A COMPARISON OF SCALABLE MULTI-THREADED STACK MECHANISMS

A Comparison of Scalable Multi-Threaded Stack Mechanisms

By

Joshua I. Moore-Oliva, B.Comp.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree of

Master of Science
Department of Computing and Software
McMaster University

MASTER OF SCIENCE (2010)
(Department of Computing and Software)

McMaster University
Hamilton, Ontario

TITLE: A Comparison of Scalable Multi-Threaded Stack Mechanisms

AUTHOR: Joshua I. Moore-Oliva, B.Comp. (University of Guelph)

SUPERVISOR: Dr. Emil Sekerinski

NUMBER OF PAGES: ix, 81

Abstract

The traditional "stack grows from the top, heap grows from the bottom" memory layout allows a single-threaded process to make use of all available address space. This layout is not ideal when multiple threads of execution need to share one address space, for memory exhaustion is no longer signified by the heap meeting the stack. In the commonly used multi-threaded memory layout where each thread has its "worst case" stack memory exclusively reserved, a process may prematurely run out of memory when one thread's stack collides with another's, even if there is unused address space elsewhere. This problem is exacerbated as the number of threads in a process increases since there is less stack space available per thread.

In this thesis, alternative stack mechanisms that attempt to alleviate this problem are reviewed, and a new stack mechanism is put forward that utilizes the MMU to detect stack overflow. An experimental compiler implementing a subset of the C language is used to implement promising stack mechanisms, and a suite of test programs are used to compare their performance and scalability under varying usage patterns.

Acknowledgements

I would like to thank my supervisor, Dr. Emil Sekerinski, for always being available with clear and unambiguous guidance. I would also like to thank Dr. William Gardner and Dr. Mark Wineberg for giving me many candid insights into academic life, both during my undergraduate degree at the University of Guelph, but also during the eight months I spent there as a graduate student before transferring to McMaster University. Last but not least, I would like to thank my long-time girlfriend Alicia Gumtie, whose strengths complement my weaknesses, and whose patience was neverending as I made my way through two schools and three sets of supervisors during my graduate career.

Contents

List of Figures

Chapter 1

Introduction

The traditional call stack mechanism – where the stack and heap grow from opposite sides – is an indispensable part of almost any program, yet it is often ignored. This is because, until now, it has been an effective solution to the problem of bookkeeping during program execution – it provides a good way to keep track of variable values and program flow. A single-threaded process, executing in a system with an MMU (Memory Management Unit), therefore has little reason to use anything but the traditional call stack mechanism. In fact, with virtual address space being so abundant, many popular operating systems take the strategy of allocating a stack area "large enough for anything", and on overflow do not attempt to extend it downwards.

However, complications arise when multiple threads of execution need to share the same address space, as they each require their own stack. Since there are multiple stacks, the traditional call stack mechanism no longer applies. In the simple case, where a program only uses a few threads (for example, one thread runs a program's user interface and another thread performs longer computations in the background), virtual address space is so abundant that allocating one fixed-size "large enough" stack for each thread is a simple solution that works well enough to be a commonly implemented technique in modern operating systems [22].

The recent trend for software development has been towards "more concurrency". There are two reasons for this: firstly, it allows for more natural modelling of systems, and secondly, it takes advantage of the hardware trend to increase performance via parallelism with multi-core processors [6], [10]. This has led to the more complex scenario: when concurrency is increased with the use of threads, the default "large enough" call stack mechanism actually causes virtual address space to become exhausted when otherwise more threads could be handled by the operating system, especially on modern multi-core systems. This is the case even though the vast majority of the address space is unused! Reducing the stack size for each thread can alleviate the issue [12], but at the expense of increasing the probability of running out of stack space for legitimate use. Stack sizes can be manually specified on a per-thread basis [22], but doing so requires programmer intervention and only slightly alleviates the problem, as each thread's stack is sized according to the amount of memory it requires at its point of highest usage (leading to situations where a program can "run out" of stack space even if there is a large amount of stack space unused by other threads).

The comparison of call stack mechanisms for highly concurrent multi-threaded programs is the topic of this thesis, with the goal of discovering or identifying an efficient multi-threaded call stack mechanism that works as well and as transparently as the call stack mechanism for single-threaded processes. The need for an improved call stack mechanism was highlighted during the development of Lime, a concurrent object-oriented language that has been designed with formal verification and refinement in mind. With Lime, in principle, every object can be concurrent. This can easily lead to programs with hundreds of threads. An implementation of this language by Xiao-lei Cui during the course of his Master's Thesis confirmed that the call stack mechanism was not scaling with the concurrency of the program. As the focus of his research at the time was proof-of-feasibility, stack sizes were intentionally set low and stack-gobbling features, most notably recursion, were disabled as a workaround [11].

Chapter 2

Related Work

Modifying the traditional single-threaded "Stack grows from the top, heap grows from the bottom" call stack mechanism for multi-threaded programs is not a new idea. The goal of this chapter is to categorize and discuss existing and proposed multi-threaded call stack mechanisms. Before that discussion takes place, this chapter briefly discusses the existing single-threaded call stack mechanism for readers that are not already familiar with it, and then explains the problems that multi-threading poses.

2.1 Background

2.1.1 Single-Threaded Call Stack

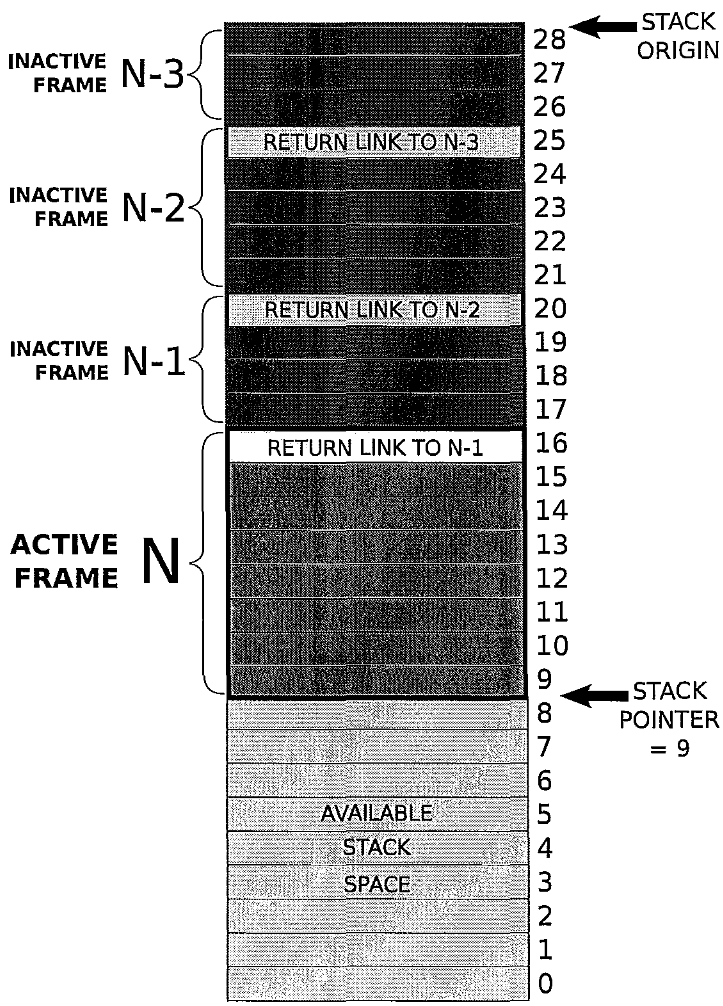
Every procedure needs somewhere to store local variables. Additionally, when a procedure A calls another procedure B, the current state for procedure A must be preserved for when procedure B returns. This information must be stored in a dynamic data structure and cannot be computed at compile time due to the following popular language mechanics:

- **Recursion** - The number of times a procedure will call itself may be dependent on input data.

- **Dynamically bound methods** - It is not always possible to determine the call path a program will take, as the procedure that will actually be invoked may not be visible (or even existent) at compile time.

- **Variable length arrays** - The length of an array that is a local variable for a procedure may not be known at compile time.
- **Interrupts** - Upon receiving a signal, a programmer defined handler may be invoked. The handler may call any number of procedures, and when they return, the program must continue from its previous state.

The fact that a procedure will not return until all other procedures it called have returned lends itself nicely to a stack, where each new procedure invocation is placed on top of all existing procedure invocations. As such, single-threaded programs tend to use the call stack structure depicted in Figure 2.1.



stdcall

Figure 2.1: Call Stack http://en.wikipedia.org/wiki/File:ProgramCallStack2.png

It is not by accident that Figure 2.1 shows the stack growing downwards. In a single-threaded program, there is only one call stack. This has led to the following memory organization where:

- The stack grows from high to low memory addresses.
- There is a dedicated region for code starting at a low memory address.
- The heap memory for a process grows from low to high memory addresses.

As can be seen in Figure 2.2, this memory organization allows for heap and stack memory to utilize all available memory (heap fragmentation issues aside). When the two memory regions meet, the program is out of memory.

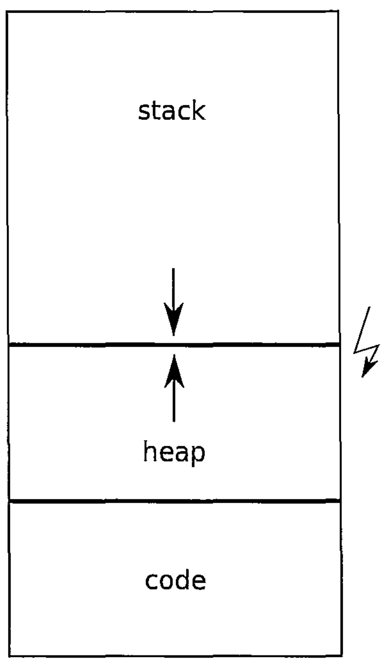


Figure 2.2: Traditional Single-Threaded Memory Organization

In practice, memory mappings (such as those used for shared libraries) situated between the heap and the stack will cause program faults before intersection of the two regions. Due to this, operating systems such as Solaris, Linux and Windows limit stack space to a fixed size that is "large enough", and if the program attempts to use more than the pre-allocated stack space, it is considered a program error [22, 19, 4].

2.1.2 Multi-Threading's Call Stack Problems

Before examining the problems that multi-threading introduces, it is a good idea to first examine why the traditional call stack mechanism works so well for single-threaded processes. Two important facts form the basis of its success:

- The MMU allows each process to use the entire address space as if it were the only process running on the system. Physical memory is not reserved for the process until it actually uses the space.
- For any single-threaded program, there is only one call stack required.

As such, the operating system can, by default, reserve a stack so large that it is usually safe to assume that a properly-functioning program will not exhaust it. Reserving such a large portion of memory does not cause any negative effects because:

- Virtual address space will not map to physical memory until the program actually uses the virtual address space.
- The operating system automatically maps the used virtual address regions to physical memory that will not conflict with other processes.

