Towards Profiling the Virtual Memory Performance of Bignums in Functional Languages

Robert S. Moore II, David Grima, and Marco T. Morazán
Seton Hall University
Department of Mathematics and Computer Science
400 South Orange Avenue
South Orange, NJ 07079 USA
E-mail: {moorerob,grimadav,morazanm}@shu.edu

ABSTRACT
Functional programming languages are based on the mathematical formalism known as λ-calculus and provide a programming environment at a very high level of abstraction. The software produced with these languages is associated with low development time, a high degree of reliability, and an ease of maintenance rarely found in other languages. One of the most attractive features that endow functional languages with these properties is their support for integers of arbitrary length and absolute precision called bignums. There have been many different implementations of bignums, but very little research has been conducted on their interaction with memory. In this article, we present empirical results obtained from studying the virtual memory performance of programs, evaluated by the MT system, that use bignums implemented using immutable lists. The empirical data presented serves as a baseline against which to compare implementation improvements and strongly suggests that $FIFO$ performs as well as LRU as a page replacement policy for the MT heap where bignums are allocated. The data also suggests implementation improvements that are discussed as part of our future work.

1. INTRODUCTION
Functional programming languages are based on the mathematical formalism known as λ-calculus and have been in existence since the 1950's when John McCarthy invented Lisp[10]. Functional languages are best known for providing a programming environment at a very high level of abstraction that encourages the use of first class and higher-order functions. This means that functions can be passed as arguments to functions and that functions can be returned as the result of function evaluation. Functional languages, however, also offer support for integers of arbitrary size and precision, called *bignums*. Bignums liberate programmers from integer representation limitations imposed by hardware. For example, the largest known prime number as of March 14, 2005, $2^{25,964,951}$-1, has $7,816,230$ digits[11] and it is impossible to represent it using a single hardware word in modern computers. In order to manipulate integers of this magnitude, bignums are used to represent integers as compound data structures occupying multiple hardware words.

Bignums are used in fields such as cryptography, security systems, and computational algebra where computations involve astronomically large integers. In cryptography, for example, bignums are used represent very large prime numbers utilized to shield hashing algorithms from attacks that can exploit the existence of small factors of a hashing key. The main advantages that bignums offer over standard hardware-limited integers are their size, precision, and the ability they provide to implement computation algorithms that are impossible to efficiently implement using standard computer hardware.

Support for bignums must include implementations for the arithmetic and relational primitive operations defined for standard hardware-limited integers. These operations include addition (+), subtraction (-), multiplication (*), integer division ($\div$), modulus (mod), less than ($<$), greater than ($>$), equality ($=$), greater than or equal ($\geq$), and less than or equal ($\leq$). Support for bignums in functional languages must also include implementations for $eq?$, $equal?$, and $number?$. Addition and subtraction are typically implemented using the algorithm utilized when performing such operations by hand. Multiplication and division, however, are usually implemented using specialized algorithms to speed-up computation.

There have been no in-depth studies of the memory performance of bignums. Existing benchmark comparisons measure only the number of milli- or micro-seconds required to perform specific tasks[17, 21]. Based on the literature it is difficult to discern which algorithms are best. The empirical data we present is not based on execution times. Instead, we measure heap performance. The data we present is much more useful for deciding the relative strength of different bignum algorithms, because it indicates the amount of memory management required. Furthermore, our data is platform independent.

In this article, we briefly describe several basic bignums algorithms for arithmetic operations and an implementation of bignums using immutable lists for the MT system. In

Proceedings of MASPLAS'05
Mid-Atlantic Student Workshop on Programming Languages and Systems
University of Delaware, April 30, 2005

MT, bignums are considered a primitive type which makes their implementation and use transparent to programmers. Empirical data on the memory allocation and paging performance of a set of benchmark programs that utilize bignums is presented. The set of benchmarks includes $factorial$, $matrix\ multiplication$, and $insertion\ sorting$. The data suggests that $FIFO$ performs as well as LRU as a page replacement policy for the MT heap where bignums are allocated. The result is significant, because it is much easier and more efficient to implement $FIFO$ in software than it is to implement LRU and it is clear that the current trend in programming languages is to use software-based virtual machines like the Java Virtual Machine[8], the DREAM machine for Eden[20], and the region-based abstract machine for ML[9]. The observed results are consistent with previous results that suggest that $FIFO$ is superior to LRU for paging the MT heap when programs that manipulate list-based data are being evaluated.

2. RELATED WORK

2.1 Addition and Subtraction

Many bignum implementations use the basic grade school methods for addition and subtraction[19]. These methods have a computation complexity of $O(n)$ where n is the number of bigits[1] in a bignum. The algorithms used in the implementation of addition and subtraction are the same as those taught in grade schools in the United States.

Addition is carried out bigit by bigit starting with the least significant bigit by adding corresponding bigits. If a carry is generated, it is added to the sum of the next most significant bigits. Subtraction is also done bigit by bigit starting from the least significant bigit. If a borrow is needed the next most significant bigit of the minuend is decreased by 1 and the base of the bignum is added to current bigit of the minuend whose difference with the corresponding bigit of the subtrahend is being computed.

2.2 Multiplication Algorithms

Multiplication of numbers that have a small number of bigits is typically implemented by an algorithm similar to the grade-school method. Each bigit of the multiplier is multiplied by each bigit of the multiplicand to produce a result that is multiplied by $base^p$ where p is the position of the bigit in the multiplier. In order to speed the multiplication process, array-based shifting is often employed instead of multiplication by $base^p$, as it typically incurs much less overhead. The results are all added to produce an answer. To compute a*b, where a has n bigits and bignum b has m bigits, this algorithm requires n hardware multiplies and 1 bignum add for each bigit of b, and has a computation complexity of $O(n * m)$. This multiplication algorithm is only efficient for bignums with a small number of bigits.

