Almost Minimum-Redundancy Construction of Balanced Codes Using Limited-Precision Integers

Danny Dubé and Mounir Mechqrane
Department of Computer Science and Software Engineering, Université Laval
Quebec City, Quebec, Canada
Email: Danny.Dube@ift.ulaval.ca and Mounir.Mechqrane.1@ulaval.ca

***Abstract*—We present a technique based on permutations, the well known arcade game Pacman, and limited-precision integers to encode data into balanced codewords. The redundancy that is introduced by the encoding is particularly low. The results are noticeably better than those of previous work. Still, the resources required by our technique remain modest: there is no need for costly calculations using large integers and the time and space complexity for encoding or decoding a block is linear.**

I. Introduction

A. Balanced Blocks

A block B of M bits is said to be *balanced* if it contains an equal number of zeros and ones. Note that M has to be an even number. Applications of balanced codes are mentioned in Subsection I-B. When one needs data to be encoded in the form of balanced blocks, one must have access to an *encoding function* 'Enc' that transforms arbitrary input data into balanced blocks. In this work, we consider *binary* input data. Encoding data into balanced blocks necessarily introduces redundancy. Indeed, only $\binom{M}{M/2}$ of the 2^M blocks of M bits happen to be balanced. Let $\mathcal{B}_M$ be the set of the M-bit balanced blocks. Let Q, $1 \leq Q < M$, be the size of the blocks of input data, i.e. $\mathrm{Enc} : 2^Q \rightarrow \mathcal{B}_M$. Let $P = M - Q$ be the number of bits of redundancy (*parity bits*) that 'Enc' introduces per block. Obviously, the smaller P is, the better 'Enc' is. Function 'Enc' is a fixed-to-fixed code. Let $\mathrm{Dec} : \mathcal{B}_M \rightarrow 2^Q$ be the corresponding decoding function.

Mathematically, devising optimal functions 'Enc' and 'Dec' is a trivial task. First, one determines Q and M. If the application specifies M, then we let Q be $\left\lfloor \log \binom{M}{M/2} \right\rfloor$.[1] Otherwise, the application specifies Q and we let M be the smallest even integer such that $\binom{M}{M/2} \geq 2^Q$. Second, one may use enumerative coding to define 'Enc' (and 'Dec') [1]. To do so, one enumerates the 2^Q unconstrained input blocks in lexicographic order and the first 2^Q M-bit balanced blocks also in lexicographic order and then lets 'Enc' be the one-to-one mapping from the former to the latter. The mapping that defines 'Enc' (and 'Dec') may be stored in a *lookup table*, like the one in Figure 1. Alternatively, the mapping may be implemented using a pair of procedures that build, by calculations, the i-th balanced block when presented the i-th input block, and vice versa.

[1] In this paper, all logarithms are to the base 2.

Unfortunately, the strategies based on enumerative coding are not practical because they do not scale well. The size of lookup tables increases exponentially with Q. On the other hand, procedures based on calculations require the manipulation of large integers, which is costly in time. The impracticality of standard enumerative coding has lead many researchers to develop faster, approximate strategies. Subsection I-C presents their work. We aim at the same goal.

B. Motivation

Balanced codes have many applications. They can be used to detect unidirectional errors [2], to detect errors due to low-frequency disturbances in magnetic storage [3], to reduce noise in VLSI integrated circuits [4], to maintain the integrity of data in write-only storage media, which may be altered due to a `0` getting changed to a `1` [5], to establish delay-insensitive communications in asynchronous systems [6], to synchronize data transmitted in fiber optic [7], and to boost data transfer rates via RFID channels [8].

C. Previous Work

Knuth presented the first practical construction technique for balanced blocks [9]. His technique is quite simple and it is based on the following observation: an arbitrary block B of bits can be made balanced by *inverting* the bits in an appropriate prefix of B. Let us denote by $\overline{\cdot}$ the inversion operator; i.e. $\overline{\mathtt{0}} = \mathtt{1}$ and $\overline{\mathtt{1}} = \mathtt{0}$. We extend the operator so that it operates bitwise on sequences. Given an arbitrary block B of *even* length Q, Knuth's technique consists in splitting B into a prefix u and a suffix v, where $0 \leq |u| < Q$, such that $\overline{u} \cdot v$ is balanced. Knuth showed that such an appropriate prefix always exists. Merely transforming input blocks that way would not make a valid (i.e. reversible) function 'Enc'. The length $|u|$ has to be encoded somewhere in the transformed block. To do so, Knuth's technique recursively relies on a shorter balanced code. The codewords of the latter have length P, where P is large enough to encode $|u|$; i.e. $\binom{P}{P/2} \geq Q$. Still, P is typically small enough to use a lookup table. So Knuth's technique encodes B by returning $\mathrm{Enc}(|u|) \cdot \overline{u} \cdot v$.