Multi-threading significantly changes the rules. A multi-threaded program requires one call stack per thread, all of which must exist within the same address space. This means that the MMU cannot help with multiple threads as it does with multiple processes. Most modern operating systems just create one "large" call stack for each thread at the top of virtual address space. However, when there are a large number of threads, this will cause the process to run out of virtual address space before it is actually out of memory. Shrinking each process's stack space until each thread's stack can fit may lead to one thread running out of stack space when there is otherwise lots of unused stack space remaining, as can be seen in Figure 2.3. This is especially likely to happen if thread stack usage patterns differ (e.g. one thread makes heavy use of recursion). It is possible to manually set stack space on a thread-by-thread basis (e.g. giving a heavy stack space using thread more stack space). However, this both increases the burden on the programmer and decreases the flexibility of the program (threads are locked into roles, not all threads have the ability to temporarily use a large amount of stack space). This harkens back to the days before the MMU when programmers used to manually give each process a certain region of memory space. Such work is tedious and error-prone, and goes against the productive trend of operating systems and languages automatically managing and sharing system resources.

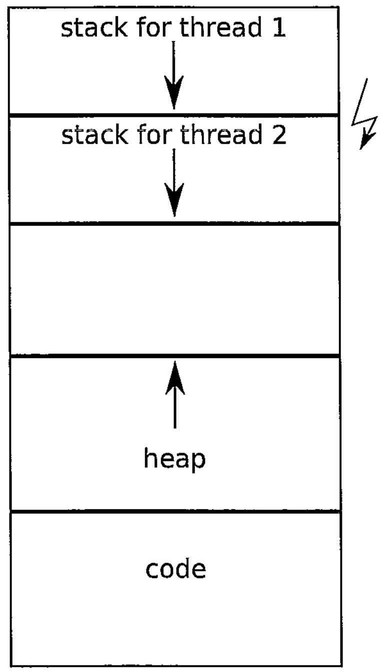


Figure 2.3: Single-Threaded Memory Organization Extended to Multiple Threads

With the number of cores on chips increasing, parallelism being touted as the way to increase performance in the future [23, 6, 10], and current operating system stack mechanics being a bottleneck for the number of threads a process can run, it is clear that the lowly call stack is in need of investigation.

2.2 Single-Threaded Stack Mechanism Extensions

This section discusses call stack mechanisms that are relatively trivial modifications of the traditional single-threaded call stack mechanism. All maintain each thread's call stack as a contiguous region of memory.

Solaris [22] uses a multi-threaded call stack mechanism that is practically the standard for modern operating systems. Each thread has its own stack space reserved near the top of virtual address space. The size of the call stack can be set to a custom value during thread creation. If no stack space size is specified, a large value (typically

2MB) will be used instead. Stack overflow is detected via the use of a "red zone", which refers to the process of appending a page of memory without read or write permissions to the end of a thread's stack space. This page will causes a memory fault if accessed.

Oberon with active objects [12] can be viewed as a subset of the above call stack mechanism specifically tailored to support a large number of small call stacks. It does this by reserving the upper 2GB of virtual address space for small call stacks that are each a maximum of 128KBytes, thereby supporting up to 16,384 call stacks simultaneously.

Concurrent Oberon [18] uses a call stack that is a fixed size determined at thread creation, but allocated on the heap. Overflow is detected before it occurs via a check at the start of every procedure, and results in termination of the offending thread. While this method increases runtime overhead, it has the advantage of working on systems that do not have an MMU. The call stack is garbage collected once the thread terminates.

US patent 7,477,829 [27] attempts to address both heap contention and stack space in its proposed memory layout, depicted in Figure 2.4. Each stack/heap block is created from an initial base address, from which the thread and heap stack grow in opposite directions. Unfortunately, the patent does not specify how the initial base addresses are computed, but from Figure 2.4 it can be inferred that the base addresses are intended to be spaced apart evenly. Doing so would require knowledge of the maximum number of threads that the program would execute at one time. Stack and heap overflow are detected via the use of "dead zones" (depicted in Figure 2.5) that are "... impossible to read from or to write to ... In so doing there is no chance of memory corruption between any of these thread heap/thread stack combinations" [27]. While the patent does not go into further details, it is inferred that these dead zones operate similarly to Solaris's red zones [22] by generating a page fault or similar hardware interrupt upon access.

All of the above methods suffer from the limitation that stack space for one thread cannot be shared with another, and each thread's stack space must always be large enough to handle the worst case stack usage, or the program will terminate with an error. This can lead to the situation where a program can prematurely "run out" of stack space due to a single thread exceeding its allotted stack space, even if there is plenty of unused stack space preallocated to other threads. These conditions also force a trade-off between the maximum allowable stack space per thread and the number of threads that can exist in a system at one time, which seems to run counter to the spirit of resource-sharing mechanics that govern system memory and hard disk space. It is my opinion that this trade-off is a vestige of the success of the MMU (which gives the most assistance to processes that do not share address space) combined with the fact that a single-threaded program only requires a single call stack.

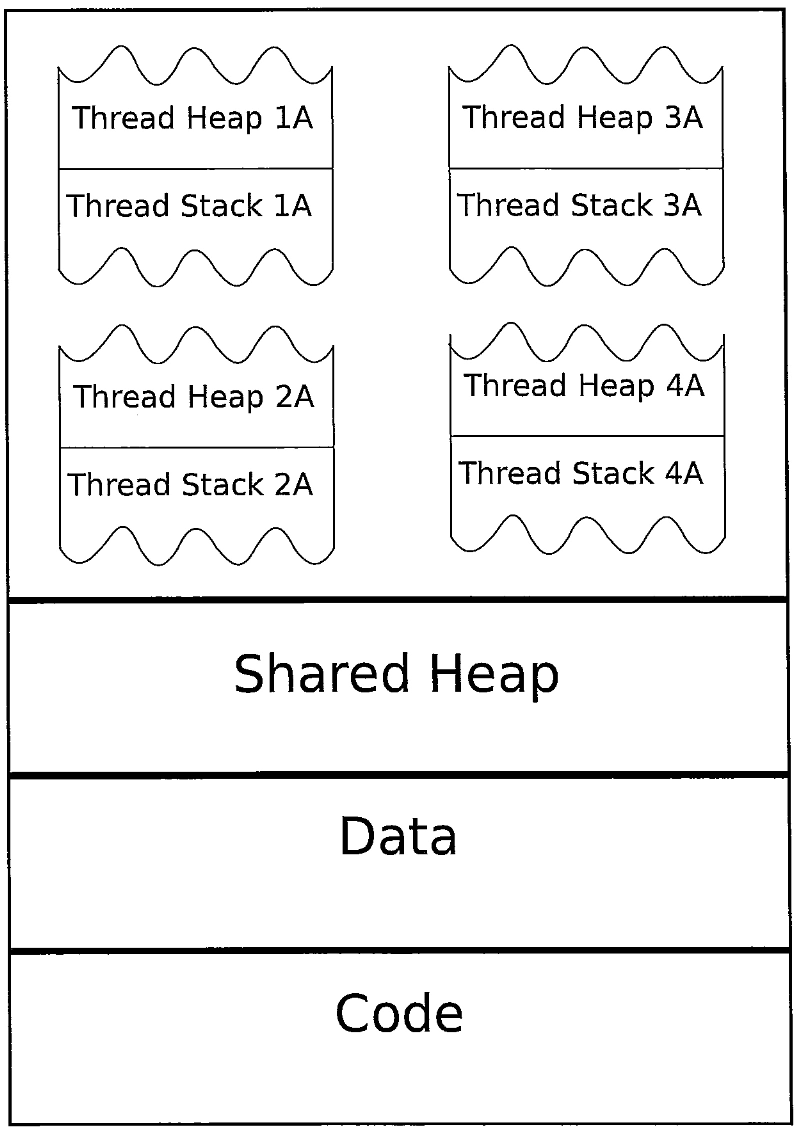


Figure 2.4: US Patent 7,477,829 Memory Organization [27]

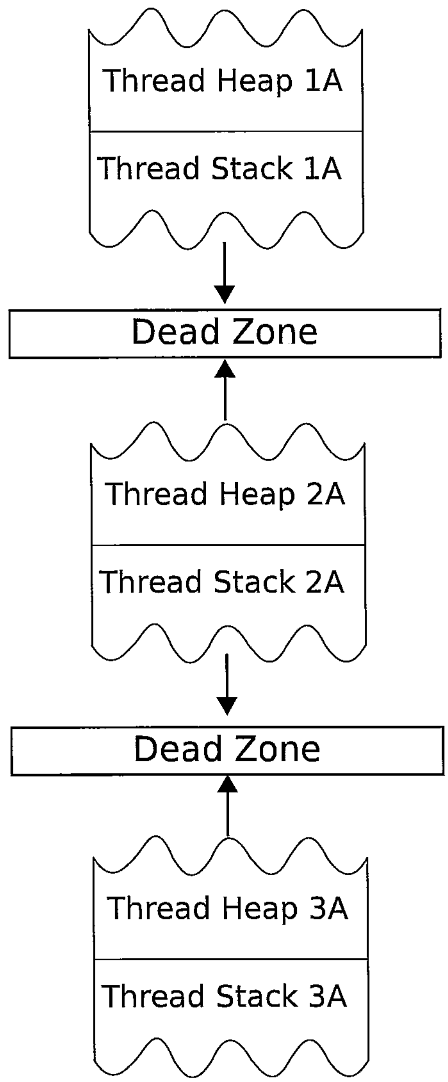


Figure 2.5: US Patent 7,477,829 Dead Zones [27]

2.3 Stack Sharing

This section discusses those call stack mechanisms with the general strategy of attempting to share stack space among many threads in some way, as opposed to the traditional strategy of each thread having an exclusively reserved stack.

Hybrid Stack Sharing [28] creates a fixed number of stacks in memory, and attempts to evenly distribute threads among them using a round-robin approach (see Figure 2.6). On a context switch, if there is not an unused stack available, the used portion of the exiting thread's stack will be copied to heap memory, and the new thread's stack data will be copied in. Hybrid Stack Sharing makes no mention of handling stack overflow, and the authors mentioned that they always kept the stack size large enough that overflow would never occur, so it is assumed that there is no mechanic for handling stack overflow. Hybrid Stack Sharing improves upon the standard multi-threaded stack handling approach [22, 19, 4] by introducing a constant amount of memory fragmentation (there are a limited number of large stack areas that take up address space).

Thread Stack Association Table			
Thread Number	Stack Index 0	Stack Index 1	Stack Index 2
thread 0	X		
thread 1		X	
thread 2			X
thread 3	X		
thread 4		X	
thread 5			X

Figure 2.6: Hybrid Stack Sharing [28]

Multi Task Stack Sharing [20] is a multi-threaded call stack mechanism designed for embedded systems where address space is limited. Each thread begins with its own call stack space, similar to the traditional mechanism employed by the Solaris stack. Overflow is detected via a runtime check at the beginning of each procedure. On overflow, a "page" is reserved at the end of another thread's stack and used for the overflowing thread's stack. The system attempts to share overflow equally among all thread stacks until there is no space left (see Figure 2.7). As such, this call stack mechanism is able to share unused stack resources, and each thread's call stack can be created smaller, reducing the amount of memory that needs to be dedicated to call stacks. While this is an improvement over the standard multi-threaded stack handling approach [22], total stack space is still a fixed size. Hence, the program can run out of either stack or heap memory when there is still unused space remaining. Additionally, the non-contiguous nature of the stack means that there is some fragmentation when a stack frame cannot fit into the free space left at the end of a stack page, and a new page must be used in another thread's stack area.

