Glibc 6 Function Optimization Report

John M. Zulauf

Sr. MTS Embedded Core Software

AMD – Longmont Design Center

June 30, 2006

Version 1.0

Introduction

This document presents the goals, methodology, approaches and results of the "6 Function" Optimization project completed June 2006.

Goals

The goals for this project where:

1. Create optimizations for 6 functions from glibc
2. Optimization targeted for Geode LX800
3. Achieve at least 20% improvement for 3 of the 6 functions.
4. Performance increase measured relative to the Gentoo Validation Image (486 target)
5. Develop deep understanding of optimization for the Geode LX and document the lessons learned.

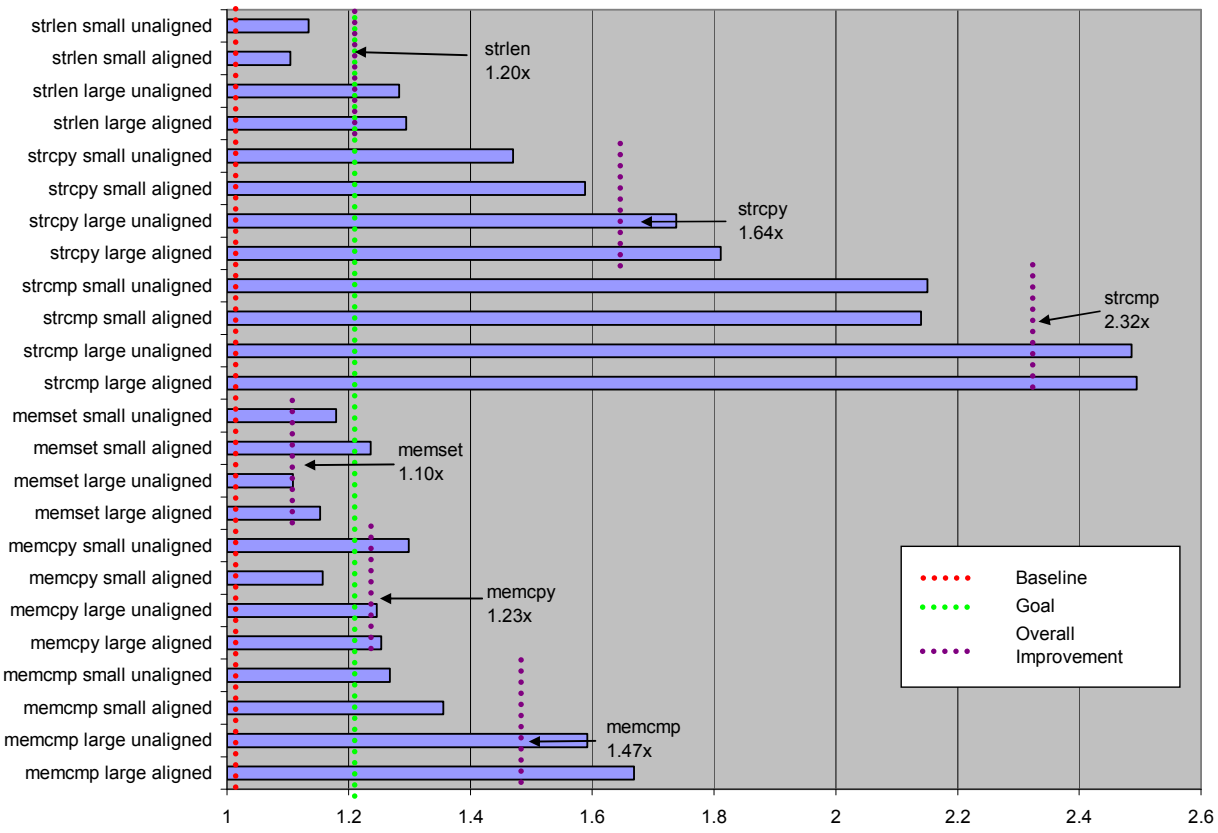
The six functions targeted were:

1. memcmp
2. memcpy
3. memset
4. strcmp
5. strcpy
6. strlen

These functions were chosen as they are both broadly and commonly used, and are representative, canonical forms of broadly useful algorithmic element (copy, set, compare, examine). However, no profiling was done to measure the frequency of use, nor the population distribution of operation size or operand alignment.

The overall results are given in the following table and chart.

Function	Improvement (overall)	Meets Goal
Memcmp	1.47	Yes
Memcpy	1.24	Yes
Memset	1.10	No
Strcmp	2.32	Yes
Strcpy	1.65	Yes
Strlen	1.20	Yes



Methodology

The methodology used was designed to balance the desire to measure each function in isolation, run these functions in a working system (not simulation), compare a number of algorithmic approaches, and easily represent the data for clear communication. As such a test framework was constructed with the following components:

- "test_" Test framework for performance testing
- "loop_" Standard loops for testing and reporting
- "stats_" Simple Approximate Statistics Package
- "opts_" Callback based command line options handler
- "deck_" Deck of cards API

Each of these components is documented in Doxygen. Collectively they underly the unit test (utest) and performance test (ptest) for each glibc function optimized. Some important details relative to these components is given below.

This framework, and the optimized glibc functions are checked into the Longmont Perforce server – ldcperforce.amd.com at path //drivers/projects/geodelx/linux/glibc/candidates.

The directory hierarchy is organized:

common	The test framework
permute	Experimental versions of candidate functions using permutation to find optimal instruction ordering
unit_test	Test functions for components of the test framework
memcmp, memcpy, memset, strcmp, strcpy, strlen	The candidate function implementations include unit and performance test applications

Note that with the discussion we will refer to filenames within the respective directory. Also note that while the function names of the candidate optimizations had some meaning to the author (for example those with iX86 are glibc sysdeps/i386/iX86 derived versions), the reasoning behind that naming has largely escaped even the author in most cases. The function names suffice to uniquely identify the functions under test, but further meaning should not be imputed.

Buffer Management

On of the underlying assumptions for the performance optimization was that these measures would be most accurate and interesting when both source and destination data (if any) where not found in the cache. Additionally, the dynamic branch prediction of the Geode LX processor made it imperative that randomization of the jump conditions be supported. The buffer management of the "test_" component supported ensuring the non-cached nature of the buffers, and random variation in the buffer size and alignment. The "deck_" component was used to allow guaranteed distributions, such that a randomization of a set of buffer sizes (or alignments) would exercise the population exhaustively, before any member of the population repeated (as does a shuffled deck of cards). A slightly better randomization could have been achieved with a "multi-deck shoe" approach, but seemed excessive, given the goal of simply preventing "conditioning" the dynamic prediction for a given size or alignment.