More efficient bignum multiplication algorithms have been developed. At a designated cut-off point, many implementations switch to an algorithm that divides the two bignums into halves or thirds. Most algorithms use a strategy similar to the Karatsuba multiplication algorithm that multiplies these pieces using the grade school algorithm and then combines the different results into a final answer[7].

[1]A bigit is a bignum digit.

The complexity of Karatsuba's algorithm is $O(n^{\frac{log3}{log2}})$. The Karatsuba algorithm achieves this by evenly splitting each input, x and y, into pieces x_1 and x_0, and y_1 and y_0 respectively. Each piece has a length that is a power of 2 such that $x = x_1 * b + x_0$, and $y = y_1 * b + y_0$, where b is the power of 2 where the split occurs. The combining formula is:

$$x * y = (b^2 + b)x_1y_1 - b(x_1 - x_0)(y_1 - y_0) + (b + 1)x_0y_0$$

2.3 Division Algorithms

Division of small bignums is typically implemented using the same algorithm used in grade school. In this algorithm, the divisor is repeatedly subtracted from the first few bigits of the dividend until the difference is non-positive to compute the first bigit of the result. The process then iterates with the difference and the remaining bigits of the dividend.

To improve the complexity of bignum division, divide and conquer algorithms have been developed[18]. These division algorithms follow a strategy similar to bignum multiplication. The input is split into pieces to create smaller problems and then the solution of these smaller problems are combined into a final answer.

3. THE MT SYSTEM

The MT system divides memory into 5 distinct spaces that are managed independently and are implemented as different address spaces in a distributed virtual memory (DVM) system implemented in software. These memory spaces are: the MT heap, the MT stack, the MT code space, the MT evaluator virtual machine (MTEVM) space, and the garbage collector space[2]. The basic units of information stored in all spaces are MT S-expressions that have a tag and a value. Tags distinguish the type of value stored. All values are represented by an integer except lists and bignums which are represented by two integers.

The MT heap only stores dynamically allocated list-based structures, closures, and bignums. Previous empirical studies suggest that the MT allocation algorithm fosters locality of reference[12] and that $FIFO$ is as competitive as LRU as a page replacement policy[14, 15]. This property of being able to perform paging using $FIFO$ is very desirable, because $FIFO$ is easily implemented in software and does not incur a per access overhead like LRU.

The MT stack is used for parameter passing and flow control. To evaluate a function, either primitive or user-defined, arguments are pushed onto the stack, a result is computed, the arguments are popped off of the stack, the result is pushed onto the stack, and control is returned to the calling function. Previous empirical studies on the MT stack validate that LRU is superior to $FIFO$ as a page replacement policy. In fact, it has been mathematically proven that LRU is optimal for paging the MT stack[16]. Based on the proof, the $MT\ Stack\ page\ Replacement\ Algorithm$ (MTSRA) was developed which implements LRU without the costly per access overhead associated with LRU. This property is very desirable for a software-based system like MT, because no hardware support for paging is assumed.

[2]The MT garbage collector is currently under development.

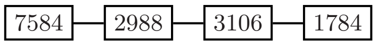

Figure 1: A list of four S-expressions is used to represent 1,784,310,629,887,584.

The MT code space is used to store sequences of MTEVM primitives in sequential memory addresses. Previous empirical studies suggest that LRU performs better than $FIFO$ and that $FIFO$, however, is a competitive page replacement policy for MT's code space[5]. The performance difference between $FIFO$ and LRU consistently becomes insignificant after a certain threshold point for each program. This threshold point occurs at the knee of a graph representing the size of memory against the number of page faults and occurs when code space is large enough to store around 50% of the total number of code space pages needed for execution.

The MTEVM is a register-based machine implemented in software that has access to the memory spaces of the MT System. These memory spaces and a set of registers define the state of MTEVM. A set of computation, flow control, and register-machine primitives are defined that operate on the MT spaces and the set of registers. The MTEVM implements a fetch-execute cycle that loops through an instruction stream stored in code space until the current computation ends. At each iteration of the loop, a primitive instruction is executed that changes the state of the machine. At any time during a computation, only a subset of heap, stack, and code space pages are stored in the MTEVM space. These pages are stored in three different sets of exclusive frames dedicated for each memory space. A demand paging algorithm is implemented for each memory space. As a result, heap, stack, and code space pages do not compete with each other for residence in the MTEVM frames. Heap pages can only be swapped out for heap pages and similarly for stack and code space pages.

4. BIGNUMS IN MT USING LISTS

4.1 Representation

In this study, a bignum is represented as an immutable list of S-expressions. The first S-expression of the list has either a $BIGNUMPOSTAG$ or a $BIGNUMNEGTAG$ tag depending on the sign of the integer represented. The use of two separate tags for positive and negative bignums makes the implementation of arithmetic addition and subtraction faster as described below. In addition to a tag, each S-expression in the list has an integer representing a bigit and a pointer to the next S-expression of the bignum. The first S-expression in the list stores the least significant bigit while the last S-expression in the list stores the most significant bigit. The pointer to the next S-expression in the S-expression storing the most significant bigit is always nil.