Knuth estimated the redundancy added by his technique to be $P \approx \log Q \approx \log M$ bits. He noted that this redundancy is about twice the optimum one:

$$M - \log \binom{M}{M/2} \approx \frac{1}{2} \log M.$$

Input	Balanced
0000	000111
0001	001011
0010	001101
0011	001110

Input	Balanced
0100	010011
0101	010101
0110	010110
0111	011001

Input	Balanced
1000	011010
1001	011100
1010	100011
1011	100101

Input	Balanced
1100	100110
1101	101001
1110	101010
1111	101100

Fig. 1. Lookup table for 'Enc' for $Q = 4$ and $M = 6$.

It means that there is room for improvement.

Indeed, much research has been conducted to reduce the redundancy of Knuth's algorithm. Weber and Immink noted that an input block B may have multiple (between 1 and $\frac{M}{2}$) adequate prefixes [10]. This freedom in selecting encodings is the cause of part of the extra redundancy introduced by Knuth's algorithm. The same authors also noted that, in theory, this selection freedom could be used to convey information and they showed that, on average, the amount of information that could be conveyed per block this way is $A_{\mathrm{SF}} \approx \frac{1}{2}\log M - 0.916$ [11]. They devised a scheme that significantly reduces the redundancy compared to Knuth's algorithm. Still, they did not succeed to fully exploit the selection freedom. Al-Rababa'a et al. noticed that this selection freedom is a good candidate for bit recycling [12]. This technique achieved a better improvement by transmitting almost A_{SF} bits per balanced block, on average.

D. Contributions

This work presents an alternative to Knuth's algorithm to create balanced codes with even less redundancy. We propose a new technique that is based on permutations and Pacman.[2] Indeed, the coding process of this algorithm can be viewed as action performed by a special Pacman that consumes *and produces* pills of *information*. We demonstrate experimentally and analytically that our algorithm closes the redundancy gap mentioned above while requiring low resources in time, space, and register size.

II. Our Technique

Our technique uses a variety of tools. Some are usual, some are new. In this section, we introduce the necessary notation, definitions, concepts, and algorithms step by step.

A. Conventional Representation of Permutations

We denote a (conventional) *permutation* of n elements by by $(a_1, \ldots, a_n)$, where $a_i \neq a_j$ whenever $1 \leq i < j \leq n$. We define $\mathcal{P}_n$ to be the set of permutations of $\{1, \ldots, n\}$.

B. Indexed Representation of Permutations

Our technique also makes use of an alternative representation of the permutations: the *indexed* representation. The indexed representation indicates the *relative position* of each of the numbers that appear in a permutation $\pi \in \mathcal{P}_n$. The leftmost position is 1. We use the term "relative" because the indexed representation indicates, for each number a, the position of a in the permutation that remains if we remove the larger numbers $a+1, \ldots, n$ from π. We denote an indexed permutation η by $\langle \iota_1, \ldots, \iota_n \rangle$, where $1 \leq \iota_i \leq i$, for $1 \leq i \leq n$. Let $\mathcal{H}_n$ be the set of indexed permutations with n indices. The conversion of permutations from the conventional representation to the indexed representation is performed using a family of functions, $\{\mathsf{H}_n\}_{n=1}^{\infty}$, where $\mathsf{H}_n : \mathcal{P}_n \to \mathcal{H}_n$, which are inductively defined as follows.

$$\begin{aligned} \mathsf{H}_1((1)) &= \langle 1 \rangle \\ \mathsf{H}_n((a_1, \ldots, a_{i-1}, n, a_{i+1}, \ldots, a_n)) & \\ = & \\ \mathsf{H}_{n-1}((a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n)) \cdot \langle i \rangle & \end{aligned}$$

Note that we overload the operator '$\cdot$' to also denote the extension of a permutation. The reverse conversion is performed using the family of functions $\{\mathsf{P}_n\}_{n=1}^{\infty}$, where $\mathsf{P}_n : \mathcal{H}_n \to \mathcal{P}_n$. An example showing both representations is given in Figure 2. An interesting property of indexed permutations is that every index is independent of the others, which is not the case with conventional permutations.

$$\pi = (2, 4, 1, 5, 3) = \mathsf{P}_5(\eta) \qquad \eta = \langle 1, 1, 3, 2, 4 \rangle = \mathsf{H}_5(\pi)$$

Fig. 2. The conventional and the indexed representations of a permutation.