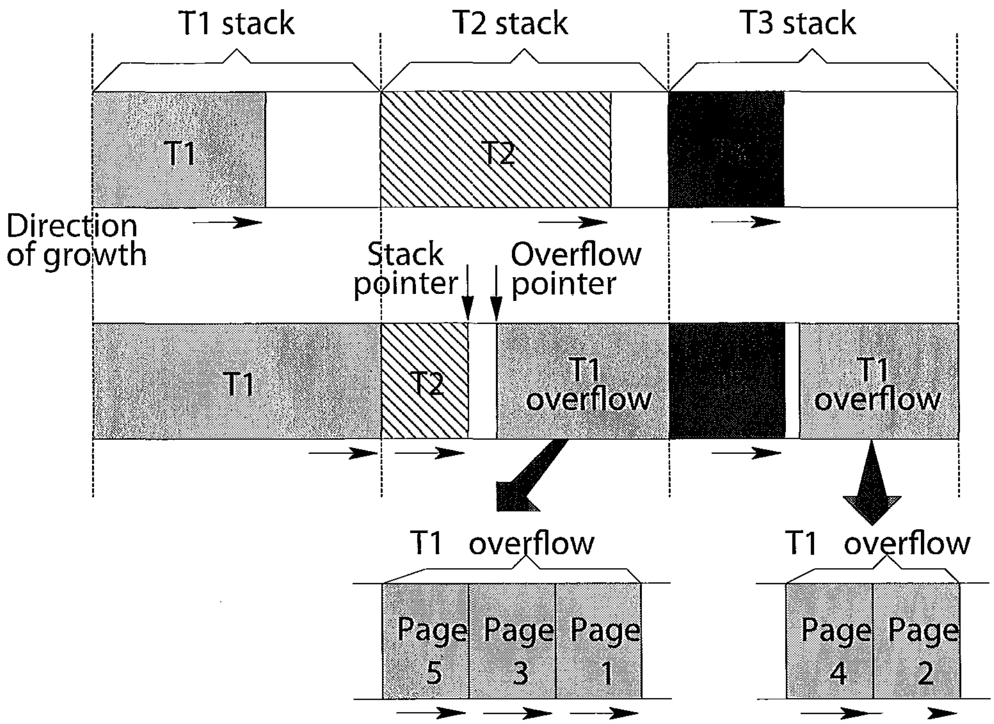


Figure 2.7: Multi Task Stack Sharing [20]

A Meshed Stack [14] is a call stack mechanism where all threads place their stack frames at the top of one central stack. When a stack frame is no longer valid, the frame is marked as garbage. A special call stack garbage collection routine is run periodically to compact the stack. This call stack mechanism inherits all the advantages of the single-threaded call stack mechanism (no fragmentation, the ability to extend stack and heap until they meet, and so on), at the expense of arbitrary program pausing during stack compaction. Further analysis of this stack mechanism is impossible, as the cited paper [14] gave only an overview referencing a thesis that was in preparation for further details. The referenced thesis was never completed.

2.4 Cactus Stacks

This section discusses those call stack mechanisms that attempt to use the cactus stack data structure to link multiple non-contiguous regions of memory together into a single call stack. A cactus stack is a tree data structure where child nodes point to their parents. Note that a linked list can be considered a subset of the cactus stack.

Stackless Python [24] is an unfortunately misleading name, but the call stack mechanism it uses is interesting nonetheless. Standard "Stackful" (as opposed to "Stackless") Python uses a mechanism where the C call stack is intertwined with the interpreter. Stackless Python moves all the data that was stored in the C call stack into linked interpreter frames that also contain code. Moving stack data into the interpreter has allowed for features such as continuations, which allows for saving and resuming program state. The stack itself is little more than a linked list of stack frames. This allows the stack to live within heap memory (pushing any fragmentation issues to the heap allocator), and removes arbitrary limits on stack size. Invoking a heap allocation for every procedure call has performance implications, but since Python already does this to keep a frame object associated with every running piece of code, moving the stack into a similar structure does not negatively affect performance.

Thread Segment Stacks [21] is a multi-threaded stack implementation for gcc [1]. To begin with, each thread gets its own contiguous stack space, just like the Solaris's [22] traditional multi-threaded stack mechanism. Stack overflow is detected via the use of inlined code around the call instructions for the prologue and epilogue of procedures. When stack overflow is detected, a "linear extension" is performed if possible, which attempts to map a new page of memory contiguously to the previous virtual address. If a linear extension cannot be performed, a new stack segment is allocated elsewhere, and a linked list is formed. This call stack mechanism removes the false "out of stack space" errors that traditional multi-threaded stack management faces, allowing for initial call stack sizes to be smaller. However, it does so at the expense of runtime overhead for every procedure call (in the average case of no stack extension, that overhead is reported as 5 + 3 additional instructions per procedure call). There is also some memory fragmentation that will occur on a non-linear extension when a stack frame cannot fit into the remaining space in a stack segment.

Capriccio [26] is a user-level thread package that uses a call stack mechanism that can be viewed as a refinement of the mechanism used in Thread Segment Stacks [21]. The major change that Capriccio makes is that it analyzes the call graph of a program, depicted in Figure 2.8, at compile time to combine many subroutines with small stack sizes into one larger block, thereby reducing the number of prologue and epilogue checks that need to be made during procedure calls. For example, two consecutive function calls, X and Y, requiring 10 and 20 bytes of call stack space respectively, would have only one prologue check before X for 30 bytes and one epilogue check at the end of Y. Calls to external functions not call-graph analyzed are handled by programmer annotations specifying minimum stack requirements for the function, or just by a default "large enough" call stack chunk. When function pointers are concerned, the compiler considers all possible functions that could match the function pointer in question. Polymorphism, while not explicitly mentioned, could conceivably be handled in a similar manner.

Capriccio, like Thread Segment Stacks, still suffers from a degree of call stack memory fragmentation. However, Behren et al. [26] have analyzed the problem as follows: "Internal wasted space" is defined as the space wasted at the end of a call stack region when a new call stack region is linked. "External wasted space" is defined as the unused (but possibly usable) space at the end of an active call stack region. The introduction of function stack check combining introduces a trade-off between internal wasted space and speed. The larger each call stack region, the less procedure checks need to be made, but the probability of a stack chunk not fitting at the end of a call stack region is increased. There is also a tradeoff between external wasted space (an issue if there are many threads running) and internal wasted space. Large stack chunks result in more external wasted space, but less frequent stack linking (resulting in less internal wasted space). Capriccio's call stack mechanism removes false "out of stack space" errors, minimizes overhead from inlined stack check code due to call graph analysis, and provides tunable parameters to balance memory fragmentation tradeoffs to application requirements.

2.5 Other

Event-based programming is a programming paradigm where a single thread of execution is responsible for detecting, and then sequentially handling, events. Protothreads [13] is an event-based thread implementation that loosely emulates threading via the use of a state machine. Context switching is performed via stack rewinding. This mechanism forces only one thread to run at any given time, and the use of blocking system calls will cause the entire program to pause. As such, this threading

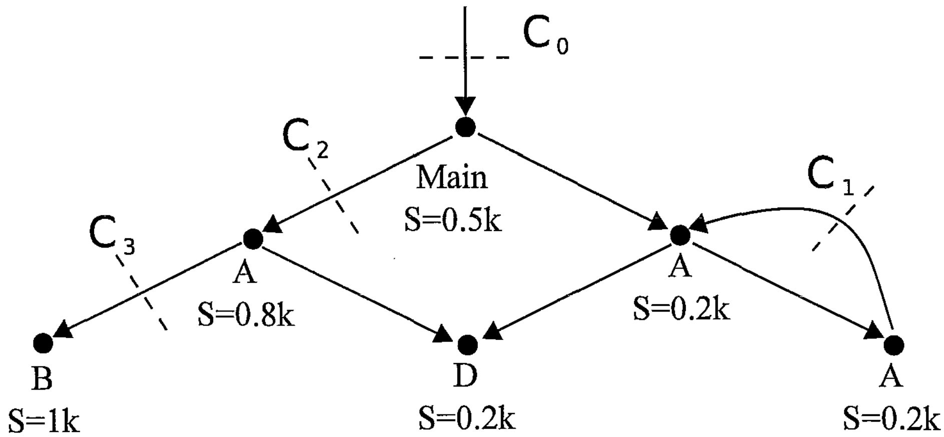


Figure 2.8: An example of a call graph annotated with stack frame sizes. The edges marked with C_i (i=0,...,3) are the checkpoints. [26]

library is not considered general purpose, and was developed specifically for embedded systems.

In contrast to Protothreads, "Why Events Are A Bad Idea" [25] was published arguing the superiority of threads over event-based systems. While I will avoid entering this debate, the paper did note stacks as an issue for systems involving a large number of threads, and offerred the following suggestions for call stack strategies in multi-threaded systems:

- Development of mechanisms that allow for dynamic call stack growth.
- Use of compiler analysis to determine the stack requirements of functions and identify areas of code that may require stack growth.
- Purge unnecessary state from the call stack before making a new function call. This would require the compiler to arrange the stack in such a way that live variables do not pin dead in the call stack when a function call is made. Figure 2.9 illustrates this concept: On the left, the live variables y and z pin the dead variable x. On the right, moving the dead variable x to the bottom of the stack allows the next procedure call to make use of x's storage.

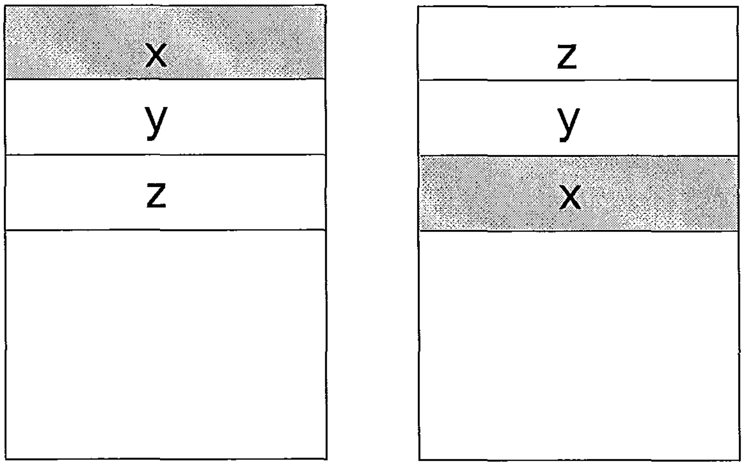


Figure 2.9: Live Variables Pinning Dead

2.6 Overview

This section contains a tabular overview of various features of the multi-threaded stack mechanisms reviewed. Preceding the table is a legend explaining the columns' meanings.

- Runtime Overhead - This refers to runtime overhead above what the traditional single-threaded call stack mechanism would incur.

  Constant No additional runtime overhead beyond initial setup.

  Procedure call Additional runtime overhead with every procedure call.

  Linear Procedure Call Grouping prevents additional runtime overhead with every procedure call, but additional runtime overhead is still asymptotically linear with respect to procedure calls.

  Context switch Additional runtime with every context switch.

  Global Global routines need to be run periodically to maintain the call stack, which result in program pausing.

- Memory Overhead - Additional call stack memory overhead above what the traditional single-threaded call stack mechanism would incur.