Each bigit of a bignum is an integer in a base greater than 10 that represents a multiple of a power of the base. The base for bignums in MT is set when the system is instantiated. For the experiments conducted in this study the base used was 10,000. Figure 1 displays the representation of 1,784,310,629,887,584. This big number is represented using 5 S-expressions. The header S-expression has a tag of

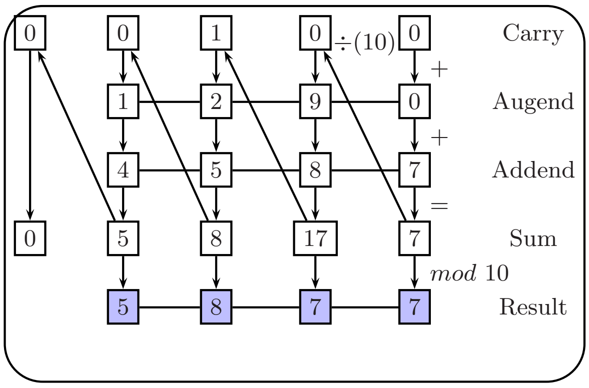

Figure 2: Adding 1,290 to 4,587 yields 5,877.

$BIGNUMPOSTAG$, the least significant bigit (i.e. 7584), and a pointer to the next S-expression of the bignum. Each subsequent S-expression in the list contains the next most significant bigit. The list representing the bignum ends with the S-expression containing the most significant bigit (i.e. 1784) and a nil next S-expression pointer.

4.2 Addition

Both addition and subtraction are performed bigit-by-bigit with carries or borrows the way it is typically taught in grade school. The internal addition functions operate only on the absolute values, or magnitudes, of the bignums and ignore the signs. The bignum returned by addition and subtraction is always positive. When addition involves opposite-signed bignums, e.g. $5 + -2$, subtraction is used, and conversely for subtraction. Addition and subtraction have a computation complexity of $O(n)$, where n is the length of the largest bignum.

The addition loop begins with a carry of 0 and the least significant bigits of the addend and augend. At the i^{th} iteration of the loop, the carry, the i^{th} bigit from the addend, and the i^{th} bigit of the augend are added. The i^{th} bigit of the result is the remainder of this sum and the bignum radix. The carry for the next iteration of the loop is set to the quotient of this sum and the bignum radix. Figure 2 illustrates the process of addition for $1,290 + 4,587$ using a radix of 10. The initial carry of 0 and the least significant bigits 0 and 7 are added. The least significant bigit of the result, 7, is 7 *mod* 10. The carry for the next iteration of the loop, 0, is $7 \div 10$. In the next iteration of the loop, the carry, 0, and the bigits 9 and 8 are added. The next bigit of the result, 7, is 17 *mod* 10. The carry for the next iteration of the loop, 1, is $17 \div 10$. In the next iteration of the loop, the carry, 1, and the bigits 2 and 5 are added. The next bigit of the result, 8, is 8 *mod* 10. The carry for the next iteration of the loop, 0, is $8 \div 10$. In the next iteration of the loop, the carry, 0, and the bigits 1 and 4 are added. The next bigit of the result, 5, is 5 *mod* 10. The carry for the next iteration of the loop, 0, is $5 \div 10$. The loop terminates when there are no more bigits in the augend and the addend. The final carry, 0, is not added to resulting bignum which is 5,877. If the final carry were not 0, it becomes the most significant digit of the result.

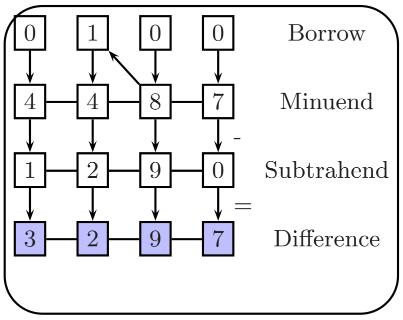

Figure 3: Subtracting 1,290 from 4,587 yields 3,297. The blue boxes indicate new bignums allocated to the heap.

4.3 Subtraction

Like addition, subtraction functions operate only on the magnitudes of the bignums and ignore the signs. When subtraction is called, the bignum with the largest absolute value, or magnitude, is passed as the minuend. This is determined by comparing the arguments and passing the argument with the largest absolute value as the minuend. If the result of these internal computations should be negative, the sign is changed after the magnitude is computed.

Subtraction is performed bigit-by-bigit beginning with the least significant bigits and a borrow of 0. At the i^{th} iteration of the loop, the i^{th} bigit of the subtrahend is subtracted from the (i^{th} bigit of the minuend - the borrow). If the difference is negative, the borrow is set to 1 and the value of the bignum radix is added to the difference to obtain the i^{th} bigit of the result. If the difference is nonnegative, the borrow is set to 0 and the i^{th} bigit is the computed difference.

Figure 3 illustrates the process of subtracting 1,290 from 4,587 using a radix of 10. For the first iteration of the loop, the borrow is 0 and the first bigit of the result, 7, is $(7-0)-0$. Since this difference in nonnegative, the borrow for the next iteration of the loop is set to 0. In the next iteration of the loop, the borrow of 0 and the bigits 8 and 9 yield -1 (i.e. $(8-0)-9$). Since the difference is negative, the next bigit, 9, is obtained by adding 10 (i.e. the radix) to -1. For the next iteration of the loop the borrow is set to 1. In the next iteration of the loop, the borrow of 1 and the bigits 5 and 2 yield 2 (i.e. $(5-1)-2$). Since the difference is positive, it is the next bigit of the result and the borrow is set to 0. The most significant bigit of the result, 3, is obtained using the borrow, 0, and the bigits 4 and 1 (i.e. $(4-0)-1$).