C. Permutations and Balanced Blocks

Permutations have some relation to balanced codes. Let M be an *even* integer. Let us suppose that we wish to build a balanced block B of length M. Let us suppose further that we have at hand a permutation $\Pi \in \mathcal{P}_M$ that, for some reason, results from processing input data. Then we can extract a balanced block B from Π by keeping the *parity* of the elements of Π. Let us denote this operation by $B = \Pi \bmod 2$. For example, if Π is $(5, 4, 2, 7, 1, 8, 3, 6)$, then $B = \texttt{10011010}$ can be extracted. If we could manage to transform the entire input data into permutations like Π, we would have a complete procedure for 'Enc'.

However, there are missing operations. We already know that we have to devise a way to convert the input data into permutations from $\mathcal{P}_M$. But we face another, less obvious challenge: we have to deal with Π once we have extracted B from it. Clearly, it would pointless to extract $\Pi \bmod 2$ again. We cannot simply throw Π away either, since Π still contains information from the input.

Let us characterize the information that remains in Π once B has been extracted. In order to do so, let us take the point of view of the decoder and assume that B is known but not Π. B describes the positions of the even and odd numbers inside of Π. However, nothing is divulged about the order of the even numbers relative to each other, neither about the odd numbers. If the decoder were to receive the relative order of the even numbers and that of the odd numbers, then the decoder would

[2]The name is inspired by the well known PAC-MAN video game. The trademark PAC-MAN is a property of BANDAI NAMCO.

hold the full information about Π. These orders are equivalent, up to renumbering, to permutations of $\frac{M}{2}$ elements. So there is a one-to-one mapping between permutations like $\Pi \in \mathcal{P}_M$ and triples like $(B, \pi, \pi') \in \mathcal{B}_M \times \mathcal{P}_{M/2} \times \mathcal{P}_{M/2}$. In the previous example, we have $\Pi = (5, 4, 2, 7, 1, 8, 3, 6)$ and $B = \Pi \bmod 2 = \texttt{10011010}$. The relative positions of the even numbers of Π are given by $\pi = (2, 1, 4, 3)$ (which is $(4, 2, 8, 6)$, monotonically renumbered); those of the odd numbers, by $\pi' = (3, 4, 1, 2)$.

This transform may serve in an effective implementation of 'Enc', which would proceed in four steps. First, input data somehow gets embedded into $\Pi \in \mathcal{P}_M$. Second, Π gets transformed into (B, π, π'). Third, B gets emitted as encoded data. Fourth, the information contained in π and π' somehow gets reused in the construction of a new permutation from $\mathcal{P}_M$. The transform can be reversed, so it may serve as well in an implementation of 'Dec'. There remains to devise a way to perform the "somehow" tasks of the first and fourth steps.

D. Rebuilding a Large Permutation from Small Permutations

It is not necessarily obvious to see how to reuse the information contained in $\pi, \pi' \in \mathcal{P}_{M/2}$ and embed it into a new $\Pi \in \mathcal{P}_M$ Fortunately, indexed permutations can be of help, here. It is less difficult to see how to reuse the information from $\eta, \eta' \in \mathcal{H}_{M/2}$ and to embed it into $H \in \mathcal{H}_M$, mainly because indices in indexed permutations are independent from each other. So we have reduced the problem to the following three, simpler steps. First, let η be $\mathsf{H}_{M/2}(\pi)$ and η' be $\mathsf{H}_{M/2}(\pi')$. Second, we somehow transfer the information from η and η' into H. Third, we let Π be $\mathsf{P}_M(H)$. In Subsections II-E to II-H, we explain how to perform the "somehow" task of the second step.