  Constant No additional memory overhead beyond initial setup.

  Procedure call Additional memory overhead with every procedure call.

On Extension Constant memory overhead on stack extension.

- Premature Out-Of-Memory

No Memory organization theoretically allows for a process to use its entire Virtual Address space before running out of memory.

Negligible Memory organization may result in fragmentation similar to heap allocation, but conceptually the entire Virtual Address space can be used.

Thread/Heap Memory organization allows sharing of call stack space among threads, but stack space is a fixed size and once that is used up, the system will be "out of memory" even if there is remaining unused memory. Similarly, if the heap runs out of space before the call stack does, any space reserved for the call stack cannot be used by the heap.

Single Thread Memory Organization is such that each thread has a fixed amount of call stack space, and if one thread exhausts its call stack space it cannot use any other available memory in the system. Heap can prematurely run out of memory as in Thread/Heap.

Approach	Runtime Overhead	Memory Overhead	Premature Out-Of-Memory
Solaris	Constant	Constant	Single Thread
Oberon with active objects	Constant	Constant	Single Thread
Concurrent Oberon	Constant	Constant	Single Thread
US Patent 7,477,829	Constant	Constant	Single Thread
Hybrid Stack Sharing	Context Switch	Constant	Single Thread
Multi Task Stack Sharing	Procedure Call	On Extension	Thread/Heap
Meshed Stack	Global	Constant	No
Stackless Python	Procedure Call	Procedure Call	No
Thread Segment Stacks	Procedure Call	On Extension	Negligible
Capriccio	Linear Procedure Call	On Extension	Negligible
Protothreads	Context Switch	Constant	No

Figure 2.10: Stack Implementation Overview

Chapter 3

Experimental Setup

As stated in Section 1, the goal is to discover or identify an efficient multi-threaded call stack mechanism that works as well and as transparently as the call stack mechanism for single-threaded processes. Therefore, any multi-threaded call stack mechanism selected for analysis must be scalable. As such, each mechanism must have the following characteristics:

- Compatible with concurrent multithreading (as opposed to user space threads where only one thread may run at a time)
- The use of a central locking mechanism must be used sparingly, if at all. Otherwise, scalability will suffer, especially if the locking is performed on a per procedure call basis.
- Dynamic sharing of memory between thread call stacks. No allocating a fixed amount memory to each thread at the start and saying "this will be enough".
- Stack data must be referencable. It cannot move around. This decision was made to maintain compatibility with existing system calls, as well as to avoid the overhead and locking associated with moving stack data around.

3.1 Rejected Mechanisms

The following methods, reviewed in Section 2, did not meet the above criteria and were not selected for experimentation.

All methods from Section 2.2 lacked dynamic sharing of memory between thread call stacks. Each method had the common mechanism of assigning each thread an exclusive, fixed size call stack.

Hybrid Stack Sharing [28], reviewed in Section 2.3, uses a fixed number of fixed size call stacks for running threads. The context switching penalty of copying stack data would be too expensive for a system running a large number of threads, which requires fast and efficient context switching.

Multi Task Stack Sharing [20], reviewed in Section 2.3, was created for embedded systems with a single processor. Extending the mechanism to allow for true multi-threading would require synchronization for every procedure call to eliminate race conditions between a thread using its call stack, and another thread allocating space in that call stack.

Meshed Stack [14], reviewed in Section 2.3, would require moving of call stack variable addresses. This disallows using stack variables as arguments to procedure calls, especially system calls. Additionally, the overhead required (stopping all threads to compact the call stack, or synchronization mechanisms) would harm scalability. Finally, the thesis that was referenced for details on the workings of the Meshed Stack was in preparation at the time of publishing [14], and it appears that the thesis was never completed.

Protothreads [13], reviewed in Section 2.5, forces only one thread to run at any given time.

3.2 Technology Overview

This section gives an overview of the technologies used in the following Sections.

3.2.1 Pthreads

In this document pthreads refers to the Native POSIX Threading Library or NPTL (as opposed to the older LinuxThreads implementation) that is the standard Linux implementation [8] of the IEEE POSIX standard [15]. "POSIX.1 specifies a set of interfaces (functions, header files) for threaded programming commonly known as POSIX threads, or Pthreads. A single process can contain multiple threads, all of which are executing the same program. These threads share the same global memory (data and heap segments), but each thread has its own stack (automatic variables)." [5].

3.2.2 Intel® x86 Assembly

The assembly code detailed in Section 3.4 is Intel® 32-bit x86 assembly code [17]. The basic architecture consists of eight 32-bit general purpose registers: EAX, EBX, ECX, EDX, ESI, EDI, ESP and EBP. While all of these registers have some special uses, by far the most specialized register is ESP, the stack pointer, whose value is changed by the CALL and RET instructions. All other registers are used as general purpose registers except where noted otherwise in Section 3.4.

3.3 C--

The stack mechanisms outlined in Section 3.4 require instruction sequences for procedure calls that cannot always rely on a contiguous stack frame. Two existing open source compiler frameworks, gcc [1] and LLVM [2] were evaluated for modification, and discarded, for the following reasons:

- Existing public interfaces to modify the instruction sequences for procedure calls were limited to modifications that still relied on a contiguous stack frame. It would have required deep understanding of the software to modify the instruction calling sequence.
- It was unknown if some optimizations relied on a contiguous stack frame.

Given the above analysis, it was decided to build a C-like compiler from scratch to save time and avoid unforeseen complications resulting from modifying an existing complicated codebase without full understanding of its workings. This C-like language is an almost perfect subset of the C language and was given the unoriginal name of C--.

3.3.1 C-- Implementation Overview

The C-- compiler has no preprocessor, and accepts only one source file as input. The C-- compiler emits 32-bit x86 instructions compatible with the open source assembler NASM [3]. All interfacing with existing C standard library routines relies on NASM's global and extern commands. Unlike the standard cdecl calling convention which requires procedures to preserve the values of EBX, ESI, EDI and EBP, C-- assumes that any procedure call can trash any register (except where a register is specially reserved by a stack mechanism).

3.3.2 Omitted C Language Features

Following is a list of C language features that C-- does not implement. These features were not omitted for any particular reason, merely that they were not needed for the experimentation and hence not implemented.

- Dynamic allocation of memory on the stack (stalloc).
- Variable array declaration on the stack.
- Variable declarations cannot have initializers. Initialization of variables must be a separate statement following the variable declaration.
- Structs cannot be assigned with the a={x,y,z} syntax. Struct members can only be set individually with the . or $->$ operators.
- Strings do not implement all escape sequences. Only \n, \r, \t, \\, and \" are implemented.
- Function pointers. A pointer to a function can be accessed by using the function name (type is int), but a function cannot be called from a pointer.
- Compound assignment operators.
- Increment and decrement operators.
- Ternary operator.
- Wide string literals.

C language keywords that C-- does not implement: auto, const, enum, goto, long, register, signed, static, switch, typedef, union, unsigned, volatile.

Data types that C-- supports: int, char, double, struct, void. Pointer variations (such as char *, void ** etc) are supported.

3.3.3 C-- Additions

The C-- compiler is a multi-pass compiler, and as such there is no need for forward declarations of any kind. For example, in C-- the C code in Listing 3.1 does not require the forward declaration at line 1.

Listing 3.1: Forward Declaration in C

```
1  int b( int y );
2
3  int a( int x ) {
4    if ( x > 0  ) {
5      x = x - 1;
6      b( x );
7    }
8  }
9
10 int b( int y ) {
11   if ( y > 0  ) {
12     y = y - 1;
13     a( y );
14   }
15 }
```

C-- has introduced a macro, stacksizeof(procedure), which like the C macro sizeof(type) returns the stack size for a given procedure.

3.4 Implemented Stack Mechanisms

3.4.1 Traditional Fixed-Size Stack

This section describes an implementation of the traditional fixed-size call stack mechanism, outlined in Section 2.2. This call stack mechanism does not meet the criteria outlined in Section 3. While I compare all implemented stack mechanisms against the traditional stack mechanism implemented in gcc with various levels of optimizations, the traditional stack mechanism is reimplemented in the C-- compiler to provide a comparison independant of variations in optimizations and code quality not directly related to the stack mechanism being evaluated.

Caller Instructions

The caller routines for this stack mechanism implement gcc's standard calling convention [7]. The caller is responsible for pushing arguments to the stack, as well as cleaning the stack on procedure exit.

Listing 3.2: Traditional Fixed-Size Stack Caller Instructions

```
PUSH arg1
...
PUSH argn
CALL callee_name
ADD  ESP, args_size
```

Callee Instructions

The callee is responsible for ensuring that the stack pointer has the same value on return from the procedure as it did on entry. This is accomplished by initially extending the stack by the amount of stack space required by the procedure (S), and then ensuring that every RET instruction is prefixed by an instruction to decrease the stack by S.

Listing 3.3: Traditional Fixed-Size Stack Callee Instructions

```
callee_name: SUB ESP, S
             ...            #Body of procedure
             ADD ESP, S
             RET
```

3.4.2 Per Procedure Heap Allocation

The call stack for a program is structured as a linked list, as shown in Figure 3.1. Each procedure invocation has its own stack frame, just large enough to hold the stack information depicted in Figure 3.2. Each thread of execution has a dedicated thread stack, which is used during stack overflow (the call stack for the thread is full, so a separate stack region is required for operations such as allocating a new stack chunk) and underflow. The EBP register is reserved for holding the address of the thread stack.

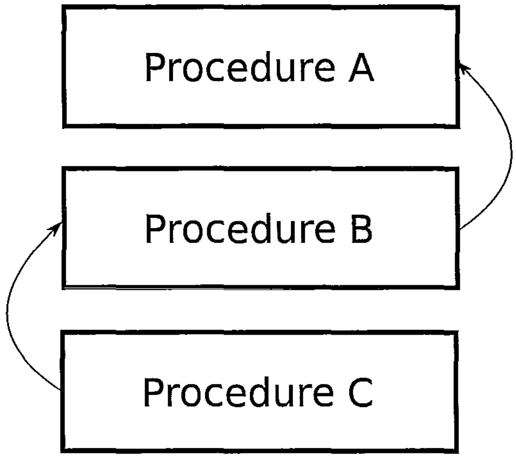


Figure 3.1: Per Procedure Heap Allocation Call Stack

Caller Instructions

The caller is responsible for creating and cleaning up a stack frame just large enough for the procedure.

Listing 3.4: Per Procedure Heap Allocation Caller Instructions

```
1  MOV  EAX, ESP
2  MOV  ESP, EBP
3  PUSH dword CALLEE_STACK_SIZE + 4
4  CALL STAMEX_OVERFLOW_HANDLER
5  PUSH arg1
6  ...
7  PUSH argn
8  CALL callee_name
9  SUB  ESP, CALLEE_STACK_SIZE - 4 - argumentsize
10 CALL STAMEX_UNDERFLOW_HANDLER
```

Annotations

Line 1 STAMEX_OVERFLOW_HANDLER requires that EAX contain the value of the caller's ESP during overflow. Additionally, since ESP might change before arguments are pushed to the stack at Line 5, EAX is used as a base pointer for accessing data in the caller's stack frame.