4.4 Multiplication

Multiplication is performed by multiplying each bigit of the multiplier, x, by the multiplicand, y. The result is constructed by adding the intermediate products together at the end of each loop iteration, resulting in a single bignum as the final product when the loop terminates. At the i^{th} iteration of the loop, the product of $x_{0...i-1} * y$ has been calculated and is stored waiting to be added to the remaining intermediate products.

To multiply a bigit from the multiplier, x_i, by the multiplicand, each bigit of the multiplicand is multiplied by x_i. The process starts with a carry of 0. At the i^{th} iteration of this loop, the i^{th} least significant bigit of the product is ((x_i * the i^{th} least significant bigit of the multiplicand) + carry) *mod* radix. The carry for the next iteration of the loop is set to ((x_i * the i^{th} least significant bigit of the multiplicand) + carry) ÷ radix. When x_i has been multiplied by each bigit of the multiplicand the loop is exited and if the carry is non-zero it becomes the most significant bigit of the product.

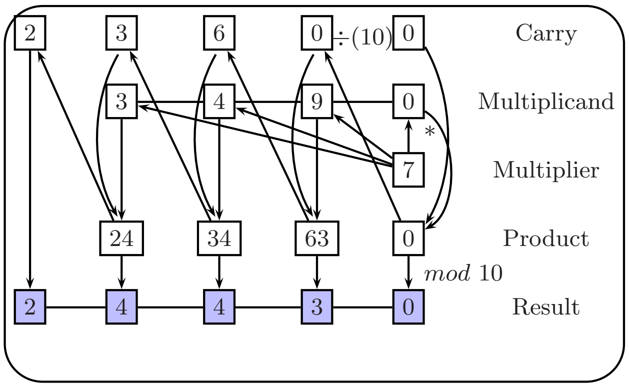

Figure 4: Multiplying 3,490 by 7 yields 24,430. The blue boxes indicate new bignums allocated to the heap.

Figure 4 illustrates the multiplication process for $3,490 * 7$ using a radix of 10. Since the multiplier only has one bigit, the product of that one bigit and the multiplicand yields the result. The process starts with a carry of 0 and the least significant bigit, 0, of the multiplicand. The least significant bigit of the result, 0, is $((7 * 0) + 0)$ *mod* 10. For the next iteration of the loop, the carry, 0, is ((7 * 0) + 0) ÷ 10. In the next iteration of the loop, the carry and the next most significant bigit of the multiplicand, 9, are used to obtain the next most significant bigit of the result, 3 (i.e. ((7 * 9) + 0) *mod* 10), and the next carry, 6 (i.e. ((7 * 9) + 0) ÷ 10). In the following iteration of the loop, the carry and the next most significant bigit of the multiplicand, 4, are used to obtain the next most significant bigit of the result, 4 (i.e. ((7 * 4) + 6) *mod* 10), and the next carry, 3 (i.e. ((7 * 4) + 6) ÷ 10). In the final iteration of the loop, the carry and the next most significant bigit of the multiplicand, 3, are used to obtain the next most significant bigit of the result, 4 (i.e. ((7 * 3) + 3) *mod* 10), and the next carry, 2 (i.e. ((7 * 3) + 3) ÷ 10). After the loop exits, the carry, 2, is non-zero and becomes the most significant bigit of the result.

Presently, the MT Evaluator Virtual Machine only uses this base case algorithm for multiplication. The reason for not implementing more efficient algorithms for this study is that this study is intended as a baseline for comparison against future improvements. This will allow us to objectively quantify gains or losses in execution and memory performance.

4.5 Division

The algorithm used for bignum integer division is based on repeated subtraction. In a loop, the divisor is repeatedly

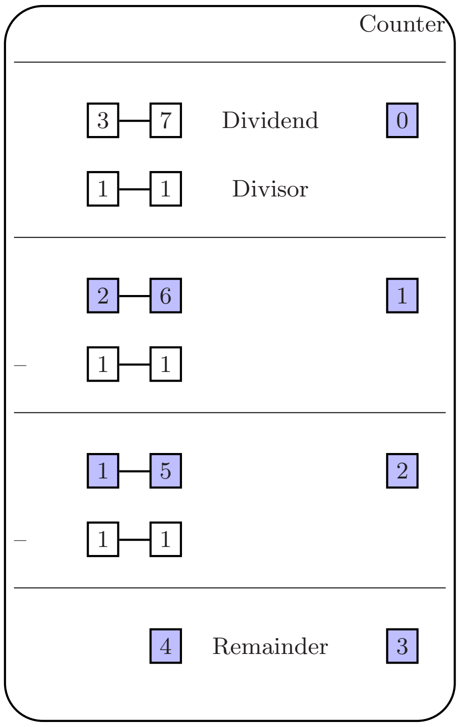

Figure 5: Dividing 37 by 11 using repeated subtraction. The numbers highlighted in light blue are bignums that are created as a result of bignum subtraction.

subtracted from the dividend until the difference is less than the divisor. A counter is kept to track the number of times the divisor is subtracted. Upon exiting the loop, the counter is returned as the quotient. The algorithm used for computing bignum remainder is also based on repeated subtraction. In a loop, the divisor is repeatedly subtracted from the dividend until the difference is less than the divisor. Upon exiting the loop, the last difference computed is returned as the remainder.

