ counting sort in order to reduce the problem of sorting L to the problem of sorting the *unique keys* of L. We utilize a reversible AVL tree (described in Section 5.3) to achieve this.

This algorithm works by reducing the array L only to its unique elements, to sort those elements, and finally to perform a Counting Sort of the original array, consulting our sorted elements to determine the final order. Let k be the number of distinct elements of L. We employ a reversible AVL tree, with actions carefully specified so as to keep them reversible. First, we read the distinct elements of L into the tree, bringing it to a size $O(kd)$, and keeping an $O(n)$-bit uniqueness log and an $O(nd)$-bit rotation log (see reversible AVL trees discussion) as we go. This step takes $O(n \lg n)$ time. Next, we apply Counting Sort on the original array (see Section 6.1.2), consulting the static tree in $O(\lg n)$ time for each element and achieving the $O(n \lg n)$ runtime in this step as well. The output array may be copied and the entire algorithm reversed (Note: not an unrolling of a log, but rather an execution of a reversed version of the algorithm) to leave us with our desired arrays L and L_{sorted}.

This algorithm works by reducing the array L only to its unique elements, to sort those elements, and finally to perform a Counting Sort of the original array, consulting our sorted elements to determine the final order. Let k be the number of distinct elements of L. We employ a reversible AVL tree, with actions carefully specified so as to keep them reversible. First, we read the distinct elements of L into the tree, bringing it to a size $O(kd)$, and keeping an $O(n)$-bit uniqueness log and an $O(nd)$-bit rotation log (see reversible AVL trees discussion) as we go. This step takes $O(n \lg n)$ time. Next, we apply Counting Sort on the original array (see Section 6.1.2), consulting the static tree in $O(\lg n)$ time for each element and achieving the $O(n \lg n)$ runtime in this step as well. The output array may be copied and the entire algorithm reversed (Note: not an unrolling of a log, but rather an execution of a reversed version of the algorithm) to leave us with our desired arrays L and L_{sorted}.

In terms of the intricate details glossed over above, the most involved are in the first step: reversibly constructing a reversible AVL tree out of the unique elements of L. We proceed as follows: making a single pass over our array L, we add every element into the tree. If an element is the first of its exact value to be encountered, we store a corresponding uniqueness bit as true and add the element to the tree. If an element's value already exists in our tree, we store its uniqueness bit as false and move on (never adding duplicates to the tree). These n uniqueness bits allow us to reverse the algorithm, as we know for which elements we modified the tree and on which ones we did not. By theorem 5.8, these insertions take $\Theta(nd)$ space and $O(n \lg n)$ time.

Once the AVL tree is constructed, the rest of the algorithm is a straightforward Counting Sort with a slower $\Theta(\lg k)$ lookup time, yielding the $O(n \lg k)$ time which is optimal for Comparison-based Sort. In addition to the fully-reversible AVL tree data structures, our algorithm employs only simple for-loop passes over the input and reversible two-way branching in the AVL tree, ensuring its reversibility. The entire algorithm takes $\Theta(nd)$ space and $O(n \lg n)$ time, matching the best irreversible comparison sorts up to constant-factors of space and time. □

THEOREM 6.5. *Reversible Duplicated Insertion Sort runs in $\Theta(n^2)$ time, $\Theta(n)$ space, and 0 energy.*

6.1.2 Counting Sort

Counting sort involves counting the number of elements at or below a specific value, and then running through them and adding them to an array based on how many elements are below them. This achieves $\Theta(n + k)$ time and space where k is the size of the maximum integer to be sorted.

THEOREM 6.6. *Counting Sort can be done in $\Theta(n + k)$ time, $\Theta(n + k)$ space, and $\Theta(wn + \lg k)$ energy.*

THEOREM 6.7. *If all entries are unique, then Counting Sort has an energy complexity of $\Theta(\lg n + \lg k)$ energy.*

THEOREM 6.8. *Reversible Counting Sort can be done in $\Theta(n + k)$ time, $\Theta(n + k)$ space, and 0 energy.*

6.2 Graph Algorithms

Frank [10] argues that Breadth-first Search and Depth-first search can be done reversibly. We reproduce this result in our model and give a different analysis.