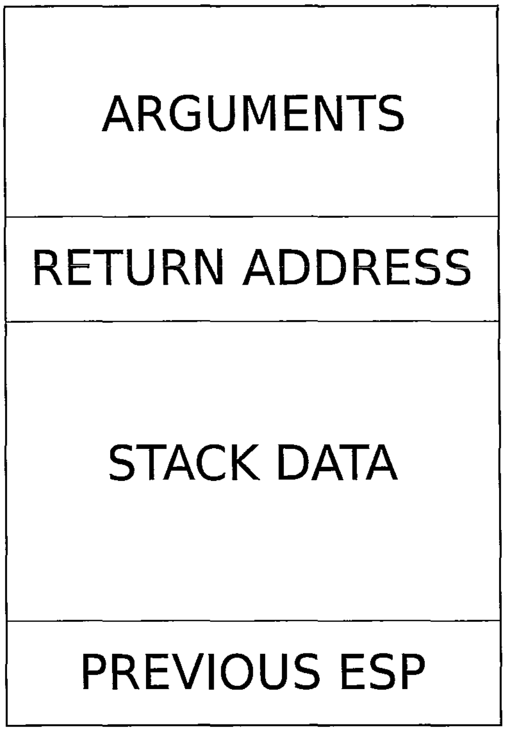


Figure 3.2: Per Procedure Heap Allocation Stack Frame

Line 2 STAMEX_OVERFLOW_HANDLER will use the thread stack

Line 3 As shown in Figure 3.2, the stack frame needs to hold the previous ESP, so CALLEE_STACK_SIZE + 4 is used

Line 4 When the overflow handler returns, ESP will be set to the top of the stack frame shown in Figure 3.2

Line 9 Align ESP to be 8 bytes away from the end of the stack frame, so there is enough room to store EIP without overwriting previous ESP

Line 10 On return, ESP will be restored to the value it contained in line 1 and the stack frame for callee_name will be freed

Callee Instructions

The callee is responsible for ensuring that the stack pointer has the same value on return from the procedure as it did on entry. This is accomplished by initially extending the stack by the amount of local stack space (stack space excluding arguments and return address) required by the procedure (S), and then ensuring that every RET instruction is prefixed by an instruction to decrease the stack by S.

Listing 3.5: Per Procedure Heap Allocation Callee Instructions

```
callee_name: SUB ESP, S
             ...              #Body of procedure
             ADD ESP, S
             RET
```

Thread Trampoline

When pthreads calls a function pointer as an entry point for a new thread, it expects that function to honour the cdecl calling convention. As outlined in Section 3.3.1, this is not the case. Additionally, the environment outlined in the start of this section must be set up. All this is accomplished with a trampoline procedure whose address is passed as the start_routine argument to pthread_create.

This pthreads compatible trampoline must accept one void * pointer as its argument. It receives a pointer to a struct of the following definition:

Listing 3.6: Per Procedure Heap Allocation Thread Trampoline Struct Definition

```
struct STAMEX_CALLBACK {
    void * arg;
    int fp;
    int fpStackSize;
};
```

The members of the struct are detailed below:

void * arg The argument being passed from the user procedure creating the thread

int fp The function pointer referring to the procedure that will serve as the entry point for the thread

int fpStackSize The required stack size for fp. This is generated by the compiler via the use of the C-- stacksizeof macro

The instructions for the thread trampoline follow:

Listing 3.7: Per Procedure Heap Allocation Thread Trampoline Instructions

```
PUSH EBX
PUSH ESI
PUSH EDI
PUSH EBP

MOV  ESI, [ESP + 20]

PUSH dword THREAD_STACK_SIZE
CALL malloc
ADD  EAX, THREAD_STACK_SIZE
MOV  EBP, EAX
ADD  ESP, 4

MOV  EBX, [ESI]
MOV  EDI, [ESI + 4]
PUSH ESI
MOV  ESI, [ESI + 8]
ADD  ESI, 8
CALL free
ADD  ESP, 4

MOV  EAX, ESP
MOV  ESP, EBP
PUSH ESI
CALL STAMEX_OVERFLOW_HANDLER

PUSH ESI
PUSH EBX
CALL EDI
MOV  ESI, [ESP + 4]

SUB  ESP, ESI
ADD  ESP, 16

```

```
35 CALL STAMEX_UNDERFLOW_HANDLER
36
37 MOV  EBX, EAX  //Store RET
38 SUB  EBP, THREAD_STACK_SIZE
39 PUSH EBP
40 CALL free
41 ADD  ESP, 4
42 MOV  EAX, EBX  //Store RET
43
44 POP  EBP
45 POP  EDI
46 POP  ESI
47 POP  EBX
48
49 RET
```

Annotations

Lines 1-4 Store non-volatile registers to be compatible with the cdecl calling convention

Line 6 Store the pointer to struct STAMEX_CALLBACK in the ESI register for later use

Lines 8-12 Allocate memory for the thread stack and store the top of the stack in the EBP register

Line 14 Store the member arg of struct STAMEX_CALLBACK in the EBX register

Line 15 Store the member fp of struct STAMEX_CALLBACK in the EDI register

Line 16 Store the address of struct STAMEX_CALLBACK on the stack in preparation for the memory to be freed

Line 17 Store the member fpStackSize of struct STAMEX_CALLBACK in ESI

Line 18 In addition to storing previous ESP as shown in Figure 3.2, the trampoline routine also needs to store the stack size of the entry procedure. Unlike caller instructions detailed in Section 3.4.2 where the stack size for the callee can be hardcoded, this thread trampoline is calling a function pointer. Therefore, the stack frame for the callee will be CALLEE_STACK_SIZE + 8 instead of the regular CALLEE_STACK_SIZE + 4 depicted in Figure 3.2

Line 19 The memory for struct STAMEX_CALLBACK can now be freed as the registers EBX, EDI and ESI now store all the struct members

Lines 22-25 Use the overflow routine as detailed in Section 3.4.2 to set up the stack frame for the callee

Line 27 Store the size of the stack frame allocated by STAMEX_OVERFLOW_HANDLER

Line 28 Push the callee's argument

Line 29 Call the callee

Line 30 Restore the size of the callee's stack frame to the ESI register

Lines 32-33 Align ESP to be 8 bytes away from the end of the stack frame, so there is enough room to store EIP without overwriting previous ESP

Line 35 Call STAMEX_UNDERFLOW_HANDLER with the same semantics as detailed in Section 3.4.2

Line 37 Store the return value in the non-volatile register EBX

Line 38 Store the start of the memory region that was returned by malloc for the thread stack in EBP

Lines 39-40 Free the memory that was allocated at line 9

Line 42 Restore the EAX register which holds the return value of the callee

Lines 44-47 Restore the non-volatile registers to be compatible with the cdecl calling convention

Stack Overflow

The stack overflow procedure is responsible for setting up the stack frame shown in Figure 3.2 for a given stack frame size.

Listing 3.8: Per Procedure Heap Allocation Thread Stack Overflow

```
1  PUSH ECX
2  PUSH EAX
3
4  PUSH dword [EBP - 4]
5  CALL malloc
6  ADD  ESP, 4
7
8  POP  EDX
9  POP  ECX
10
11 MOV  [EAX], EDX
12 ADD  EAX, [EBP - 4]
13
14 MOV  ESP, EAX
15 MOV  EAX, EDX
16 JMP  [EBP - 8]
```

Annotations

Line 1 Store the volatile register ECX on the stack so it is not modified by the malloc call on Line 5

Line 2 Store the caller's ESP on the thread stack. This procedure ensures that all registers except for EDX retain their original values at the end of this call

Line 4-5 Allocate a stack frame of the requested size

Line 8 Retreive the caller's ESP into EDX

Line 9 Restore the volatile register ECX

Line 11 Store the caller's ESP at the end of the callee's stack frame

Line 12 Store the top of the stack frame in EAX

Line 14 Set ESP to the top of the callee's stack frame

Line 15 Restore EAX to its initial value

Line 16 Return from this overflow procedure. However, the return address is stored on the thread stack, and ESP is already set to the value it should have after this procedure, so the JMP instruction is used instead of RET

Stack Underflow

The stack underflow procedure is responsible for restoring the ESP register to the previous ESP value stored in, and freeing, the stack frame shown in Figure 3.2. The stack underflow procedure expects ESP to be 4 bytes into the stack frame memory region that was allocated at Line 5 in Listing 3.8, with previous ESP at [ESP-4] and the return address for this procedure to be at [ESP].

Listing 3.9: Per Procedure Heap Allocation Thread Stack Underflow

```
1  MOV   EBX, EAX
2  MOV   ESI, [ESP - 4]
3  MOV   EDI, [ESP]
4
5  SUB   ESP, 4
6  MOV   [EBP-4], ESP
7  MOV   ESP, EBP
8  SUB   ESP, 4
9  CALL  free
10
11 MOV   EAX, EBX
12 MOV   ESP, ESI
13 JMP   EDI
```

Annotations

Line 1 Store the return value of the callee into the non-volatile register EBX

Line 2 Store the caller's ESP into the non-volatile ESI register

Line 3 Store the return address for this procedure into the non-volatile EDI register

Line 4 Store the start of the memory region that was allocated at Line 5 in Listing 3.8 into the ESP register

Line 5 Store the value of ESP at the top of the thread stack

Line 6 Set the current stack to the thread stack

Line 7 Adjust the current stack as if the instruction at Line 5 were a PUSH instruction

Line 8 Free the callee's thread stack

Line 10 Restore EAX

Line 11 Restore ESP to the caller's ESP

Line 12 Return from this underflow procedure. As the return address is stored in the EDI register, and ESP is already set to the value it should have after this procedure, the JMP instruction is used instead of RET

3.4.3 Linked Stack Chunks with Look-Ahead Overflow Detection

The call stack for a program is structured as a linked list of stack chunks. Unlike Per Procedure Heap Allocation where each procedure has a region of memory dynamically allocated by calling malloc [9] containing just one stack frame (see Figure 3.1), this mechanism employs the use of stack chunks which may contain many different stack frames, as depicted in Figure 3.3. When a procedure call would cause a stack chunk to overflow, a new stack chunk as depicted in Figure 3.4 is created and linked. The EBP register is reserved for maintaining a pointer to the book keeping information at the top of the current stack chunk. The stack overflow detection mechanism is an implementation of Capriccio's [26] call stack mechanism outlined in Section 2.4.

Caller Instructions

The instructions detailed here are those generated when the procedure call is a checkpoint. When the procedure call is not a checkpoint, the caller instructions are identical to those detailed in Section 3.4.1.