E. Pacman

In order to extract the information from $\eta, \eta' \in \mathcal{H}_{M/2}$ and embed it into (a new) $H \in \mathcal{H}_M$, we take inspiration from the famous video game Pacman. The original Pacman consumes pills with the intent to make points. In our setting, instead of performing only "input" actions, Pacman performs both "input" and "output" actions. It does so with the intent to transfer the information from η and η' into H. During a transformation cycle, Pacman consumes all the indices of η and η' and produces all the indices of H. Since $\eta, \eta' \in \mathcal{H}_{M/2}$ together contain less information than $H \in \mathcal{H}_M$, we take the opportunity to have Pacman consume input bits, also.

Since Pacman's goal is to transfer information, it has a memory. Its memory enlarges when it consumes an index and its memory shrinks when it produces an index. If Pacman consumes many indices in a row, its memory enlarges considerably. It is preferable to have Pacman alternate between consumption and production, allowing its memory to remain at a reasonable size.

F. Small Memories

We say that Pacman has a *small memory*. Its small memory is intended to hold a single value: an integer in the range $1, \ldots, \sigma$, where $\sigma \geq 1$. The conventional way of measuring the size of the memory would consider it to be $(\log \sigma)$-bits wide. Instead, we choose to consider it to have size σ.

Pacman's operations have the following effect on its memory. Let σ and σ' be the memory *sizes* and v and v' be the *values* stored in the memory before and after a given operation, respectively. If Pacman consumes an index of value i of range size ρ, then we have:

$$(\sigma', v') \quad = \quad (\rho \times \sigma,\ \rho \times (v-1) + i)\,.$$

If Pacman produces an index i of range size ρ, then we have:

$$(\sigma', i, v') \quad = \quad \left(\left\lceil \frac{\sigma}{\rho} \right\rceil,\ ((v-1) \bmod \rho) + 1,\ \left\lceil \frac{v}{\rho} \right\rceil\right).$$

Rounding is necessary because σ need not be divisible by ρ, in general. Note that we view the consumption of an input bit as the consumption of an index of range size 2.

We say that index consumption introduces *no* redundancy. On the other hand, we say that index production *does* introduce redundancy, in general, because of rounding. However, the worst-case added redundancy remains modest as $\sigma' \leq \frac{\sigma+\rho-1}{\rho}$.

G. Pacman's Programming

We choose to use the same sequence of operations each time Pacman transforms $\eta, \eta' \in \mathcal{H}_{M/2}$ and input bits $b_1, \ldots, b_Q$ into a new $H \in \mathcal{H}_M$. We call that sequence *Pacman's programming*.

We define a programming $\mathbb{P}$ to be a permutation of the following instructions: $\mathsf{E}_1, \ldots, \mathsf{E}_{M/2}, \mathsf{O}_1, \ldots, \mathsf{O}_{M/2}, \mathsf{B}_1, \ldots, \mathsf{B}_Q$, and $\mathsf{L}_1, \ldots, \mathsf{L}_M$. Note that $\mathbb{P}$ contains $2 \times M + Q$ instructions. There are no *syntactic* restrictions on $\mathbb{P}$.

Let η be $\langle \iota_1, \ldots, \iota_{M/2} \rangle$, η' be $\left\langle \iota'_1, \ldots, \iota'_{M/2} \right\rangle$, and H be $\langle \iota''_1, \ldots, \iota''_M \rangle$. The semantics of the instructions is the following: instruction E_i (for "even") directs Pacman to consume ι_i; instruction O_i (for "odd") directs Pacman to consume ι'_i; instruction B_i (for "bit") directs Pacman to consume b_i; and instruction L_i (for "large") directs Pacman to produce ι''_i.

An additional piece of information gets attached to $\mathbb{P}$. It is Pacman's memory size σ_0 at the beginning of the execution of $\mathbb{P}$. Note that, given σ_0, it is possible to determine Pacman's memory size at any step during the execution of $\mathbb{P}$. Let σ_i be the memory size after the first i instructions from $\mathbb{P}$ have been executed. In particular, the memory size at the end of the execution of $\mathbb{P}$ is $\sigma_{2\times M+Q}$.

H. Validity of a Programming

Although that there are no syntactic restrictions on $\mathbb{P}$, there are *semantic* restrictions, namely on Pacman's memory size. We impose a maximum memory size Ω. $\mathbb{P}$ is *valid* if its instructions are arranged so that:

$$\forall 0 \leq i \leq 2 \times M + Q.\ \sigma_i \leq \Omega \quad \text{and} \quad \sigma_{2\times M+Q} \leq \sigma_0.$$

A subtle consequence of the validity condition is that, if one chooses too large a Q, then it becomes impossible to establish a valid $\mathbb{P}$.