Statistics Gathering

Early in the development of the framework it became clear, that even at highest priority performance test applications will be interrupted on a regular basis by other system loads. Examples of those loads are

1. HZ interrupt – Linux kernel “heartbeat” by which the kernel manages timed or “soft” interrupts, and time sharing
2. HW refresh tasks – the video and DRAM refresh cycles
3. SMI/NMI – VSA/background event to support HW virtualization for LX/5536

To allow for the timing of individual functions calls lasting as few as 50 instructions, a statistical approach was taken. However, the broad variation in timing possible given interrupt handling, task switching, and SMI's the data gathering had a broad dynamic range. A novel approach was taken, allowing broad dynamic range with constant relative precision. Effectively the data was first indexed by the msb, then by a configurable number of bit following the msb. This two stage hashing function allow the “buckets” into which the data was sorted the need characteristics. Values were additively accumulated into the buckets and an average for each bucket computed. The most populous bucket (i.e. the modes) were then population weighted and average, with mode representing < 1/256 of the total samples discarded. This approach, for large enough sample sizes, resulted in repeatable comparable measures between algorithms and from test run to test run.

Function Measurement

The timing measurement of the function was further complicated by the nature of the rdtsc instruction on the Geode LX processor. The “serializing” nature of the RDTSC instruction, varies greatly from x86 implementation to implementation. On the LX, RDTSC is “mostly serializing” -- in that in waits until all pending data transaction through the DM block complete. This lead to misleading results where a function's performance timing would include the cost of speculative (but unused) PREFETCH instruction. Thus, the following algorithm was used.

read starting timestamp counter
execute 100 noop instructions
read ending timestamp counter
record difference of ending and starting in “empty loop” statistics record
read starting timestamp counter
execute function under test
execute 100 noop instructions
read ending timestamp counter
record difference of ending and starting in “timed function” statistics record

The function time was then computed as the difference of the "empty loop" time and the "timed function" time.

By inserting the NOP instructions, any pending memory transactions would be hidden, hiding the serializing nature of the RDTSC instruction.

Design Targets

Each of the optimizations faced a common issue – optimization for larger buffer sizes decreased the performance on trivial (1-16 bytes) buffer sizes. Effectively, the setup phase adds clocks to discriminate small-large path. From examination of optimizations written for other architectures within Glibc, this seems to be an acceptable design trade-off.The balance chosen was to pick an average of 64 bytes as a nominal "small" size and 1024 bytes as a nominal "large" size. Given the lack of profiling, these numbers are arbitrary, but intuitively reasonable for most of the functions

Results Reporting

The framework supports flexible test and reporting options, suitable for both development and final reporting. Tabular and summary output in plain text, html, and csv allowed for flexible reporting. Final output was generated in .csv format and imported into Microsoft Excel.

For each function we will report the Trivial (<4), Small (average 64), and Large (average 1024) cases, and show a typical graph for the "Small" case. The numbers represent the relative performance ($cycles_{ref}/cycles_{opt}$)[1]. The Overall score is the average of the Small and Large scores for both aligned and unaligned buffers.

Approach

In order to optimize these functions a common approach was taken for each.

1. Identify extant implementations from glibc, BSD or other publically available sources

[1] Pg 25, "Computer Architecture – A Quantitative Approach, Third Edition" Hennessy and Patterson, Morgan / Kaufmann ,1990

2. Implement and test a ptest (performance test) and utest (unit test) application using the framework components customized to the specifics of the function under test.
3. Test the initial candidate and identify potential strategies to optimize.
4. Target the 0-128 and 0-2049 size ranges, testing both aligned and randomly unaligned data.
5. Iterative improve either the most promising of the initial candidates, and/or develop a "from scratch" or hybrid approach based on lessons learned
6. Continue to test and improve, including input from peer reviews – with final polishing on "short" buffer length cases.

Each of the functions had common elements:

1. Setup phase – compute alignment, select small-path(s) vs. large-path(s)
2. Inner loop – the core algorithm, assuming size and/or alignment constraints have been satisfied
3. Clean-up/Remainder – complete the operation. Usually, either a size remainder, or other byte accurate operation the left by the exit criteria of the inner loop.

A common set of optimizations were useful

1. Avoid mis-predicted conditional jump instructions
2. Use PREFETCH to avoid mov stalls
3. Unroll loops to support PREFETCH, and minimize loop and increment overhead
4. Convert from byte sized to double-word size operations as possible
5. Align source or destination addresses
6. Simplify function preamble and return clauses – avoiding registers protected by the ABI
7. Hide CLD EX stall with MOV AC stall
8. Avoid multiple short REP
9. Use src + (dst-src) single pointer addressing tricks
10. Use base and offset in addressing modes to minimize pointer arithmetic.
11. Balance complexity of "setup" stage with performance gains in inner loop stage

For each function we also list potential areas for further investigation, however, none of these should be pursued until profiling and usage data have been compiled.

Function: memcmp

```
int memcmp(const void *s1, const void *s2, size_t n);
```

> The **memcmp**() function compares the first n bytes of the memory areas s1 and s2. It returns an integer less than, equal to, or greater than zero if s1 is found, respectively, to be less than, to match, or be greater than s2.

Approach

The memcmp implementation "lxd_memcmp.S" uses the "use dword operations instead of byte-operations". For n > 3, the first first dword values are compared using a REP; CMPSL, and depending on the remainder of n/4, and the presence of a data mismatch, the appropriate bytes are compared using a REP;CMPSB.

Pseudocode for lxd_memcmp.S

Push esi
Store edi in edx
Store s1 in esi
Store s2 in edi
Store n in ecx
Clear the direction flag
Store 0 in eax (with xor)
Set repeat count to number of whole dwords (Shift ecx right 2)
Using repe;cmpsl to compare the dwords
If the dwords compare equal goto to "Do remainder"
Adjust esi, edi, to point to the last word checked (-4)
Set repeat count (ecx) to 4
Goto "Compare bytes"
Label: "Do remainder"
Store n into ecx
And ecx with 3 to set repeat to the number of remainder bytes
Label: "Compare bytes"
Using repe;compsb, compare the bytes (remainder or mismatched dword)
If bytes compare (zero flag is set) Goto "Exit"
Set ecx to 1
Using conditional move set eax to ecx if "above"
Set ecx to -1
Using conditional move set eax to ecx if "below"
Label: "Exit"
Pop esi
Restore edi from edx