THEOREM 6.9. *Breadth-first Search runs in $\Theta(V + E)$ time, $\Theta(V)$ space, and $\Theta(wV + E)$ energy.*

THEOREM 6.10. *Reversible Breadth-first Search can runs in $\Theta(V)$ time, $\Theta(V + E)$ space, and 0 energy.*

COROLLARY 6.11. *Reversible Depth First Search can runs in $\Theta(V + E)$ time, $\Theta(V)$ space, and 0 energy.*

6.3 Bellman-Ford

THEOREM 6.12. *Bellman-Ford runs in $\Theta(VE)$ time, $\Theta(V + E)$ space, and $\Theta(VEw)$ energy.*

THEOREM 6.13. *Reversible Bellman-Ford runs in $\Theta(VE)$ time, $\Theta(VE)$ space, and 0 energy.*

6.4 Floyd-Warshall

Frank [10] argues that the Floyd-Warshall algorithm can be adapted to run reversibly with $\Theta(V^3)$ space. This is a substantial increase in space to make the program reversible and thus save energy.

THEOREM 6.14. *Floyd-Warshall runs in $\Theta(V^3)$ time, $\Theta(V^2)$ space, and $\Theta(V^3 w)$ energy.*

THEOREM 6.15. *Reversible Floyd-Warshall runs in $\Theta(V^3)$ time, $O(V^3)$ space, and 0 energy.*

6.5 All Pairs Shortest Path via $(\min, +)$ Matrix Multiplication

Another algorithm for solving APSP involves using the adjacency matrix representation of a graph A and noticing that the relaxations over the edges can be expressed by calculating a new matrix, C, whose entries are given by $c_{ij} = \min_k(a_{ik} + a_{kj})$. Further, this operation is associative, so we can speed up the calculation by using repeated squaring. Thus we have $O(\lg V)$ iterations over (V^2) elements which take $O(V)$ time to compute. Frank [10] claims without proof that this leads to a reversible algorithm that runs in $\Theta(V^3 \lg V)$ time and $\Theta(V^2 \lg V)$ space. We give a proof of this result.

THEOREM 6.16. *Reversible APSP using repeated squaring with (min,+) matrix multiplication runs in $O(V^3 \lg V)$ time, $O(V^2 \lg V)$ space, and 0 energy.*

THEOREM 6.17. *APSP using repeated squaring with (min,+) matrix multiplication runs in $O(V^3 \lg V)$ time, $O(V^2)$ space, and $O(wV^3 \lg V)$ energy.*

We now present a new variation on APSP which demonstrates a non-trivial trade-off between energy and space. By exploiting reversible subroutines, we're able to reach the APSP with repeated squaring bounds on time and space, but beat it in energy cost. The reversible, semi-reversible, and standard APSP using repeated squaring demonstrate there are semi-reversible algorithms that actually achieve bounds not reached by the fully reversible or fully irreversible counterparts.

THEOREM 6.18. *Semi-reversible APSP using repeated squaring with (min,+) matrix multiplication runs in $O(V^3 \lg V)$ time, $O(V^2)$ space, and $O(wV^2 \lg V)$ energy.*

PROOF. To begin, we will examine how each individual entry in the matrix is updated. Say we have a graph represented by adjacency matrix $W = (w_{i,j})$, and a matrix $L^{(m)} = (l_{i,j}^{(m)})$ representing the shortest paths between two vertices with path length at most m. Each entry is updated as

$$l_{i,j}^{(m+1)} = \min_{1 \le k \le |V|} \left(l_{i,j}^{(m)} + w_{k,j} \right)$$

This subroutine runs in $O(V)$ time and $O(wV)$ energy and can thus be trivially made reversible by logging everything, using $O(V)$ time and space. We replace our normal update function with the new reversible one, and by Theorem 3.6 we have a new, more energy efficient algorithm. The subroutine does not use asymptotically more time than before, the temporary use of $O(V)$ space is much smaller than that needed to store the matrices and is freed upon completion of the subroutine, and the energy cost drops by a factor of V which reduces the algorithms total energy cost by a factor of V. □

7. FUTURE DIRECTIONS

This paper built up a framework for designing and analyzing the energy cost of algorithms caused by irreversibility, and started the quest for positive results for basic algorithms and data structures. In many cases, we obtained fully reversible versions of algorithms, but other problems seem more resistant. For example, is there a reversible all-pairs shortest path algorithm with only constant factor overheads in time and space? We managed to give a reduced-energy semi-reversible algorithm for the problem, but a fully reversible algorithm still seems elusive. Shortest-path algorithms more generally seem like a category that are difficult to make reversible, as they use very little space and make frequent use of rewriting old values. We anticipate other graph problems such as max-flow/min-cut may also be challenging and interesting for similar reasons.

There are more fundamental algorithms that should be given high priority given their use in many other results: hashing, predecessor data structures (e.g., van Emde Boas trees), max-flow/min-cut, Fast Fourier Transforms, and dynamic programming. Geometric algorithms offer more nontrivial challenges to attain reversibility, such as line intersection, orthogonal range finding, convex hull, and Delaunay triangulations. We also see the field of machine learning being an interesting target for analysis in the semi-reversible model: these algorithms often have significantly higher time complexities than space complexities, fundamental updates (such as Bayes' rule) which appear reversible, and many conditional updates or data overwrites.