I. Encoding Cycle

We define an *encoding cycle* to be the computations that cause the consumption of a block of Q input bits and the production of a balanced block. The computations of an encoding cycle also involve the manipulation of data that is internal to the encoder: Pacman's memory and some permutations. Internal data is passed over from one cycle to the next. Let us number the current cycle as cycle $\#t$. The internal data that is passed over is Pacman's memory and two small permutations, π_{t-1} and π'_{t-1}. Pacman's memory has size $\sigma_{2\times M+Q}$ and contains a certain value.[3] The first step of cycle $\#t$ is the following: $\eta_t = \mathsf{H}_{M/2}(\pi_{t-1})$ and $\eta'_t = \mathsf{H}_{M/2}(\pi'_{t-1})$. The second step of cycle $\#t$ consists in resizing Pacman's memory: the new size is σ_0. Note, however, that the *value* in Pacman's memory is left unchanged. This resizing is consistent since $\sigma_{2\times M+Q} \leq \sigma_0$. The third step consists in executing $\mathbb{P}$. This causes Pacman to consume η_t and η'_t completely. Additionally, this causes Pacman to produce H_t. The Q bits $b_1, \ldots, b_Q$ that are also consumed by Pacman get read from the input. The fourth step is the following: $\Pi_t = \mathsf{P}_M(H_t)$. The fifth step decomposes Π_t into (π_t, π'_t, B_t). The sixth step writes B_t to the output. Finally, Pacman's memory, π_t, and π'_t are passed over to cycle $\#t+1$.

J. Decoding Cycle

The decoding cycle $\#t$ merely undoes what the encoding cycle $\#t$ did. We assume that Pacman's memory, π_t and π'_t are passed over by cycle $\#t+1$. (Note that the cycles themselves have to be reversed, not just the computations in each cycle.) The value in Pacman's memory, π_t and π'_t are exactly the same as those that existed at the end of the *encoding* cycle $\#t$. The balanced block B_t gets read from the input. The triple (π_t, π'_t, B_t) gets blended to rebuild Π_t. Next, $H_t = \mathsf{H}_M(\Pi_t)$. Then, Pacman performs a *reversed* execution of $\mathbb{P}$. The reversal not only means that the instructions are executed from right to left but also that the instructions that triggered consumption in the encoder now trigger production in the decoder, and vice versa. The reversed execution rebuilds η_t, η'_t, and Q bits. The Q bits get pushed on a stack to be written to the output later. They must not be written to the output immediately because the rebuilt input blocks are recovered from the last of the first. Finally, $\mathsf{P}_{M/2}(\eta_t)$ and $\mathsf{P}_{M/2}(\eta'_t)$ are passed over to cycle $\#t-1$.

K. Initialization and Termination

There remains to present procedures for initialization and finalization. We present those of the encoder. Those for the decoder are simply reversed procedures, in the reversed order.

The initialization prepares the encoder's internal data. We choose to initialize it with fixed values. Pacman's memory is initialized to value 1 and size $\sigma_{2\times M+Q}$, a plausible state left by a hypothetic cycle $\#0$. Permutations π_0 and π'_0 are initialized to the identity permutation, $\left(1, \ldots, \frac{M}{2}\right)$.

[3]Note that the index in variable $\sigma_{2\times M+Q}$ is *not* the cycle number: it is the step number in $\mathbb{P}$.

The finalization has to deal with many details. In the first step, the encoder has to handle the possibility that the sequence of input bits may have a length that is not a multiple of Q. Consequently, the encoder appends between 0 and $Q-1$ bits of padding to the input. The following steps are intended to flush out all the internal data, including the length of the padding. Note that the encoder cannot use its Pacman-and-permutations machinery anymore as it is precisely the internal data maintained by that machinery that the encoder needs to flush out. Finally, note that the encoder is not relieved of the obligation to output balanced blocks. In the following steps, we make use of classical enumerative coding (denoted by 'Enc') to encode various small integers into balanced blocks. So, in the second and third steps, the encoder emits $\mathrm{Enc}(l)$, where l is the length of the padding, and $\mathrm{Enc}(v_T)$, where v_T is the value stored in Pacman's memory after the last cycle (cycle $\#T$). In the last step, it emits one block per element of π_T and π'_T.