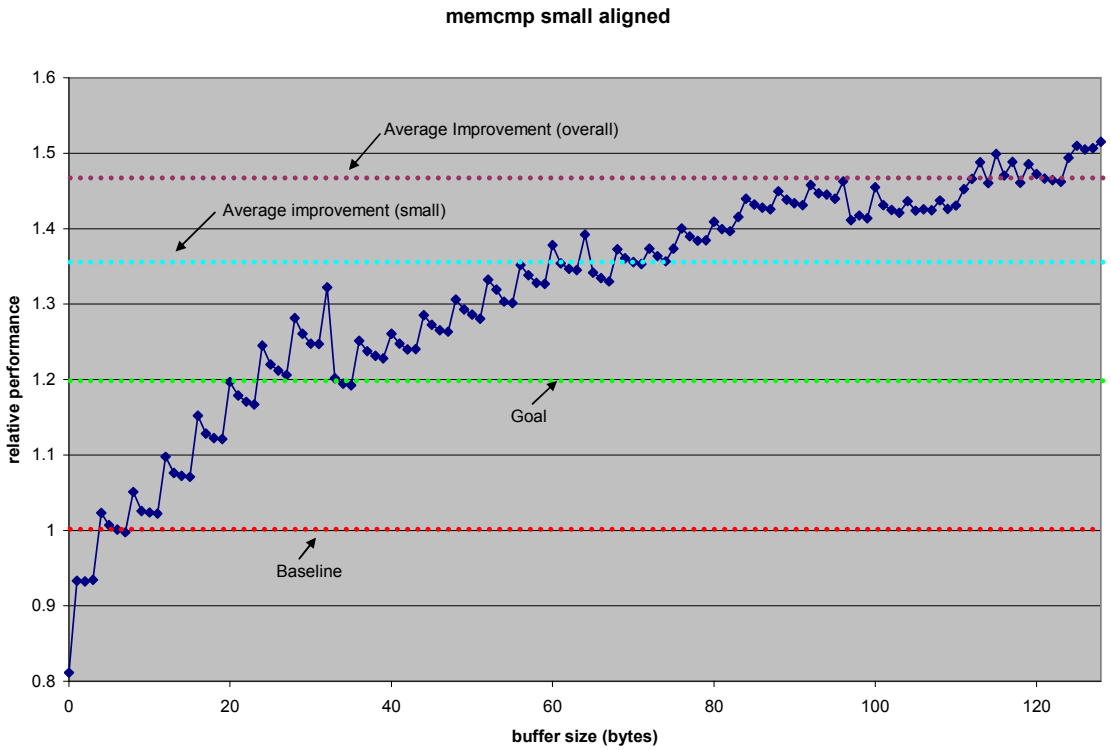
Derivation

The general derivation of the routine is from “i386_memcmp.S” -- licensed under the LPGL, but the dword-then-byte-logic, and the return values computation is original with the author and AMD.

Results

For memcmp operations of < 5 bytes, lxd_memcmp is slower than the reference, by as much a 20% for the zero length case. However, the overall performance substantially exceeds the project goals.

Length	Aligned	Unaligned
Trivial	0.92[2]	0.91
Small	1.36	1.27
Large	1.66	1.59
Overall	1.47	



[2] Note that values < 1 indicate a performance reduction. These are common in the Trivial case, both for these optimizations and those extant in the glibc codebase.

Future Work

Areas not explored for potential further optimization are:

- unroll loop into 32 or 64 byte REP's and to allow the use of PREFETCH
- MMX comparison operations
- byte-wise only search for < 7 bytes (to improve "Trivial" case)
- Review return value computation

memcpy

```
void *memcpy(void *dest, const void *src, size_t n);

     The memcpy() function copies n bytes from memory area
     src to memory area dest. The memory areas may not
     overlap. Use memmove(3) if the memory areas do
     overlap.
```

Approach

Based on the results of the RedCapFX project, memcpy optimization was approach as an unrolled, prefetch loop. The optimum unroll size was measured as 64 bytes (2 cache lines). Initially, REP;MOVSL was used for the inner loop, but the final optimization "lx_memcpy_movq_r.s" uses the MMX MOVQ instruction to do quad-word movement. In addition, the "Trivial" case performance was aided by adding an additional test for < 16 bytes.

The optimization of memcpy is unique in that the function preamble and setup phase were optimized by using an exhaustive search for the best order of lines 11-16. Over 600 possibilities were tested, and the best selected. The resulting gain was 2-4 cycles over the original version.

Pseudocode for lx_memcpy_movq_r.s

Store edi in edx
Store dest in edi
Clear the direction flag
Push esi onto the stack
Store n in ecx
Store src in esi
Compare ecx with 15
If ecx <= 15 Goto "Copy bytes"
Prefetch next cacheline after src (+32)
Store ecx in eax

Shift eax right 6 (eax is the number of 64 byte blocks)
If eax is 0 Goto "Copy Remainder"
Prefetch next +2 cacheline after src (+64)
Label: 64 byte loop
Store esi+0,8,16,24 into mm0-3 respectively
Prefetch next +3 source buffer cacheline (esi+96) after movq->mm0
Store mm0-3 to edi+0,8,16,24 respectively

Store esi+32, 40, 48, 56 into mm4-7 respectively
Prefetch next +4 source buffer cacheline (esi+128) after movq->mm4
Store mm4-7 to edi+032, 40, 48, 56 respectively
Increment esi and edi by 64
Decrement the loop counter (eax)
If eax > 0 goto "64 byte loop"
End mmx mode (emms)

Compute remainder by Anding 63 with ecx
If ecx is 0 goto "Common Return"
Label: Copy Remainder
Save remainder cl to al
Compute dwords in remainder by shifting ecx right 2
Copy remainder dwords using rep; movsl
Restore remainder copying al to cl
Compute remainder bytes by anding cl with 3
Label: Copy Bytes
Copy the remainder bytes using rep; movsb
Label: Common Return
Pop esi from stack
Restore edi from edx
Store dest in eax (from stack)