Figure 5 illustrates the process of dividing 11 into 37 using a radix of 10. For the first iteration of the loop, the counter is 0 and the divisor, 11, is subtracted from the dividend, 37. The result, 26, is stored in a new bignum and the counter is incremented to 1. On the next iteration, the divisor, 11, is subtracted from the new dividend, 26. The result, 15, is stored in a new bignum and the counter is incremented to 2. On the next iteration, the divisor, 11, is subtracted from the new dividend, 15. The result, 4, is stored in a new bignum and the counter is incremented to 3. Since the result of the last difference, 4, is less than the divisor, 11, the loop terminates and the counter, 3, is returned as the quotient. The process for computing the modulus is the same. Instead of returning the counter, the value of the last difference, 4, is returned.

We note that for the benchmarks in this study the division and modulus operations are not used. An extended baseline study will include benchmarks that rely on division and modulus operations.

4.6 Relational Operations

Bignum relational operations $<$, $\leq$, $=$, $\geq$, and $>$ are implemented in MT by a sequenced set of tests. If necessary, an argument is converted from a fixed integer to a bignum. The first test checks the tags of the bignums. If the tags are different then the result is known. If the tags are the same, then the lengths of the bignums are tested for equality. If the lengths are different then the result is known. If the lengths are the same, then bigit-wise comparisons are made starting with the most significant bigit.

Consider, for example, the $=$ operation that tests for numerical equality. If the tags of the bignums differ, then the numbers are not equal. If the tags are the same (i.e. both are positive or both are negative), then the lengths of the bignums are compared. This requires traversing both bignums. If the lengths are different, then the numbers are not equal. If the lengths are the same, then each bignum is traversed again and pairs of bigits (one from each bignum) are pushed onto the stack. After all bigits are stacked, bigit-wise comparison for equality is performed. If any pair of bigits fails to be equal then the bignums are not equal and the remaining pairs of bigits are popped off the stack. If all pairs of bigits are equal, then the bignums are equal.

4.7 The number?, equal?, and eq? Primitives

The *number*? primitive in Scheme checks to see if an argument is a number. If the argument has a positive or negative bignum tag, then *number*? returns true.

The *equal*? primitive in Scheme checks if two arguments have the same value. If both arguments to *equal*? have bignum tags, or one has a bignum tag and the other has an integer tag, then the primitive $=$ is applied to them to obtain the result. The primitive $=$ converts an argument from an integer to a bignum if necessary.

The *eq*? primitive in Scheme checks if two arguments are the same object in memory. If both arguments to *eq*? are bignums and their tags differ then the result is false. If the tags match, then the magnitudes of the least significant bigits are checked for numerical equality and the pointers to the next bigit for each bignum are checked for equality. If both test are successful the result is true. Otherwise, the result is false.

5. EMPIRICAL RESULTS

This study measured, for three benchmark programs, the number of heap allocations, the number of heap accesses, and the heap paging performance of the first-in first-out (FIFO) and the least recently used (LRU) page replacement algorithms. When a page fault occurs, the FIFO selects for replacemnet the heap page that has been in the MTEVM space for the longest period of time. In contrast, LRU selects the MTEVM resident page that has not been used for the longest amount of time.

It is generally believed, on the basis of incompletely explained empirical observations, that LRU is superior to FIFO. Baer[1] states that no analytic result is known, but that very strong experimental evidence shows that on average, for a given program, the number of faults incurred by FIFO is greater than the number of faults incurred by LRU. Belady[2] and later studies by Denning[4], showed no clear winner among several page replacement policies. There remains a need in the literature to explain why, for example, the performance of LRU is superior, or why FIFO performs, under different circumstances, poorly or as well as LRU. As recently as 1995, Wilson et. al.[22] state that even locality, the basis of modern paging algorithms, is poorly understood.

Given this, it is not surprising that the empirical record on paging algorithms is somewhat sparse, and even inconsistent. Hayes[6], for example, states that 'the page hit ratio of LRU is quite close to that of the optimal page replacement algorithm, a property that seems to be generally true.' Yet a study by Coffman and Varian[3] found that LRU yields performance within 30-40% of OPT – not close at all. This seems to be representative of the state of empirical work in the field, and, therefore, there is no clear conclusion to be drawn from the numbers reported in the literature.

5.1 Benchmarks

The memory performance of bignums in MT was studied using three benchmarks. One benchmark, *factorial*, is a purely numerical algorithm and two benchmarks, *matrix multiplication* and *insertion sorting*, are a combination of numerical and list-based algorithms. The benchmarks are described as follows:

- **Factorial**: This is the recursive version of the algorithm to compute $n!$. If $n = 0$ the function returns 1. If $n \neq 0$ the function returns the product of n and the recursively obtained result for the factorial of $n - 1$. This benchmark was used to compute 4000!. Using 10,000 as the bignum radix and for any $n > 7$, $n!$ is represented by a bignum. This means that the majority of multiplications involved bignums.
- **Matrix Multiplication**: This benchmark takes as input an integer n, creates two randomly generated square matrices, M_1 and M_2, of size n, and returns M_1 x M_2. The matrices are represented in row major form and each row is represented by a list. The resulting matrix is built one row at a time. This means that computing M_1 x M_2 requires each column of M_2 to be extracted multiple times (i.e. once for each row of M_1). After generating the matrices M_1 and M_2 the benchmark spends most of its time traversing list-based structures which represent either a row, a column, or a bignum. This benchmark was used to multiply two 15x15 matrices. Each matrix element was a bignum containing between 1 and 100 bigits.
- **Insertion Sorting**: This is the classic Computer Science sorting algorithm that takes as input a list of numbers, L, and returns a list that contains the elements of L in nondecreasing order. This algorithm works by recursively inserting the first element of L into the list obtained from sorting the rest of the elements of L. The input list L is traversed once and the list of sorted numbers being constructed is repeatedly traversed to find the proper location of each element L in the sorted list returned. The repeated traversal of the sorted list being constructed ends when this list contains all the elements of the input list. Insertion sorting was applied to a list containing 4000 randomly generated bignums. Each bignum in the list contained between 1 and 100 bigits.