L. Design of a Programming

In subsections II-G and II-H, even if we formally explain what $\mathbb{P}$ is and under which condition $\mathbb{P}$ is valid, we do not really explain what $\mathbb{P}$ one should try to get nor how to design it. The design of $\mathbb{P}$ has to be made carefully.

We have not formally analyzed the complexity of the design of $\mathbb{P}$ but we suspect it is NP-hard. Indeed, it does not seem easy to design an optimal $\mathbb{P}$, for different definitions of "optimal". In our experiments, we have considered two definitions of optimality. In each definition, we assume that M is imposed by the application. The first definition consists in choosing $Q \leq \log \binom{M}{M/2}$ a priori and trying to determine $\Omega_{\min}$, which is the smallest Ω for which there exists a valid $\mathbb{P}_\Omega$. The second one consists in choosing Ω and $\sigma_0 \leq \Omega$ a priori and trying to determine $Q_{\max}$, which is the largest Q for which there exists a valid $\mathbb{P}_Q$.

We have not tried to find optimal values under either definition. Instead, we rely on two heuristics.

The first heuristics relies on an a priori choice of Q, Ω, and σ_0 and designs a programming in a *greedy* way. The consumption and production instructions are put in two separate queues in arbitrary order. Then, an iterative process successively picks one new instruction at a time from one of the queues, favouring the consumption instructions over the production ones whenever possible and favouring the first consumption instructions that allows Pacman's memory size to remain within Ω. This heuristics is not guaranteed to be able to design a valid programming.

The second heuristics relies on an a priori choice of σ_0 and the availability of a programming $\mathbb{P}$, where a priori values of Q and Ω implicitly follow from these. The heuristics considers the swapping of two arbitrary instructions in $\mathbb{P}$ and checks whether it would reduce Pacman's memory size at least at certain steps, especially at steps where the maximum memory size is reached. The heuristics continues until no more swappings may lead to a reduction.

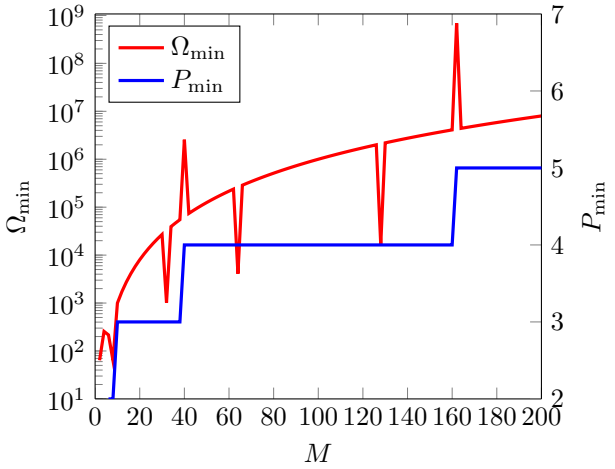

Fig. 3. Requirements on Pacman's memory size and parity bits for various balanced-block sizes.

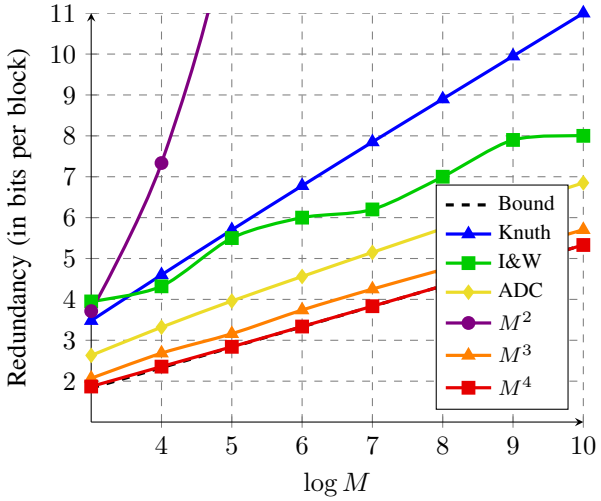

Fig. 4. Redundancy introduced by different techniques for various balanced-block sizes.

III. Experimental and Theoretical Results

We ran experiments in two modes: one for minimum redundancy, the other for limited-precision integers. Also, we made a theoretical analysis to obtain upper bounds on the redundancy in the mode for limited-precision integers.