Derivation

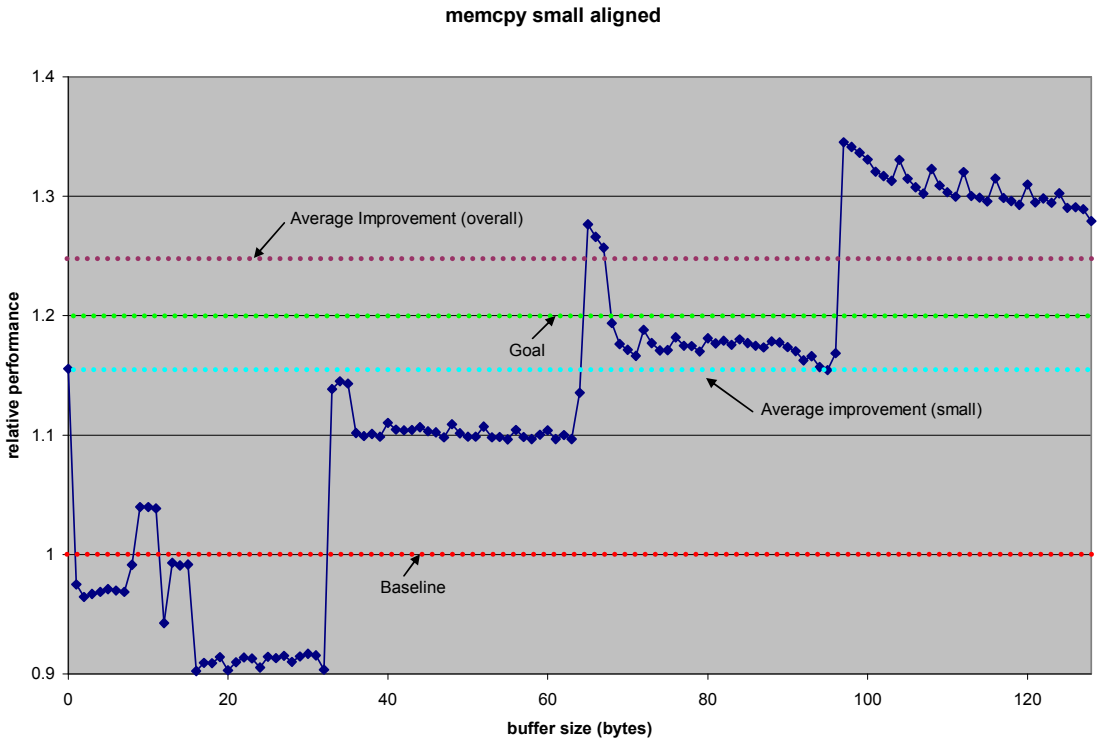
The entire implementation of lx_memcpy_movq_r.s is original with the author.

Results

While the Trivial Case (<4 bytes) here performs as well as the reference implementation, clearly there is a performance issue in the 16 – 32 byte range. Overall we see about 20% improvement, with peak throughput close to the fastest rates observed in peak rate simulation testing.

Length	Aligned	Unaligned
Trivial	0.99	1.02
Small	1.16	1.30

Large	1.25	1.25
Overall	1.23	



Future Work

Clearly the performance at less than one cacheline needs to be addresses. However, before additional effort is expended profiling of usage should be obtained.

Memset

```
void *memset(void *s, int c, size_t n);

      The  memset()  function  fills  the  first  n bytes of
      the memory area pointed to by s with the constant byte
      c.
```

Approach

With the exception of the extremely slow i586 implementation, each of the glibc implementations use a combination of byte and dword sized REP;STOS. The pseudo-code is:

Repeat Store String until Destination is Dword aligned
Repeat Store String of the Dwords
Repeat Store String of the remaining bytes

Not surprisingly this limits the optimization potential. Dword sized operations are already being used, alignment sensitivity is being accounted for and PREFETCH doesn't help memset performance (or is considered unacceptable cache pollution). The avenues for attack are better alignment logic, and better "bulk write". With regard to these, the memset function is highly sensitive to the domain, as regards the fastest algorithm. For small buffer sizes, and dword-aligned destination addresses, the simplest rep;stosl (dword); rep;stosb (byte) implementation (lx_memset_simp.S), is more than 20% *better* than the baseline. However, for unaligned, large buffers, this same implementation is 14% *worse* than baseline. Thus, the best overall improvement was derived from optimizing for each domain, and finding an effective hybrid of the locally optimized solution. Confusing this issue further, was the fact that the "hand-off" criteria were in the same sections of the domain as the small/large boundaries, causing measurement anomalies w.r.t. the domain selection logic.

The overall optimized solution combines an optimized alignment logic and a domain split at 512byte selecting between a REP;STOS solution and an MMX based solution. For n > 511, using MOVQ becomes substantially faster than using REP;STOS, while for smaller set operations, the opposite is true. As the discriminant is outside the "small" domain, the selection overhead is under estimated in the "small" test case, by 7% of the total cycles. Thus in a mix of large and small cases, the small domain performance would only be 1.05 better than baseline.

The optimization logic uses cmov to combine the alignment and size logic into a single conditional jump. For small enough values of "n", the traversing the alignment logic slows overall performance. In order to trigger the alignment logic, the destination address is masked against 7 (for quad-word alignment). However, if the parameter n < 64, the cmov instruction sets the mask to zero before it is used. Thus the alignment logic is only walked for n > 63 and unaligned target addresses.

Psuedocode for lx_memset_mmx_h2.S

Note: this preamble has not been optimized
Clear direction flag
Push edi onto the stack
Store s (destination pointer) in edi
Store n (the fill count) in edx
Note: This is the single test/jmp "should we align" test
Store the alignment mask (7) in ecx
Copy n from edx to eax
Shift eax right 6
Using cmov, set ecx to eax (clear to zero) if eax is zero (i.e. n < 64)
Store c (the byte to fill) into eax, with zero fill high

Fill c into each byte of eax by multiply by 0x01010101
And edi (the dest address) with ecx (as the dest argument)
If ecx is zero (ie. Aligned or n < 64) Goto "Skip Alignment"
Note: This is the alignment code
Set ecx to 8 – ecx (neg, add) – number of bytes to the next boundary
Subtract the ecx from edx (to account for the alignment bytes)
Set the alignment bytes with rep;stosb (the value in eax, the count in ecx)
Label: Skip Alignment
Note: edx has the "remaining bytes to set" regardless of path to this point
Compare edx to 511
If below or equal 511 Goto "Set Remainder"
Copy edx (bytes remaining to copy) to ecx
Shift ecx right 6 (divide by 64)
And 63 with edx
Note: ecx has the number of 64 byte block and edx has the remainder
Store %eax in the low dword of mm0
"Unpack" (PUNPCKLDQ) mm0 to mm0 (filling mm0 with "c")

Note: The following is the MMX unrolled 64 byte movq memset
Label: Movq 64 Bytes
Store mm0 to edi+(0,8,16,24,32,40,48,56)
Increment edi by 64
Using "loop", decrement ecx and Goto "Movq 64 bytes" if > 0

Clear mmx mode using emms

Note: The following is the basic approach from all but the i586 example
Label: Set Remainder
Copy edx to ecx
Shift ecx right 2 (divide by 4), ecx now has the number of dwords to set
And 3 with edx, edx now has the remaining bytes to set
Repeat Store String Long (dwords, ecx has the count eax the value)
Copy edx to ecx
Repeat Store String Byte (bytes, same note as above)