One important question for any practical application is how to deal with long-running programs. Although we are perfectly happy to log some auxiliary information during the execution of a specific program, it may be more problematic to maintain reversibility for the entire operating system of a computer or a long-lived database. This is an area we believe ideas like semi-reversibility and periodic rebuilding will become particularly important.

There are some areas where we see slight extensions of the model opening up interesting questions. First, incorporating randomness seems a practical necessity and carries interesting thermodynamic implications depending on how it is modeled. Assuming there is an energy cost associated with the production of randomness (say, equal to the number of random bits), this may give further reason to investigate exactly how much randomness is needed for an algorithm's correctness. Streaming algorithms and other models where the working space is much smaller than the problem input seem like a rich source of problems. Because we now use sublinear space, our trivial transform is no longer applicable. Further, the larger the gap in space and time, the less ability we have to accrue garbage. Finally, succinct data structures, which try to minimize the bits of space used up to sublinear factors, seem like another challenge: many of our transforms double or triple the space being used by an algorithm, while in the succinct setting, this overhead must be considered.

Finally, a major open direction is to obtain lower bounds. The additional constraints on semi-reversible algorithm design might allow showing algorithms cannot be obtained without some minimum time-space-energy trade-off.

Acknowledgments

We thank Martin Demaine and Kevin Kelley for helpful early discussions about this project, in particular early formulations of the models of computation. We also thank Maria L. Messick and Licheng Rao for useful discussion and help clarifying our pseudocode and model.

References

[1] S. Albers. Energy-efficient algorithms. *Commun. ACM*, 53(5):86–96, May 2010.

[2] H. B. Axelsen. Clean translation of an imperative reversible programming language. In *Compiler Construction*, pages 144–163. Springer, 2011.

[3] C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973.

[4] C. H. Bennett. Time/space trade-offs for reversible computation. *SIAM Journal on Computing*, 18(4):766–776, Aug. 1989.

[5] C. H. Bennett, P. Gács, M. Li, P. M. Vitányi, and W. H. Zurek. Information distance. *IEEE Transactions on Information Theory*, 44(4):1407–1423, 1998.

[6] A. Berut, A. Arakelyan, A. Petrosyan, S. Ciliberto, R. Dillenschneider, and E. Lutz. Experimental verification of Landauer's principle linking information and thermodynamics. *Nature*, 483(7388):187–189, 2012.

[7] H. Buhrman, J. Tromp, and P. VitÃąnyi. Time and space bounds for reversible simulation. In F. Orejas, P. G. Spirakis,

and J. van Leeuwen, editors, *Proceedings of the 28th International Colloquium on Automata, Languages and Programming*, volume 2076 of *Lecture Notes in Computer Science*, pages 1017–1027. Springer Berlin Heidelberg, 2001.

[8] Cyclos Semiconductor. Cyclos Semiconductor announces first commercial implementation of resonant clock mesh technology. http://www.cyclos-semi.com/news/first_commercial_implementation.html, Feb. 2012.

[9] eMarketer. Smartphone users worldwide will total 1.75 billion in 2014. http://www.emarketer.com/Article/Smartphone-Users-Worldwide-Will-Total-175-Billion-2014/1010536, Jan. 2014.

[10] M. P. Frank. *Reversibility for efficient computing*. PhD thesis, Massachusetts Institute of Technology, 1999.

[11] R. Jain, D. Molnar, and Z. Ramzan. Towards a model of energy complexity for algorithms. In *Proceedings of the 2005 IEEE Wireless Communications and Networking Conference*, volume 3, pages 1884–1890, 2005.

[12] G. Kissin. Upper and lower bounds on switching energy in vlsi. *J. ACM*, 38(1):222–254, Jan. 1991.

[13] J. G. Koomey. *Growth in data center electricity use 2005 to 2010*. Analytics Press, Aug. 2011.

[14] J. G. Koomey, S. Berard, M. Sanchez, and H. Wong. Implications of historical trends in the electrical efficiency of computing. *IEEE Annals of the History of Computing*, 33(3):46–54, March 2011.

[15] V. A. Korthikanti and G. Agha. Towards optimizing energy costs of algorithms for shared memory architectures. In *Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 157–165, New York, NY, USA, 2010. ACM.

[16] V. A. Korthikanti, G. Agha, and M. R. Greenstreet. On the energy complexity of parallel algorithms. In G. R. Gao and Y.-C. Tseng, editors, *Proceedings of the 2011 International Conference on Parallel Processing*, pages 562–570, 2011.