Benchmark	Bignum	Heap
Matrix Multiplication	11085897	11110815
Insertion Sorting	195607	8248721
Factorial	11805615	11805629

Figure 6: Total number of bignum and heap S-expressions allocated by each benchmark program.

Benchmark	Accesses
Matrix Multiplication	24735592
Insertion Sorting	220211306
Factorial	23611262

Figure 7: The total number of heap accesses per benchmark.

5.2 Allocations and Accesses

Figure 6 displays the total number of bignum allocations and the total number of heap allocations. For matrix multiplication 11.09 million out of 11.11 million S-expressions allocated were for bignums. For insertion sorting 0.20 million S-expressions of 8.25 million S-expressions were for bignums. For factorial all but 14 of the 11.8 million allocated S-expressions were for bignums. For factorial, the 14 non-bignum S-expressions were allocated by the MTEVM to parse and evaluate its input. For matrix multiplication the difference between the total number of heap allocations and the total number of bignum allocations is due to the multiple extraction of columns, from a matrix represented as a list of rows, needed to multiply every row of the first matrix by every column of the second matrix. For insertion sorting, the overwhelming majority of heap allocations are not for bignums. This is to be expected despite the large magnitude of the bignums in the list being sorted. For insertion sorting, a new list is allocated every time a bignum is inserted into a sorted sublist of the input. Bignums are only allocated at the beginning when the list of random numbers is created. Therefore, most of the work and the allocations correspond to the list creation and not to the creation of bignums. This is a clear indicator that the use of bignums does not always lead to an explosion in heap allocations.

Figure 7 displays the total number of heap accesses made by each benchmark. For each benchmark, after a heap allocation occurs, on average, it is accessed more than once. The number of heap accesses for insertion sorting, however, explodes by a factor of 10 over the other two benchmarks. This explosion is due to the use of the relational operator $<$ when inserting a number into a sorted list. The data reveals an inherent inefficiency in the manner relational operators are implemented. In order to determine the result of applying $<$ to two bignums with the same tag and the same length, two traversals of each bignum must take place. The first set of traversals is done to determine if the bignums have the same length. The second is done to perform the

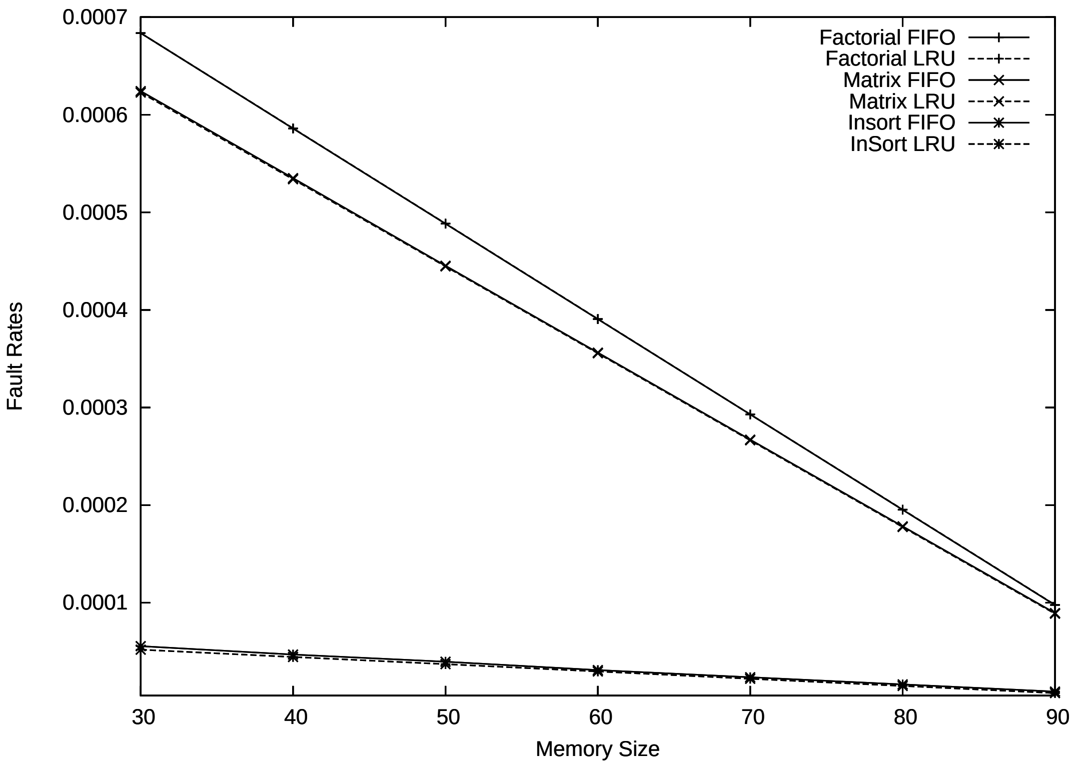

Figure 8: Paging Performance of benchmarks as memory is incremented.

bigit-wise comparisons. The data suggests that a single traversal of each bignum would be much more efficient. As the length of the bignums is being determined pairs of bignums (one from each input to $<$) can be stacked and later popped off to do the bigit-wise comparisons.