Store s (from the stack) into eax
Pop edi

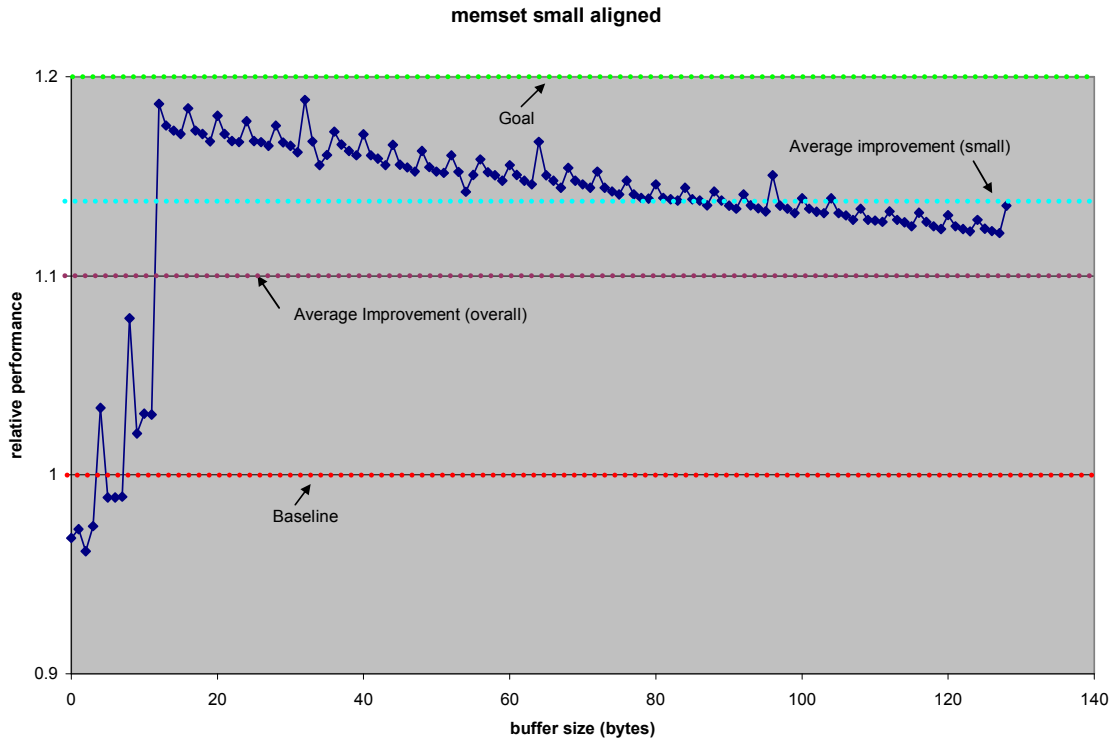
Derivation

Lx_memset_mmx_h2 is derived from LGPL licensed glibc/sysdeps/i386/i686/memset.S, though only the preamble dword-then-byte, and return sections are retained from the original. The alignment logic and MMX unroll are original with the author.

Results

The results reflect the compromise of the final design. While not as fast as any of the targeted implementations, the final does no worse than the baseline for small memset, and is within 1% of the best performing targeted implementations for large, aligned copies.

Length	Aligned	Unaligned
Trivial	0.97	0.89
Small	1.14	1.01
Large	1.14	1.11
Overall	1.10	



Future Work

Given the compromise design of the memset final candidate, clearly tuning can be done to the function once the actual domain and distribution are better know. Additionally, the "hand off" from MMX to non-MMX paths could be further refined, assuming such design is still meaningful once the profiling and domain definition are complete.

strcmp

```
int strcmp(const char *s1, const char *s2);
```

```
The strcmp() function compares the two strings s1 and
s2.  It returns an integer less than, equal to, or
greater than zero if s1 is found, respectively, to be
less than, to match, or be greater than s2.
```

Approach

Strcmp is a particularly difficult function to optimize. As it has two "source" pointers, a common dword alignment cannot be guaranteed. Additionally, either string can terminate at a give byte offset. While a partial dword (using the alignment of one string was attempted, no approach proved better than a character-by-character implementation.

The basis for the optimization was the i686/strcmp.S (as this did a simple character-by-character comparison). However the final form i686_strcmp_r2.S has no algorithmic elements remaining from the original except the stack fetch and the return value computation.

This initial optimization (i686_r1i.S), used the base pointer + register offset trick p2 = p1 + (p2-p1), where (p2-p1) is the offset to eliminate the doubly incremented pointers of the original i686 implementation. This simple trick saved 10% performance on the Geode LX, but slowed a test Athlon system by 10%. In addition to the single pointer, the inner byte-wise loop was replaced with a 16 byte unroll, allowing for addition of PREFETCH for both source pointers. Each byte still requires two conditional jump, but all should be strong predicted as not taken, causing only single misprediction when the end of string or byte mismatch is found – the minimum possible number.

Psuedocode for i686_strcmp_r2.S

Note: this preamble has not been optimized
Store s1 in ecx
Store s2 in edx
*Store s1-s2 in ecx (**offset** of s1 from s2)*
Label: Unroll loop
Store byte at edx+ecx in al
Compare byte at edx to al
If the bytes are not equal Goto "Not Equal"
Test al with al (flags are set as if and al, al)
If al is zero, Goto "Al is zero" (as s1 equals s2 and is ended)

Store byte at 1+edx+ecx in al
Prefetch 32+edx+ecx
Compare byte at 1+edx to al
If the bytes are not equal Goto "Not Equal"
Test al with al (flags are set as if and al, al)
If al is zero, Goto "Al is zero" (as s1 equals s2 and is ended)

Store byte at 2+edx+ecx in al

Prefetch 32+edx
Compare byte at 2+edx to al
If the bytes are not equal Goto "Not Equal"
Test al with al (flags are set as if and al, al)
If al is zero, Goto "Al is zero" (as s1 equals s2 and is ended)

Unroll: N in range 3 – 15, inclusive
Store byte at N+edx+ecx in al
Compare byte at N+edx to al
If the bytes are not equal Goto "Not Equal"
Test al with al (flags are set as if and al, al)
If al is zero, Goto "Al is zero" (as s1 equals s2 and is ended)
End Unroll
Increment edx by 16 (implementation uses leal)
Goto "Unroll Loop"