[17] J. Ladyman, S. Presnell, A. J. Short, and B. Groisman. The connection between logical and thermodynamic irreversibility. *Studies In History and Philosophy of Science Part B: Studies In History and Philosophy of Modern Physics*, 38(1):58–79, 2007.

[18] B. Lambson, D. Carlton, and J. Bokor. Exploring the thermodynamic limits of computation in integrated systems: magnetic memory, nanomagnetic logic, and the Landauer limit. *Phys Rev Lett*, 107(1):010604, 2011.

[19] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):261–269, 1961.

[20] K.-J. Lange, P. Mckenzie, and A. Tapp. Reversible space equals deterministic space. In *Proceedings of the 12th Annual IEEE Conference on Computational Complexity*, pages 45–50, 1997.

[21] Y. Lecerf. Logique mathematique. machines de Turing réversibles. récursive insolubilité en $n \in N$ de l'équation $u = \theta^n u$, où θ est un "isomorphisme de codes". note. *Comptes rendus hebdomadaires des séances de l'Academie des sciences*, 257(18):2597–2600, October 1963.

[22] M. Li and P. Vitányi. Theory of thermodynamics of computation. In *Proceedings of the IEEE Physics of Computation Workshop*, pages 42–46, 1992.

[23] M. Li and P. Vitányi. Reversibility and adiabatic computation: trading time and space for energy. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 452(1947):769–789, 1996.

[24] M. Li and P. Vitányi. *An introduction to Kolmogorov Complexity and its Applications*. Springer, 3rd edition, 2008.

[25] C. Lutz and H. Derby. Janus: a time-reversible language. *Caltech class project*, 1982.

[26] O. J. E. Maroney. Generalizing Landauer's principle. *Phys. Rev. E*, 79:031105, Mar 2009.

[27] M. Morrison and N. Ranganathan. Design of a reversible alu based on novel programmable reversible logic gate structures. In *Proceedings of the 2011 IEEE Computer Society Annual Symposium on VLSI*, pages 126–131, 2011.

[28] M. Nachtigal, H. Thapliyal, and N. Ranganathan. Design of a reversible single precision floating point multiplier based on operand decomposition. In *Proceedings of the 10th IEEE Conference on Nanotechnology*, pages 233–237, Aug 2010.

[29] M. Nachtigal, H. Thapliyal, and N. Ranganathan. Design of a reversible floating-point adder architecture. In *Proceedings of the 11th IEEE Conference on Nanotechnology*, pages 451–456, Aug 2011.

[30] B. Piechocinska. Information erasure. *Phys. Rev. A*, 61:062314, May 2000.

[31] V. S. Sathe, S. Arekapudi, A. T. Ishii, C. Ouyang, M. C. Papaefthymiou, and S. Naffziger. Resonant-clock design for a power-efficient, high-volume x86-64 microprocessor. *IEEE Journal of Solid-State Circuits*, 48(1):140–149, 2013.

[32] M. K. Thomsen, H. B. Axelsen, and R. Glück. A reversible processor architecture and its reversible logic design. In *Proceedings of the 3rd International Conference on Reversible Computation*, pages 30–42, Gent, Belgium, 2012.

[33] C. Velazco. AMD's new FX processor reaches world record clock speed. http://techcrunch.com/2011/09/13/amds-new-fx-processor-reaches-world-record-clock-speed/, Sept. 2011.

[34] C. Vieri, M. J. Ammer, M. Frank, N. Margolus, and T. Knight. A fully reversible asymptotically zero energy microprocessor. In *Proceedings of the Power Driven Microarchitecture Workshop*, pages 138–142, 1998.

[35] C. J. Vieri. *Reversible computer engineering and architecture*. PhD thesis, Massachusetts Institute of Technology, 1999.

[36] P. Vitányi. Time, space, and energy in reversible computing. In *Proceedings of the 2Nd Conference on Computing Frontiers*, CF '05, pages 435–444, New York, NY, USA, 2005. ACM.

[37] R. Williams. Space-efficient reversible simulations. Technical report, DIMACS REU report, 2000. http://web.stanford.edu/~rrwill/spacesim9_22.pdf.

[38] S. Yates, E. Daley, B. Gray, J. P. Gownder, and R. Batiancila. *Worldwide PC Adoption Forecast, 2007 To 2015*. Forrester, June 2007. http://www.foresightfordevelopment.org/sobipro/download-file/46-724/54.

[39] T. Yokoyama. Reversible computation and reversible programming languages. *Electronic Notes in Theoretical Computer Science*, 253(6):71–81, 2010.

[40] W. H. Zurek. Thermodynamic cost of computation, algorithmic complexity and the information metric. *Nature*, 341:119–124, Sept. 1989.