5.3 Paging Performance

A total of 42 experiments observing the heap-paging performance of FIFO and LRU were conducted. For each benchmark, the total number of heap frames (i.e. the memory size) in the MTEVM memory space was incremented in units of 10% of the total number of pages needed. Memory sizes were varied from 30% to 90% of the total number of pages needed. For each memory size, the page fault rate for FIFO and LRU were recorded. The heap page fault rate is defined as $\frac{the number of heap page faults}{the number of heap accesses}$. The results are displayed in Figure 8.

The first observation is that FIFO does not exhibit any anomalous behavior. That is, as the size of memory increases the page fault rate under FIFO decreases. Second, both FIFO and LRU are competitive paging algorithms for the MT heap. The page fault rate for all experiments was below 0.0007. That is, less than one tenth of a percent of the heap accesses caused a page fault. For insertion sorting, the page fault rate dropped below one hundredth of a percent. Third, we observe that the performance of FIFO is as good as the performance of LRU. The performance is so close that it is difficult to distinguish the differences in the graph.

To appreciate how close the performance of FIFO and LRU is, the relative difference between the two for all experiments is presented in Figure 9. The relative difference is defined as $\frac{faultrate_{FIFO} - faultrate_{LRU}}{faultrate_{FIFO}}$. If the relative difference is negative it means that FIFO performed better, fault-wise, than LRU. The converse is true if the relative difference is positive.

For factorial, FIFO did slightly better than LRU in all experiments. For matrix multiplication, LRU did slightly better than FIFO with the relative difference always being less than 0.01. For insertion sorting, the relative difference gets as high as 0.217. This seems to suggest that there is a larger performance gap between FIFO and LRU for insertion sorting than for the other two benchmarks. There is, however, no significant performance difference. The page fault rates incurred for insertion sort are all below 0.0001. The relative difference is high, because the difference between the fault rates is lower and closer to the FIFO fault rate for this benchmark. To clarify what is occurring, let us suppose that FIFO incurs 2 page faults and that LRU incurs 1 page fault. The relative difference in this case is 0.5, but no one could convincingly argue that there is a significant performance gap between FIFO and LRU.

FIFO is highly competitive due to the data structures used to represent a bignum. Consider the creation of a bignum. The bigits are stacked and then allocated into an immutable list-based structure with S-expressions allocated in consecutive addresses in the MT heap. If the bignum occupies several pages, these pages are traversed one after another.

Benchmark	30%	40%	50%	60%	70%	80%	90%
Factorial	-.00006195	-.00014454	-.00026015	-.00043351	-.00028906	-.00065005	-.00216357
Matrix Multiplication	.00246721	.00204468	.00136388	.00170551	.00227514	.00341219	.00682439
Insertion Sorting	.06877291	.05491594	.0666993	.04435607	.06512862	.10515021	.21659549

Figure 9: The relative difference between FIFO and LRU for each benchmark and memory size.

Both FIFO and LRU incur the same number of page faults for such a linear traversal of pages, because the least recently used page is also the page that has been in memory for the longest period of time. Also consider the traversal of a bignum, for example, when $<$ is evaluated. Since the S-expressions of a bignum are allocated in consecutive addresses in the MT heap, a bignum traversal is a traversal of pages if the bignum occupies more than one page for which the performance of FIFO and LRU are the same. Finally, consider the process of creating a bignum and then traversing it. This occurs, for example, in factorial when a bignum is created for $(n-1)!$ and then traversed to multiply it by n. For the creation of the bignum for $(n-1)!$ FIFO performs as well as LRU. This creation leaves the least significant bigits of $(n-1)!$ in memory. These bigits are traversed without faults by the multiplication by n and then FIFO and LRU both fault on the same pages that hold the bigits that are not in memory. These observations clearly explain why the performance of FIFO and LRU is so close.

The results obtained are consistent with past empirical data that suggests that FIFO is as good as LRU as a page replacement algorithm for the MT heap[13, 15, 14]. This is significant, because $FIFO$ can be implemented in software much more easily and efficiently than LRU. FIFO does not incur the per access overhead associated with LRU and, therefore, makes it a more attractive candidate for a software implementation that assumes no hardware support for paging.

6. CONCLUSIONS AND FUTURE WORK

This article presents empirical data on the memory performance of programs that utilize bignums represented as immutable lists. The data suggests that for numerical computations, as expected, the majority of heap allocations are due to bignums and that for non-numerical computations (e.g. insertion sorting) the majority of heap allocations are not due to bignums. This means that the use of bignums does not imply an explosion in the number of heap allocations.

Most of the literature on bignums focuses on the development of arithmetic algorithms and there is a notable absence of discussion on the design of algorithms for relational operators. Our data clearly suggests, that it is also very important to carefully design algorithms to implement relational operators. The number of heap accesses can explode if relational operators are not carefully designed.

The data also suggests that FIFO is as good as LRU as a page replacement algorithm for the MT heap when programs that utilize bignums are evaluated. This follows from how the data structure used to store bignums is created and accessed. The creation and traversal of bignums forces both FIFO and LRU to fault on the same pages. Furthermore, when the creation of a bignum is followed by a traversal of that bignum, FIFO and LRU fault on the same pages.