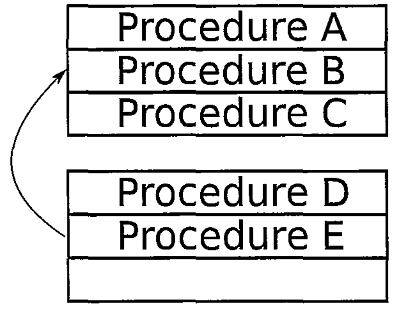


Figure 3.3: Linked Stack Chunks

Listing 3.10: Look-Ahead Overflow Detection Caller Instructions

```
1          MOV   EAX, ESP
2          MOV   EDX, ESP
3          ADD   EDX, ( STACK_CHUNK_SIZE
4                     - LONGEST_PATH( callee_name ) - 16 )
5          CMP   EDX, EBP
6          JGE   .L1
7          CALL  STAMEX_OVERFLOW_HANDLER
8    .L1:  PUSH  arg1
9          ...
10         PUSH  argn
11         CALL  callee_name
12         ADD   ESP, args_size
13         CMP   EBP, ESP
14         JNE   L2
15         CALL  STAMEX_UNDERFLOW_HANDLER
16   .L2:  ...
```

Annotations

Line 1 STAMEX_OVERFLOW_HANDLER requires that EAX contain the value of the callers ESP during overflow. Additionally, since ESP might change before arguments are pushed to the stack at Line 7, EAX is used as a base pointer for accessing data in the caller's stack frame

Line 2 Copy ESP into EDX in preparation for checking if stack overflow will occur

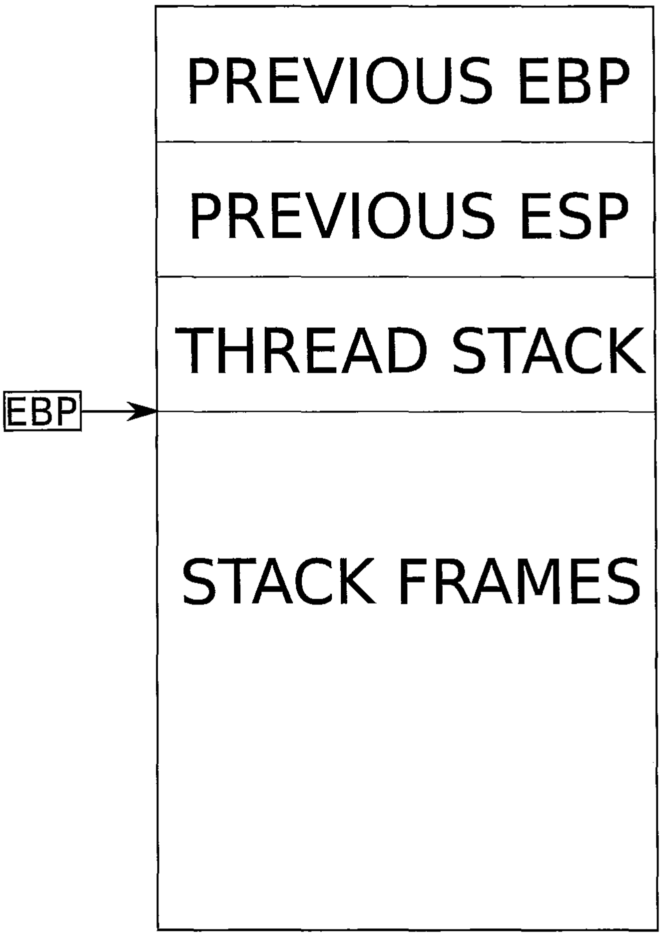


Figure 3.4: Stack Chunk for Look-Ahead Overflow Detection

Line 3 LONGEST_PATH(callee_name) is the maximum stack size that this checkpoint is reserving. The value 16 is used to adjust for previous EBP, previous ESP and the thread stack shown in Figure 3.4, as well as ensure that there are 4 bytes for the value of EIP during calls to STAMEX_OVERFLOW_HANDLER or STAMEX_UNDERFLOW_HANDLER

Lines 5-6 If overflow would occur, fall through to Line 6 and create a new stack chunk. Otherwise, jump to Line 7 and start pushing arguments

Lines 13-14 If stack underflow would occur, fall through to Line 15 and return to the previous stack chunk. Otherwise, jump to Line 16 and continue with execution of the caller

Callee Instructions

The callee is responsible for ensuring that the stack pointer has the same value on return from the procedure as it did on entry. This is accomplished by initially extending the stack by the amount of stack space required by the procedure (S), and then ensuring that every RET instruction is prefixed by an instruction to decrease the stack by S.

Listing 3.11: Look-Ahead Overflow Detection Callee Instructions

```
callee_name: SUB ESP, S
             ...           #Body of procedure
             ADD ESP, S
             RET
```

Thread Trampoline

The purpose of this thread trampoline is the same as the one explained in Section 3.4.2.

Listing 3.12: Look-Ahead Overflow Detection Thread Trampoline Struct Definition

```
struct STAMEX_CALLBACK {
    void * arg;
    int fp;
};
```

Notice that unlike the thread trampolines in Sections 3.4.2 and 3.4.4, there is no fpStackSize member. This is because this stack mechanism uses a fixed size stack chunk, and does not allow any checkpoint that would exceed the size of the fixed size stack chunk. Since this trampoline is setting up a new stack chunk, no overflow detection check that would require knowledge of the size of the checkpoint need be performed. The members of the struct are detailed below:

void * arg The argument being passed from the user procedure creating the thread

int fp The function pointer referring to the procedure that will serve as the entry point for the thread

The instructions for the thread trampoline follow:

Listing 3.13: Look-Ahead Overflow Detection Thread Trampoline Instructions

```
PUSH EBX
PUSH ESI
PUSH EDI
PUSH EBP

MOV  ESI, [ESP + 20]

PUSH STACK_CHUNK_SIZE
CALL malloc
MOV  EBX, EAX

MOV  dword [ESP], THREAD_STACK_SIZE
CALL malloc
MOV  [ESP], EAX
ADD  EAX, THREAD_STACK_SIZE

MOV  [EBX + STACK_CHUNK_SIZE - 4], EBP
MOV  [EBX + STACK_CHUNK_SIZE - 8], ESP
MOV  [EBX + STACK_CHUNK_SIZE - 12], EAX

ADD  EBX, STACK_CHUNK_SIZE - 12
MOV  EBP, EBX
MOV  ESP, EBX

PUSH dword [ESI]

PUSH ESI
MOV  ESI, [ESI + 4]
CALL free
ADD  ESP, 4

CALL ESI

CALL STAMEX_UNDERFLOW_HANDLER
```

```
35
36 MOV  ESI, EAX
37
38 CALL free
39 ADD  ESP, 4
40
41 MOV  EAX, ESI
42
43 POP  EBP
44 POP  EDI
45 POP  ESI
46 POP  EBX
47
48 RET
```

Annotations

Lines 1-4 Store volatile registers to be compatible with the cdecl calling convention

Line 6 Store the pointer to struct STAMEX_CALLBACK in the ESI register for later use

Lines 8-10 Allocate the initial stack chunk, and store it in EBX

Lines 12-14 Allocate the thread stack, and store its address on the stack

Line 15 Store the top of the thread stack in the EAX register

Line 17 Store previous EBP as depicted in Figure 3.4

Line 18 Store previous ESP as depicted in Figure 3.4

Line 19 Store a pointer to the thread stack as depicted in Figure 3.4

Lines 21-22 Calculate and store the pointer to the stack chunk's book keeping information in EBP

Line 23 Initialize ESP to the top of the stack chunk

Line 25 Push the callee's argument to the stack chunk

Line 27 Store the address of struct STAMEX_CALLBACK on the stack in preparation for the memory to be freed

Line 28 Store the member fp of struct STAMEX_CALLBACK in the ESI register

Line 29 Free the memory associated with struct STAMEX_CALLBACK

Line 32 Call the callee

Line 34 Call STAMEX_UNDERFLOW_HANDLER which will cleanup the stack chunk and restore the EBP and ESP registers

Line 36 Store the return value in the non-volatile register ESI

Line 38 Free the stack chunk pointer which was stored on the stack at Line 14

Line 41 Restore the EAX register which holds the return value of the callee

Lines 43-46 Restore the non-volatile registers to be compatible with the cdecl calling convention

Stack Overflow

The stack overflow procedure is responsible for setting up the stack chunk depicted in Figure 3.4.

Listing 3.14: Look-Ahead Overflow Detection Stack Overflow Instructions

```
MOV EDX, ESP
MOV ESP, [EBP]

PUSH EAX
PUSH ECX
PUSH EDX

PUSH STACK_CHUNK_SIZE
CALL malloc
ADD ESP, 4

MOV [EAX + STACK_CHUNK_SIZE - 4], EBP
```

```
13
14 MOV EDX, ESP
15 ADD EDX, 12
16 MOV [EAX + STACK_CHUNK_SIZE - 12], EDX
17
18 POP EDX
19 MOV EBP, EDX
20 ADD EDX, 4
21 MOV [EAX + STACK_CHUNK_SIZE - 8], EDX
22
23 MOV EDX, EAX
24 POP ECX
25 POP EAX
26
27 MOV ESP, EDX
28 ADD ESP, STACK_CHUNK_SIZE - 12
29
30 MOV EDX, EBP
31 MOV EBP, ESP
32 JMP [EDX]
```

Annotations

Line 2 Use the thread stack for creating a new stack chunk

Lines 4-6 Store volatile registers, as caller does not protect registers on stack overflow

Lines 8-10 Allocate a new stack chunk

Line 12 Store previous EBP as depicted in Figure 3.4

Lines 14-16 Store thread stack as depicted in Figure 3.4

Lines 18-19 Restore ESP as it was on entry to this procedure, and make a copy in EBP

Line 20 Calling this procedure pushed the return address to the stack - store the value of ESP before STAMEX_OVERFLOW_HANDLER was called in EDX

Line 21 Store previous ESP as depicted in Figure 3.4

Lines 23-25 EAX now contains the new stack pointer value - back it up and restore the volatile registers ECX and EAX

Lines 27-28 Set ESP to point to the top of the available stack chunk, right after the book keeping information depicted in Figure 3.4

Lines 30-32 Set EBP to point to the top of the available stack chunk, right after the book keeping information depicted in Figure 3.4, and return from this overflow procedure. The return address is not stored on the active stack, and ESP is already set to the value it should have after this procedure, so the JMP instruction is used instead of RET

Stack Underflow

The stack underflow procedure is responsible for restoring the previous stack chunk and freeing the existing stack chunk shown in Figure 3.4.

Listing 3.15: Look-Ahead Underflow Detection Stack Overflow Instructions

```
MOV ESI, [ESP]
MOV ESP, [EBP]

MOV EBX, [EBP + 8]
MOV EDI, [EBP + 4]

SUB EBP, ( STACK_CHUNK_SIZE - 12 )
PUSH EBP
MOV EBP, EBX
MOV EBX, EAX
CALL free
MOV ESP, EDI
MOV EAX, EBX
JMP ESI
```

Annotations

Line 1 Store the return address for this procedure in the non-volatile register ESI

Line 2 Use the thread stack for the underflow operation

Line 4 Store previous EBP depicted in Figure 3.4 in the non-volatile register EBX

Line 5 Store previous ESP depicted in Figure 3.4 in the non-volatile register EDI

Line 7 Store the start of the stack chunk's memory area in EBP

Line 8 Store the address of the stack chunk to free on the stack

Line 9 Restore EBP to the previous EBP value

Line 10 Store the return value of the last called user procedure in the non-volatile register EBX before calling free so that it is not lost

Line 11 Free the stack chunk that was pushed at Line 8

Line 12 Restore ESP to the previous ESP value

Line 13 Restore the return value of the last called user procedure

Line 14 Return from this underflow procedure. The return address is not stored on the active stack, and ESP is already set to the value it should have after this procedure, so JMP is used instead of RET

3.4.4 Linked Stack Chunks with MMU Overflow Detection

The call stack for a program is structured as a linked list of stack chunks, as depicted in Figure 3.3. On overflow, a new stack chunk as depicted in Figure 3.5 is created. The caller sequence is modified to ensure that the deepest region of memory that the callee will use is accessed first. If the accessed memory is beyond the available stack space, it will touch the guard page (a region of memory with no read or write access) and trigger the SIGSEGV signal. All SIGSEGV's are trapped and the signal handler performs stack extension for the thread from which the signal was raised. The C-- compiler assumes that the stack frame for a procedure is always smaller than the guard page.