A. Mode for Minimum Redundancy

In this mode, we set Q to $\left\lfloor \log \binom{M}{M/2} \right\rfloor$, for different values of M, and observe the effect on $\Omega_{\min}$. We choose the initial memory size of Pacman to be $\sigma_0 = \max(64, 2^{\lceil \log M \rceil})$. The design of the programming is done using our two heuristics. The first curve in Figure 3 shows the results. The growth rate of $\Omega_{\min}$ seems to be polynomial since we can roughly estimate that $\Omega_{\min}$ is multiplied by 10 each time M doubles.

B. Mode for Limited-Precision Integers

In this mode, we arbitrarily set Ω and M to different values and observe the effet on $P_{\min}$ ($P_{\min} = M - Q_{\max}$). More precisely, we set Ω to be one of different *functions* of M: M^2, M^3, M^4, and other functions of M but results for the latter are not reported here. Once again, σ_0 is set to $\max(64, 2^{\lceil \log M \rceil})$. In Figure 3, the second curve shows the growth of $P_{\min}$ when Ω is set to M^3. Choosing $\Omega = M^3$ means that it is sufficient for the registers (and the memory cells) of the computer to be three times as large as the size of the encoding of M in binary, which is reasonable.

C. Upper Bounds on the Redundancy

We have derived upper bounds on the worst-case redundancy that one could face in limited-precision mode, if we set Ω to M^k, for $k \geq 2$. Here are the key ideas. During a cycle, when an index of range size ρ is produced, rounding may increase σ by $\rho - 1$, in absolute terms. In relative terms, this increase is small because we assume $\sigma > M^{k-1}$, thanks to the greedy heuristics. There remains to cumulate all the increases of the cycle. We omit the rest of the details. The bounds guarantee that the redundancy converges very quickly towards the theoretical minimum redundancy when k grows. Figure 4 presents three curves that show the upper bounds on the redundancy when Ω is set to M^2, M^3, and M^4. The other curves show the lower bound on the redundancy that has to be introduced per block, the redundancy by Knuth's algorithm[9], that by Immink and Weber's technique [10], and that by Al-Rababa'a et al.'s technique [12].

Acknowledgment

The authors wish to thank the reviewers for their comments.

References

[1] T. Cover, "Enumerative source encoding," *IEEE Trans. on Information Theory*, vol. 19, no. 1, pp. 73–77, 1973.

[2] S. J. Piestrak, "Design of self-testing checkers for unidirectional error detecting codes," *Scientific Papers of the Institute of Technical Cybernetics of the Technical University of Wroclaw*, 1995.

[3] K. A. S. Immink, "Coding techniques for the noisy magnetic recording channel: A state-of-the-art report," *IEEE Trans. on Communications*, vol. 37, no. 5, pp. 413–419, 1989.

[4] J. F. Tabor, "Noise reduction using low weight and constant weight coding techniques," Computer Science and Artificial Intelligence Lab, MIT, Tech. Rep. AITR-1232, May 1990.

[5] E. L. Leiss, "Data integrty in digital optical disks," *IEEE Trans. on Computers*, vol. 100, no. 9, pp. 818–827, 1984.

[6] M. Blaum and J. Bruck, "Coding for skew-tolerant parallel asynchronous communications," *IEEE Trans. on Information Theory*, vol. 39, no. 2, pp. 379–388, 1993.

[7] A. X. Widmer and P. A. Franaszek, "A DC-balanced, partitioned-block, 8B/10B transmission code," *IBM Journal of Research and Development*, vol. 27, no. 5, pp. 440–451, 1983.

[8] G. D. Durgin, "Balanced codes for more throughput in RFID and backscatter links," in *IEEE International Conference on RFID Technology and Applications*, 2015, pp. 65–70.

[9] D. E. Knuth, "Efficient balanced codes," *IEEE Trans. on Information Theory*, vol. 32, no. 1, pp. 51–53, 1986.

[10] K. A. S. Immink and J. H. Weber, "Very efficient balanced codes," *IEEE Journal on Selected Areas in Communications*, vol. 28, no. 2, pp. 188–192, 2010.

[11] J. H. Weber and K. A. S. Immink, "Knuth's balanced codes revisited," *IEEE Trans. on Information Theory*, vol. 56, no. 4, pp. 1673–1679, 2010.

[12] A. Al-Rababa'a, D. Dubé, and J.-Y. Chouinard, "Using bit recycling to reduce Knuth's balanced codes redundancy," in *Canadian Workshop on Information Theory*, 2013, pp. 6–11.