Label: Al Is Zero
Set eax to 0 (xorl works)
Return

Label: Not Equal
Note: Flag values still reflect the Compare prior to the jne above
Store 1 in eax
Store -1 in ecx
Conditionally store ecx in eax if "below"

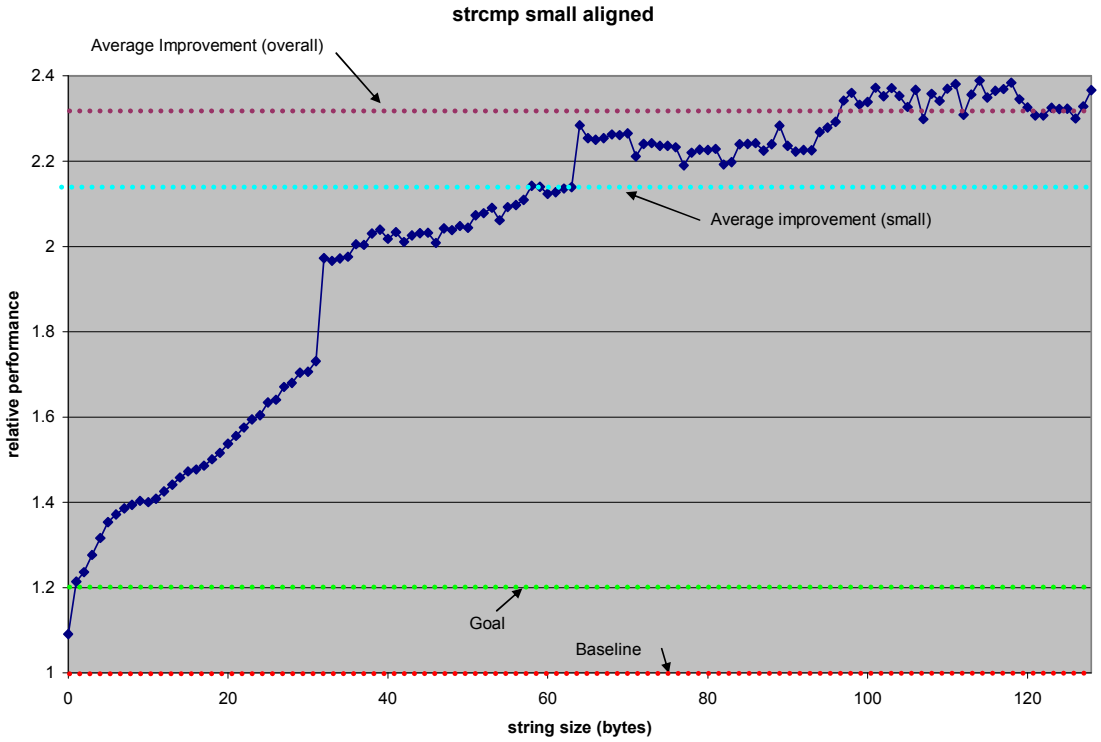
Derivation

The optimized routine is derived from sysdeps/i386/i686/strcmp.S, which is licensed un the GPL. However the core of the implementation, lines 51-159 are original with the author.

Results

The combination of single increment indexing, loop unroll and single index increment, and prefetching of both source buffers yields a strong performance increase of up to 2.5 times, with no adverse trivial range effects.

Length	Aligned	Unaligned
Trivial	1.21	1.21
Small	2.14	2.15
Large	2.49	2.49
Overall	2.31	



Future Work

Given the performance improvement, and the unlikelihood of strcmp being on the inner loop of an application, this function likely requires no future refinement.

strcpy

```
char *strcpy(char *dest, const char *src);
```

> The strcpy() function copies the string pointed to by src (including the terminating '\0' character) to the array pointed to by dest. The strings may not overlap, and the destination string dest must be large enough to receive the copy.

Approach

The optimization of strcpy includes a variety of elements. In rough terms the code breaks done into the following sections

Section	Description	Optimizations
Initialization	Save state, get arguments	eliminated two push operations

Section	Description	Optimizations
Alignment	compute number of alignment block don't need to be done jump the unneed ones, execute 0-3 alignment blocks	eliminated pointer increment from alignment block. (original concept from i586 version, rewritten to eliminate increment and cope with codesize impact of unroll.
Dword aligned copy	check for zero byte in dword, copy if no zero	Unrolled to 4 dword (half cacheline) loop, utilize more efficient "magic number" algorithm, added prefetch to unrolled loop
Final Dword handling	Find the actual zero byte, copy to that point	used constant offsets to avoid pointer increment and movs performance hits.
Exit	Set return, restore state	Two fewer pop's

Psuedocode for lx_strcpy_r3.S

Push esi onto the stack
Store edi in edx
Clear the direction flag
Store src in esi
Note these two step compute the number of alignment byte-by-byte blocks to skip
Load esi -1 in ecx
And 3 with ecx
Set eax to 0 (the alignment block will need this)
Add 3 to esi (this is part of finding the next aligned address below)
Store dest in edi
And NOT(3) with esi (this is the "next" aligned address or src if aligned)
*Set ecx to Label "Alignment Blocks" + ecx * 8 (the size of the alignment block)*
Goto the address in ecx
Note: the following Label is quad word aligned, though I don't know if it matters
Label: Alignment Blocks
Unroll: N in the set (-3, -2, -1)
Or byte at (esi + N) with al (result in al)
Store String Byte (always, even for terminating zero)
If "Or" set Zero Flag, Goto "Landing" (see Note below)
Xorl eax with eax
End Unroll
Note: The pair of Goto's are needed to keep the alignment blocks the same size, as the blocks are straddling the 128 byte bound to "end" from the block. "Landing", is a near target s.t. the offset can be 1 byte
Goto "Unroll Loop"
Goto "End"
Label: Unroll Loop

Unroll: For N=(0,1,2,3)
Load Dword at esi into eax
If N is 3 prefetch esi+32
Load magic (0xfefefeff) + eax into ecx
Not eax
And eax with ecx, store in ecx
Test 0x80808080 with ecx
If not zero, Goto "Do Last Dword"
Move string Long (Dword) (dword at esi is copied to edi, increment esi and edi)
End Unroll
Goto "Unroll loop"

Label: Do Last Dword
Unroll: For N = (0,1,2,3)
Load byte at N+esi into al
Test 0xFF with al (or al with al)
Store al into N+edi
If N < 3 If Test is zero Goto "End"