Underflow is not explicitly detected. On creation of a new stack chunk the return address for the first procedure in the stack chunk is replaced with the address of the stack underflow procedure, and the return address is stored at the top of the stack chunk ('PROC A' and 'PROC A RETURN ADDR' in Figure 3.5). All other procedures in the stack chunk store their actual return address in the stack frame ('PROC B' in Figure 3.3). When the first procedure in the stack chunk returns, execution will continue with the underflow procedure, which will clean up the current stack chunk, reactivate the previous stack chunk, and continue with program execution.

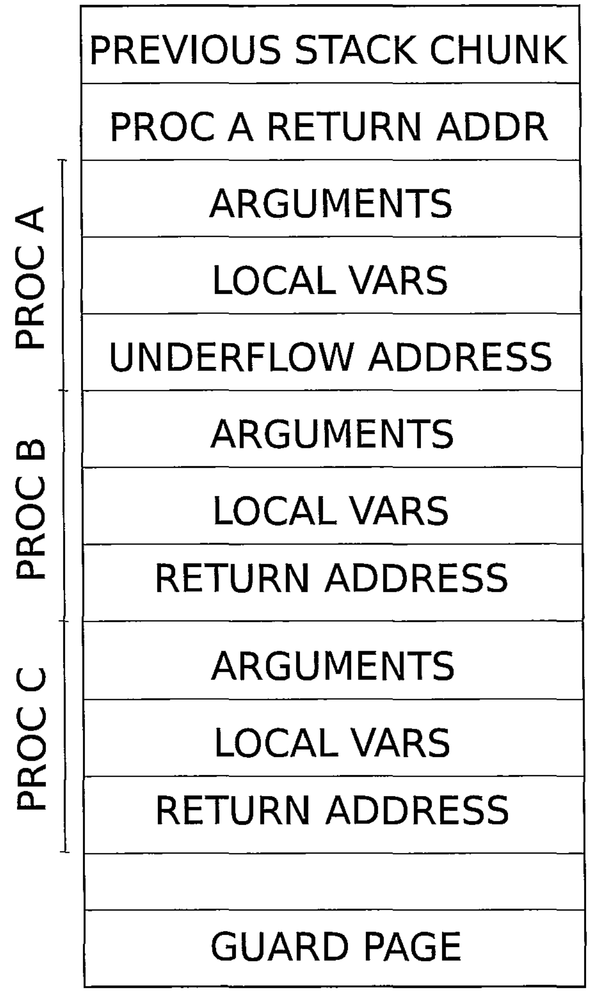


Figure 3.5: Stack Chunk for MMU Overflow Detection

Caller Instructions

As shown in Figure 3.5, the layout for a procedure differs from the C standard layout in that the return address is at the end of the stack instead of right after the arguments. As such, the caller instructions detailed in this section are only for calling other C-- procedures that adhere to this layout. In order to call external C functions that adhere to the C standard layout, a trampoline routine (detailed later in this section) is required.

To test if the existing stack chunk is able to hold the next procedure call, the return address is stored in the EDX register and an attempt is made to write that return address to the stack. If the write fails, a SIGSEGV is generated and the signal handler will create the new stack chunk, store the return address at the top of the stack chunk and place the address of the stack underflow procedure in the EDX register. On return of the signal handler, control continues with the instruction that caused the signal (the instruction that attempts to write the return address) and the address of the stack underflow procedure will be written in place of the return address.

Listing 3.16: MMU Overflow Detection Caller Instructions

```
1                MOV   EAX, ESP
2                MOV   EDX, return_label
3                MOV   [ESP - CALLEE_STACK_SIZE], EDX
4                PUSH  arg1
5                ...
6                PUSH  argn
7                JMP   callee_name
8  return_label: ...
```

Annotations

Line 1 The signal handler requires that EAX contain the value of the caller's ESP during overflow. Additionally, since ESP might change before arguments are pushed to the stack at Line 4, EAX is used as a base pointer for accessing data in the caller's stack frame

Line 2 Store the return address in the EDX register

Line 3 Attempt to write the return address to the stack. This instruction may be called again if it causes a SIGSEGV

Line 7 Since the return address was already written at Line 3, use JMP instead of CALL to begin execution of the callee

Callee Instructions

The callee is responsible for ensuring that the entire stack frame, including arguments, is clean before returning. This deviation from the standard C calling convention is required to handle the case when the return address is the stack underflow address, as the stack underflow procedure requires the stack pointer to be at the top of the stack upon entry.

Listing 3.17: MMU Overflow Detection Callee Instructions

```
1 callee_name: SUB ESP, LOCAL_STACK_SIZE
2              ... #Body of procedure
3              ADD ESP, COMPLETE_STACK_SIZE
4              JMP [ESP - COMPLETE_STACK_SIZE]
```

Annotations

Line 1 The caller will have moved the stack pointer down by args_size during argument PUSH. Move the stack pointer to the end of the stack frame

Line 3 Move the stack pointer to the top of the stack before returning

Line 4 Since the stack pointer was moved at Line 3 and is no longer pointing at the return address (and should not be moved any further), the RET instruction is not applicable. Continue program execution at the return address using a JMP instruction

Extern Call

When calling a procedure that uses the C standard calling convention, the caller sets up the stack frame depicted in Figure 3.6 after ensuring that there is enough stack space for the external procedure. The extern trampoline then stores ARGS SIZE in a non-volatile register and calls the external procedure call overwriting ARGS SIZE and

EXTERN FUNCTION PTR as depicted in Figure 3.7. After the external procedure call has returned, the extern trampoline returns ensuring the conditions described in Section 3.4.4 are met. Without this extern trampoline the C standard calling convention would return with the stack pointer not at the top of the callee's stack frame, but rather underneath the callee's arguments.

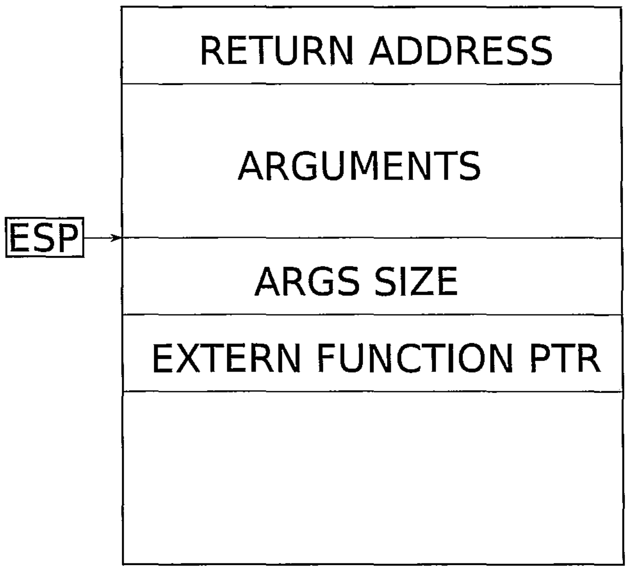


Figure 3.6: MMU Overflow Detection Extern Trampoline Pre-Call

Listing 3.18: MMU Overflow Detection Extern Caller Instructions

```
              MOV  EAX, ESP
              MOV  EDX, return_label
              CMP  [ESP - EXTERN_SIZE], EAX
              PUSH EDX
              PUSH arg1
              ...
              PUSH argn
              MOV [ESP - 4], args_size
              MOV [ESP - 8], extern_function_ptr
              JMP  extern_trampoline
return_label: ...
```

Annotations

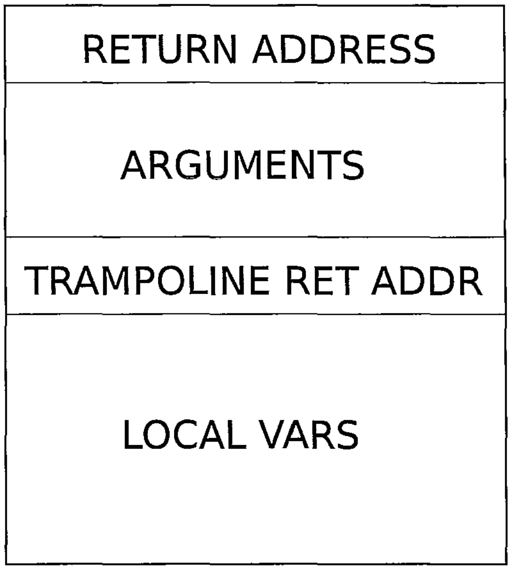


Figure 3.7: MMU Overflow Detection Extern Trampoline Post-Call

Line 1 The signal handler requires that EAX contain the value of the callers ESP during overflow. Additionally, since ESP might change before arguments are pushed to the stack at Line 4, EAX is used as a base pointer for accessing data in the caller's stack frame

Line 2 Store the return address in the EDX register

Line 3 Since the exact stack size of the external procedure is not known (and may be variable), a constant EXTERN_SIZE is used as a "large enough" stack size. A CMP instruction is used to access the memory without modifying the contents of any registers or memory. Should the access be invalid, a SIGSEGV will be generated and a stack extension will occur, and this instruction will be executed again

Line 4 Store the return address (which may be the actual return address or the address of the stack underflow procedure) in the location depicted in Figure 3.6

Lines 5-7 Push the extern procedure's arguments to the stack

Line 8 Store the total size of the arguments from Lines 5-7 on the stack without modifying the stack pointer, as depicted in Figure 3.6

Line 9 Store the address of the external procedure on the stack without modifying the stack pointer, as depicted in Figure 3.6

Line 10 Begin execution of the external trampoline procedure. Use a JMP instead of a CALL instruction since the return address is already stored on the stack and the stack pointer should not be modified

Listing 3.19: MMU Overflow Detection Extern Trampoline Instructions

```
1 MOV  EBP, [ESP - 4]
2 CALL [ESP - 8]
3 ADD  ESP, EBP
4 RET
```

Annotations

Line 1 Store ARGS SIZE in the non-volatile register EBP

Line 2 Call the external procedure, overwriting ARGS SIZE and EXTERN FUNCTION PTR, transitioning from Figure 3.6 to Figure 3.7

Line 3 The external procedure has returned - move the stack pointer to the top of the stack frame, just under RETURN ADDRESS depicted in Figure 3.7

Line 4 Return from this trampoline, either directly to the caller or to the stack underflow procedure

Thread Trampoline

The purpose of this thread trampoline is the same as the one explained in Section 3.4.2.