Our future work includes expanding our set of benchmarks to include programs that utilize division, modulus, and more relational operators. We are also working on improving the implementation of relational operators to reduce the number of heap accesses and on implementing more multiplication and division algorithms including grade-school style division, divide and conquer division, and Karatsuba-like multiplication.

Future work also includes developing support for bignums based on immutable arrays instead of immutable lists and to compare its memory performance with the list-based implementation. The initial array-based implementation will utilize the same algorithms for arithmetic and relational operators as the list-based implementation so that a direct comparison can be made. The array-based implementation will use 1 S-expression as a header to store the sign, the number of bigits, and the virtual memory address of the first S-expression of the array representing a bignum in the heap. The header will never reside in the heap, and each S-expression in the heap will be able to store 2 bigits. We expect this implementation to result in fewer heap allocations as well as fewer heap and stack access due to less memory consumption. Furthermore, we expect the number of page faults and the page fault rates to go down making the performance gap between FIFO and LRU even tighter.

7. ACKNOWLEDGEMENTS

The authors would like to thank the Department of Math and Computer Science of Seton Hall University for their support and encouragement in the development of this study. We would also like to thank our colleagues Ms. Barbara Mucha for her development of the MT-Scheme parser and evaluator, and Mr. Victor Encarnacion and Ms. Kristine Joy S. Apon for their development of the MT-Scheme compiler.

8. REFERENCES

[1] Jean-Loup Baer. Computer Systems Architecture. Computer Science Press, 1980.

[2] L.A. Belady. A Study of Replacement Algorithms for a Virtual-storage Computer. IBM Systems Journal, 5:78–101, 1966.

[3] E.G. Coffman and L.C. Varian. Further Experimental Data on the Behavior of Programs in a Paging Environment. Communications of the ACM, 11:471–474, 1968.

[4] Peter Denning. Working Sets Past and Present. IEEE Transactions on Software Engineering, SE-6:64–84, 1980.

[5] Victor Encarnacion, Patrick DeSomma, and Marco T. Morazán. Paging Performance in the MT Evaluator Virtual Machine. In Marco T. Morazán, editor, Proceedings of the 10th Annual Mid-Atlantic Student Workshop on Programming Languages and Systems, pages 9.1–9.9, 2004.

[6] Hayes. Computer Architecture and Organization. McGraw-Hill, 1988.

[7] A. Karatsuba and Y. Ofman. Multiplication of Multidigit Numbers on Automata. Soviet Phys. Dokl., 7:595–596, 1963.

[8] T. Lindholm and F. Yellin. The Java Virtual Machine Specification. Addison-Wesley, Reading, MA, 1996.

[9] Mads Tofte. A Brief Introduction to Regions. In Proc. of the 1998 ACM Int. Symp. on Memory Management, pages 186–195. ACM Press, 1998.

[10] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. Communications of the ACM, 3(4):184–195, 1960.

[11] Mersenne.org. Mersenne.org Project Discovers New Largest Known Prime Number, 225,964,951-1. http://www.mersenne.org/25964951.htm, February 2005.

[12] Marco T. Morazán and Douglas R. Troeger. The MT Architecture and Allocation Algorithm. In Greg Michaelson, Phil Trinder, and Hans-Wolfgang Loidl, editors, Trends in Functional Programming, volume 1, pages 97–104, Bristol, UK, 2000. Intellect.

[13] Marco T. Morazán and Douglas R. Troeger. A Case Study of List-Memory Paging in a Distributed Memory System for Functional Languages. In Martin A. Musicante and E. Hermann Haeusler, editors, Proc. of The 5^{th} Brazilian Symposium on Programming Languages, pages 80–95, Curitiba, Brasil, 2001. Universidade Federal do Paraná.

[14] Marco T. Morazán and Douglas R. Troeger. List-Heap Paging in a Distributed Virtual Memory System for Functional Languages. In Hamid Arabnia, editor, Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, pages 1689–1695. CSREA Press, 2003.

[15] Marco T. Morazán, Douglas R. Troeger, and Myles Nash. Paging in a Distributed Virtual Memory. In Kevin Hammond and Sharon Curtis, editors, Trends in Functional Programming, volume 3, pages 75–86, Bristol, UK, 2002. Intellect.

[16] Marco T. Morazán, Douglas R. Troeger, and Myles Nash. The MT Stack: Paging Algorithm and Performance in a Distributed Virtual Memory System. CLEI Journal, 5(1), 2002.

[17] Tim O'Reilly. Das BSD-Unix-Nutshell-Buch. (German) [The BSD UNIX Nutshell Book]. Addison-Wesley, Reading, MA, USA, 1990.

[18] Kevin Ryde. GNU MP - Divide and Conquer Division. http://www.swox.com/gmp/manual/Divide-and-Conquer-Division.html#Divide%20and%20Conquer%20Division, 2004.

[19] Kevin Ryde. The GNU MP Bignum Library. http://swox.com/gmp/manual/Assembler-Carry-Propagation.html, April 2005.

[20] S. Breitinger, U. Klusik, R. Loogen, Y. Ortega-Mallón, and R. Peña. DREAM: The Distributed Eden Abstract Machine. In Chris Clack, Kevin Hammond, and Antony J. T. Davie, editors, IFL, volume 1467 of Lecture Notes in Computer Science, pages 250–269. Springer-Verlag, 1998.

[21] Unknown?? GMPbench 0.1 results for GMP 4.1.4. http://swox.com/gmp/gmpbench.html, April 2005.

[22] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic Storage Allocation: A Survey and Critical Review . Proc. of the 1995 International Workshop on Memory Management, Springer-Verlag LNCS, 1995.