Label: End
Load dest from stack into eax
Pop esi
Restore edi from edx

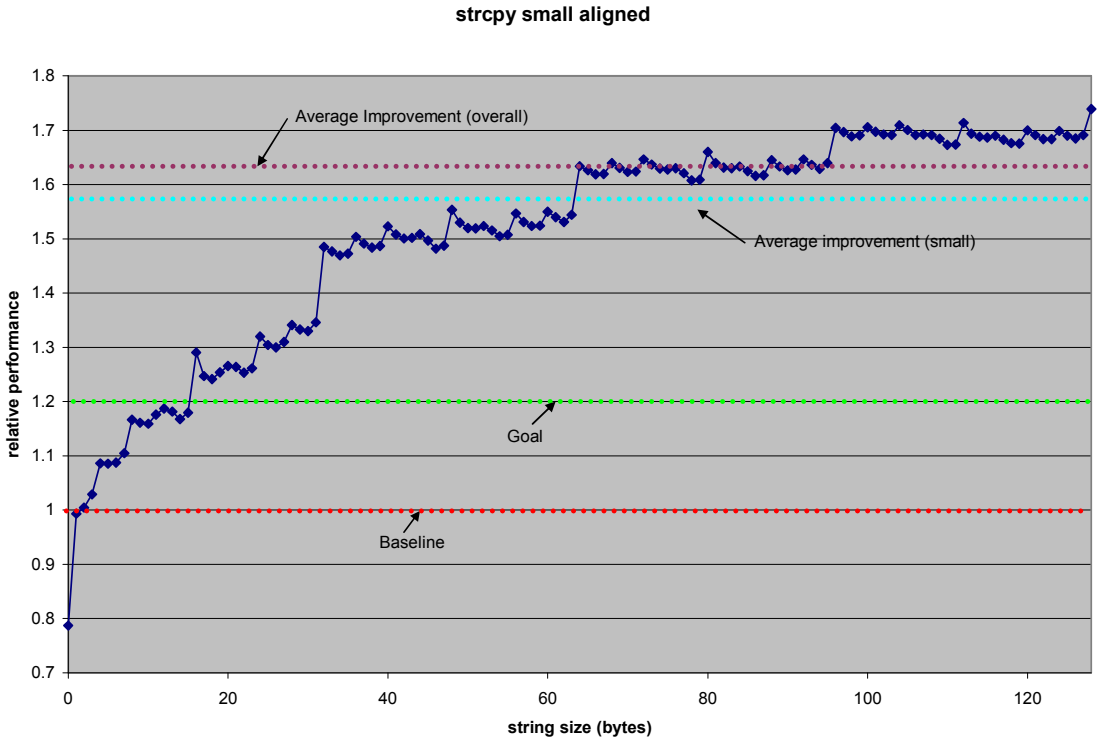
Derivation

The orginal source file for lx_strcpy_r3.S is sysdeps/i386/i586/strcpy.S, however aside from the "glib-isms" and the usage of the registers, the whole of the algorithm is original by the author. The "magic" number algorithm was take from comment in the PowerPC implementation of strlen in Darwin, though the implementation is original with the author

Results

With a 10% negative impact for zero-length string copies, the optimized version exceeds the project goal for all strings longer than 15 characters and 50% faster for strings averaging 64 characters.

Length	Aligned	Unaligned
Trivial	0.95	0.90
Small	1.58	1.47
Large	1.81	1.74
Overall	1.65	



Future Work

- The single pointer increment "trick" could save a clock per dword potentially. This would imply a potential change away from using REP;MOV entirely, however the impact on the codesize sensitive "Aligment Block" section would be significant.
- The "if" driven alignment code used in strlen may be a more efficient approach than the indirect jump.
- The alignment block logic is broken for –fPIC compiles. Work needed to clean up for mainstream.

strlen

```
size_t strlen(const char *s);

    The strlen() function calculates the length of the
    string s, not including the terminating '\0'
    character.
```

Approach

The strlen optimization was a derivative of the strcpy optimization above. The algorithm shares the same general structure

1. Preamble
2. Alignment
3. Dword zero-search, unrolled with prefetch
4. Byte accurate zero find
5. Clean-up

The code shares the same "magic" number zero check as strcpy. However, comparisons for trivial length strings showed the conditional jump based alignment block of the 486 glibc implementation to be more efficient than the computed jump of strcpy above. The dword zero search unroll uses constant offsets to reduce pointer overhead, though for the last iteration of the loop, these increments are executed on exit from the loop. The net effect is that pointer math is only reduced for string of more that 15 characters. Given the need to know the pointer to the last byte, the constant offsets cannot be used in the "byte accurate" zero finding section.

Examining the performance characteristics of the derived function led to the integration of several concepts from the i486 glibc implementation. Useful in this evaluation was comparing the performance (cycle counts) at a variety of buffer lengths which isolated the effects of various code elements. In the final version, lx_strlen_5a_r2.S, the key difference from the i486 (baseline version) are the use of prefetch, and a simpler implementation of the "magic" number algorithm.

Psuedocode for lx_strlen_5a_r2.S

Load s into eax
Note: the following algorithm is derived from sysdeps/i386/i486/strlen.S
Copy eax to ecx
And 3 with ecx
If zero, Goto "Dword Unroll" (eax is dword aligned)
Compare byte at eax with 0
Note:486 implementation use ch for the compare, not Imm8 0. Ch is 0, but is this better?
If equal, Goto "End"
Increment eax
Xor 3 with ecx
If zero, Goto "Dword Unroll
Compare byte at eax with 0 (see note above)
If equal, Goto "End"
Increment eax
Subtract 1 from ecx (Note: if (s & 3) == 2, ecx is 1)
If zero, Goto "Dword Unroll"
Compare 0 with byte at eax (same note)
If equal, Goto "End"
Increment eax
Note: at this point (eax & 3) == 0
Note: end of 486 glibc dword-align algorithm

Label: Dword Unroll
Load dword at eax into edx
Prefetch eax + 32
Set (leal) ecx equal to edx + magic (0xfefefeff)
Not edx
And edx with ecx, store in ecx
Test ecx against 0x80808080
If not zero, Goto "Find zero byte"

Unroll: N in (4, 8, 12)
Load dword at N+ eax into edx
Set (leal) ecx equal to edx + magic (0xfefefeff)
Not edx
And edx with ecx, store in ecx
Test ecx against 0x80808080
If not zero, Goto "Fix up pointer N"
End Unroll

Add 16 to eax
Goto "Dword unroll"

Note: the pointer fixup logic was adapted from the i486 strlen implementation

Label: Fix up pointer 12:
Add 4 to eax
Label: Fix up pointer 8:
Add 4 to eax
Label: Fix up pointer 4:
Add 4 to eax
Label: Find zero byte
Note: This logic is also adapted from the i486 implementation
Load dword at eax into edx
Test dl and dl
If zero, Goto "End"
Increment eax

Test dh and dh
If zero, Goto "End"
Increment eax

Test 0x00FF0000 and edx
If zero, Goto "End"
Increment eax
Note: We don't test 0xFF000000 as we know it has to be 0

Label: End

Substract s (from the stack) from eax (difference is the length)

Return

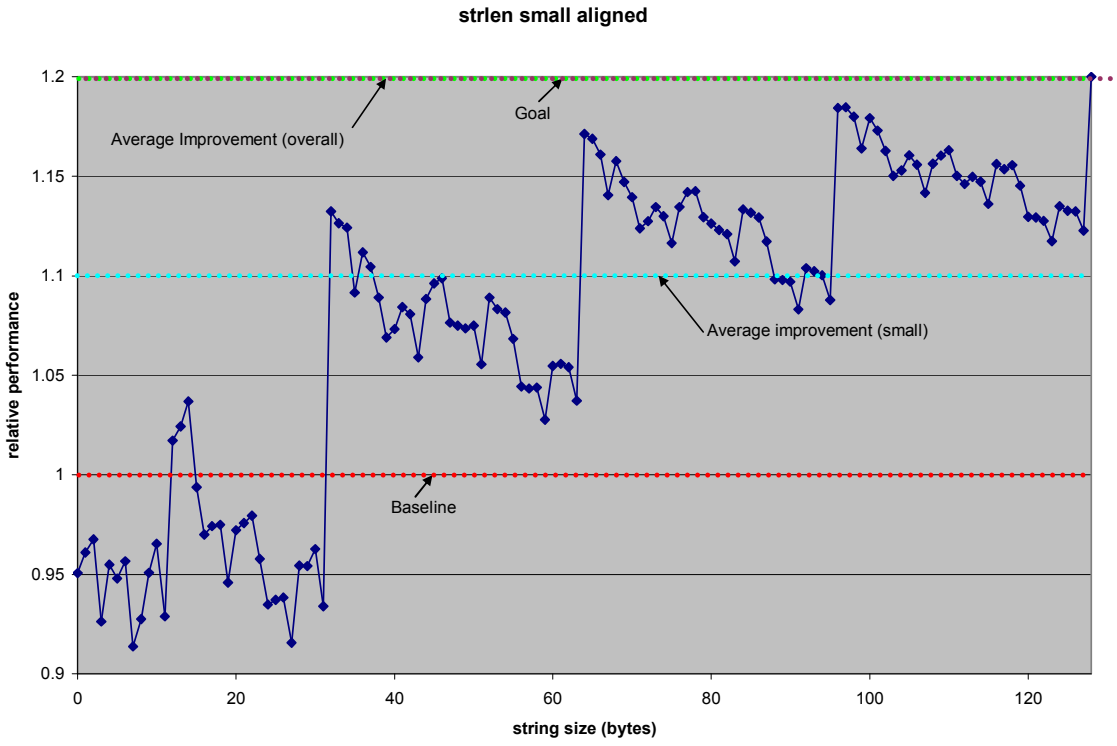
Derivation

While the bulk of the code is algorithmically similar (see notes above) to glibc sysdeps/i386/i486/strlen.S, only the alignment code is quoted directly. In addition the framework (the glibc-isms) are derived (indirectly) from sysdeps/i386/i586/strcpy.S. The Dword Unroll loop is original with the author.

Results

The i486 baseline version was nearly optimal for Geode LX initially. By adding prefetch to the i486 version a 10-20% performance gain was achieved. The refined "Dword Unroll" loop above added an addition 1-3%

Length	Aligned	Unaligned
Trivial	0.95	1.01
Small	1.10	1.13
Large	1.28	1.28
Overall	1.20	



Future Work

- Examine computed jump versus condition jump preamble performance, gain understanding as to the reasons for the better performance, potentially retrofitting strcpy
- In the find zero section the one could use conditional move to clear an increment value (instead of jumping on zero). This would eliminate the mis-predictions, but would ensure that 3 bytes were always checked for 0 (as opposed to the 1-3 checks in the current version)

Future Work

Future work falls into two sections, applying the current work product, and further optimizations.

The current work has two direct applications, potential development of a Geode LX targeted glibc. The first step in this process is to review the product of this project with the AMD Gnu Tools team. Given their feedback, additional work to prepare for mainstreaming will be required. Additionally the current work can potentially be useful to Geode LX customers. Technical Marketing has expressed interest in access to the optimized functions. Additional work will be required to productize these functions, comprising cleanup for publishing, implementation of LPGL-free versions, function renaming, tools and documentation development.

Future work would include additional optimizations to the current set of functions and potentially other functions. The first step in this process must be the profiling of glibc usage in targeted applications, benchmarks, and/or usage cases. Without this step, no additional work should be considered. Work required will include building interposers and other profiling tools, selection of target applications etc., gathering of usage data, and evaluation of the likely optimizations available for the candidates, and the optimization of the most import N functions. Assuming there is a meaningful set of optimizations, preparations for mainstreaming and distribution to customers should be including the planning.

Conclusion

This project optimized 6 select functions from the libc interface. These optimizations were tested in isolation, with 5 of 6 meeting or exceeding the targeted performance gain of 20%. However, these functions were neither selected or measure in the context of typical application usage. Thus while we have potentially substantial local improvement in these functions, we do not have direct or indirect measure of the overall improvement of a given application or benchmark.

The project was particularly challenging, as the baseline code from glibc is the product of expert x86 assembly developers. In spite of this, we discovered several examples where

targeted optimizations for other x86 platform were substantially worse for Geode LX. We also discovered examples where optimizations for the LX decreased performance on Athlon or Pentium processors. Both of these together validate the need for Geode specific optimizations, as the community clear won't code optimizations that slow the mainstream processors.

Some clear common themes emerged from the optimizations emerge:

Prefetch	Effective prefetch of a single data source can contribute up to 20% performance win.
Misprediction elimination	coin-flip conditional jumps can be replaced by conditional moves, combined into a single condition jump, or by empty REP's,
Dword-then-byte	this is a general approach, but not uniformly implemented
Unroll	another general approach, frequently needed to use prefetch
Pointer math simplification	A win on Geode, sometime a loss on Athlon or Pentium
Detailed ordering	A weaker effect, but can save 20% in very localized ways, though a single block reorder of several instructions never affected an overall result by more than 1%.

The results here do not seem evocative that an optimized compiler supporting the overall performance goals of 25% (or more), but do indicate that targeted optimizations can achieve this range of optimization at least locally. In any case, future candidates for optimization should be selected careful profiling of target applications and use cases.