Listing 3.20: MMU Overflow Detection Thread Trampoline Struct Definition

```
struct STAMEX_CALLBACK {
    void * arg;
    int fp;
    int fpStackSize;
};
```

The members of the struct are detailed below:

void * arg The argument being passed from the user procedure creating the thread

int fp The function pointer referring to the procedure that will serve as the entry point for the thread

int fpStackSize The required stack size for fp. This is generated by the compiler via the use of the C-- stacksizeof macro

The instructions for the thread trampoline follow:

Listing 3.21: MMU Overflow Detection Thread Trampoline Instructions

```
              PUSH EBX
              PUSH ESI
              PUSH EDI
              PUSH EBP

              CALL STAMEX_SETUP_SIGNAL_STACK

              MOV  EBP, [ESP+20]

              PUSH dword [EBP]
              PUSH dword [EBP+8]

              PUSH EBP
              MOV  EBP, [EBP+4]
              CALL free

              ADD  ESP, 8

              MOV  EAX, ESP
              SUB  EAX, [ESP-4]
              ADD  EAX, 4
              MOV  dword [EAX], return_label

              JMP  EBP
return_label: MOV  EBP, EAX
              CALL STAMEX_TEARDOWN_SIGNAL_STACK
```

```
27        MOV   EAX, EBP
28
29        POP   EBP
30        POP   EDI
31        POP   ESI
32        POP   EBX
33        RET
```

Annotations

Lines 1-4 Store volatile registers to be compatible with the cdecl calling convention

Line 6 Set up the stack that the signal handler will use

Line 8 Store the pointer for struct STAMEX_CALLBACK in the non-volatile register EBP

Line 10 Store the member arg of struct STAMEX_CALLBACK to the stack

Line 11 Store the member fpStackSize of struct STAMEX_CALLBACK to the stack

Line 13 Store the address for struct STAMEX_CALLBACK to the stack in preparation for freeing the memory

Line 14 Store the member fp of struct STAMEX_CALLBACK in the non-volatile register EBP

Line 15 Free the memory associated with struct STAMEX_CALLBACK

Line 17 Align ESP directly under arg on the stack, in preparation for calling fp

Lines 19-22 Store the return address at the end of the callee's stack frame (as depicted in Figure 3.5) without moving ESP from it's current position

Line 24 Call the thread entry routine which was stored in the EBP register at Line 14

Line 25 After the thread entry routine is finished, store the return value in the non volatile register EBP

Line 26 Clean up the signal stack that was set up at Line 6

Line 27 Store the return value for this function

Lines 28-32 Restore the volatile registers

Signal Stack Setup

A C function used by the thread trampoline that uses sigaltstack to set up a thread's signal stack.

Listing 3.22: MMU Overflow Detection Signal Stack Setup

```
void STAMEX_SETUP_SIGNAL_STACK() {
    stack_t s;
    s.ss_sp = malloc(SIGSTKSZ);
    s.ss_size = SIGSTKSZ;
    s.ss_flags = 0;
    if (sigaltstack(&s, NULL) < 0)
        perror("sigaltstack()");
}
```

Signal Stack Teardown

A C function used by the thread trampoline to clean up a thread's signal stack.

Listing 3.23: MMU Overflow Detection Signal Stack Teardown

```
void STAMEX_TEARDOWN_SIGNAL_STACK() {
    stack_t s;

    if (sigaltstack(NULL, &s) < 0)
        perror("sigaltstack()");

    free( s.ss_sp );

    s.ss_flags = SS_DISABLE;
}
```

Stack Overflow

The stack overflow procedure is invoked when the SIGSEGV signal is generated, and is responsible for setting up and activating the stack chunk depicted in Figure 3.5.

Listing 3.24: MMU Overflow Detection Stack Overflow

```
1  void STAMEX_OVERFLOW_HANDLER( int signum
2                              , siginfo_t * siginfo
3                              , void * ucontext ) {
4      ucontext_t * ut = (ucontext_t *)ucontext;
5      void * stack;
6
7      stack = STAMEX_STACK_ALLOC(
8            ut->uc_mcontext.gregs[REG_EDX]
9          , ut->uc_mcontext.gregs[REG_ESP] );
10
11     ut->uc_mcontext.gregs[REG_EDX]
12         = (int)&STAMEX_UNDERFLOW_HANDLER;
13     ut->uc_mcontext.gregs[REG_ESP] = (int)stack;
14 }
```

Annotations

Line 7 Set up the stack chunk depicted in Figure 3.5

Line 11 Store the address of the stack underflow procedure in the EDX register. On return from this procedure, the caller will write the contents of EDX register into the return address location as depicted in 'PROC A' of Figure 3.5

Line 13 Store the address of the new stack chunk in ESP

Stack Chunk Allocation

This stack allocation procedure allocates the stack depicted in Figure 3.5.

Listing 3.25: MMU Overflow Detection Stack Chunk Allocation

```
void * STAMEX_STACK_ALLOC( int prevResume
```

```
2                                    , int prevESP ) {
3     void * stack;
4
5     if ( posix_memalign( &stack , STAMEX_PAGE_SIZE
6                        , STAMEX_STACK_SIZE
7                        + STAMEX_PAGE_SIZE
8                        ) != 0 ) {
9         fprintf( stderr , "Failed␣to␣allocate␣stack\n" );
10    }
11
12    if ( mprotect( stack , STAMEX_PAGE_SIZE
13                 , PROT_NONE ) != 0 ) {
14        perror( "Failed␣to␣set␣up␣guard␣page␣for␣stack\n" );
15    }
16
17    *((int *)(stack + STAMEX_STACK_SIZE + STAMEX_PAGE_SIZE
18                     - sizeof(int))) = prevESP;
19    *((int*)(stack + STAMEX_STACK_SIZE + STAMEX_PAGE_SIZE
20                     - sizeof(int)*2)) = prevResume;
21
22    return stack + STAMEX_STACK_SIZE + STAMEX_PAGE_SIZE
23                 - sizeof(int)*2;
24 }
```

Annotations

Line 5 Allocate a memory aligned stack chunk with a guard page

Line 12 Disallow any memory access to the guard page of the stack, ensuring that any attempt at memory access will cause a SIGSEGV

Line 17 Store 'PREVIOUS STACK CHUNK' depicted in Figure 3.5

Line 19 Store the return address for the caller (depicted as 'PROC A RETURN ADDR' in Figure 3.5)

Line 22 Return the address of the top of the stack, right after the book keeping information 'PREVIOUS STACK CHUNK' and 'PROC A RETURN ADDR' depicted in Figure 3.5

Stack Underflow

The stack underflow procedure is responsible for restoring the previous stack chunk and freeing the existing stack chunk shown in Figure 3.5. This procedure is not called, but rather jumped to when the first procedure in a stack chunk returns.

Listing 3.26: MMU Overflow Detection Stack Underflow

```
1 MOV EBP, EAX
2
3 CALL STAMEX_GET_SIGNAL_STACK
4 MOV EDX, ESP
5 MOV ESP, EAX
6 MOV EAX, EDX
7
8 ADD EAX, 8
9 PUSH dword [EAX - 4]
10 PUSH dword [EAX - 8]
11
12 PUSH EAX
13 CALL STAMEX_STACK_FREE
14
15 MOV EAX, EBP
16
17 MOV EBX, ESP
18 MOV ESP, [EBX + 8]
19 JMP [EBX + 4]
```

Annotations

Line 1 Store the return value of the procedure at the top of the stack chunk in the non-volatile register EBP

Lines 3-6 Use the signal stack as the stack space when cleaning up the existing stack chunk. Store the stack chunk's pointer in EAX. The STAMEX_GET_SIGNAL_STACK procedure is detailed in Listing 3.28

Line 8 Store the top of the stack chunk in the EAX register, moving past the book keeping information 'PREVIOUS STACK CHUNK' and 'PROC A RETURN ADDR' depicted in Figure 3.5

Line 9 Store 'PREVIOUS STACK CHUNK' depicted in Figure 3.5 to the stack

Line 10 Store 'PROC A RETURN ADDR' depicted in Figure 3.5 to the stack

Lines 12-13 Free the stack chunk. The STAMEX_STACK_FREE procedure is detailed in Listing 3.27

Line 15 Restore the return value of the returning procedure

Lines 17-18 Set ESP to the previous stack chunk, and store the temporary working stack in EBX

Line 19 Continue execution with the caller of the procedure that triggered this stack underflow

Listing 3.27: MMU Overflow Detection Stack Chunk Free

```
void STAMEX_STACK_FREE( void * stack ) {
    stack = stack - STAMEX_STACK_SIZE - STAMEX_PAGE_SIZE;

    if ( mprotect( stack, STAMEX_PAGE_SIZE
                   , PROT_READ | PROT_WRITE ) != 0 ) {
        fprintf( stderr, "Failed to disable guard page\n" );
    }

    free( stack );
}
```

Annotations

Line 2 Store the start of the stack's memory address in the stack variable

Line 4 Remove the memory protection from the guard page

Line 8 Free the stack chunk

Listing 3.28: MMU Overflow Detection Get Signal Stack

```
1  void * STAMEX_GET_SIGNAL_STACK( ) {
2      void * ret;
3      stack_t s;
4
5      if ( sigaltstack(NULL, &s) < 0 ) {
6          perror( "sigaltstack()" );
7      }
8
9      ret = s.ss_sp;
10     ret += s.ss_size;
11
12     return ret;
13 }
```

Annotations

Line 5 Retrieve the start of the signal stack's memory

Lines 9-10 Store the top of the signal stack in the ret variable

Line 12 Return the top of the signal stack

3.5 Experiments

Three C-- programs were used to compare the stack mechanisms detailed in Section 3.4. The full listings for these C-- programs are in Appendix A. All of these C-- programs can be compiled with gcc using the command line arguments "-DNULL=0 -Dstacksizeof(x)=0".

All experiments were run on the following machine:

CPU Intel Core i7 940. Contains 4 cores, with each core containing Hyper-Threading Technology.

Memory 3GB of DDR 3 memory

Operating System 32-bit Gentoo Linux using gcc version 4.3.4

Each experiment has a single-threaded and multi-threaded version. Each multi-threaded version has two variations: The "cores" variation tests one to eight threads to test scalability over four individual cores, as well as Intel's® Hyper-Threading Technology ("Hyper-Threading Technology delivers two logical processors that can execute different tasks simultaneously using shared hardware resources" [16]). A "quantity" variation tests scalability across a number of threads which greatly exceeds available cores in the system. For the Linked Stack Chunk experiments (detailed in Sections 3.4.3 and 3.4.4), the stack chunk size was 8 pages or 32 kilobytes, excluding the space for the guard page if applicable.

3.5.1 Summation

This program sums the numbers from 1 to n recursively, as shown in Listing 3.29.

Listing 3.29: Summation Snippet

```
int summation( int n ) {
    int ret;

    if ( n == 0 ) {
        return 0;
    }

    ret = n + summation( n - 1 );

    return ret;
}
```

This experiment aims to magnify the procedure calling overhead of the various stack implementations by calling a heavily recursive procedure that contains a minimum of computation. In the multi-threaded version of this experiment, each thread sums the numbers from 1 to (n div number_of_threads). The "cores" variation was run with 1, 2, 3, 4, 5, 6, 7 and 8 threads, and the "quantity" variation was run with 8, 32, 64, 128, 256, 512 and 1024 threads.

3.5.2 Unbalanced Binary Tree

This experiment is an implementation of a simple binary tree. The tree itself is a balanced binary tree of integers that is 20 levels deep, and an unbalanced branch of 1 million integers as depicted in Figure 3.8. 70% of the time the program will search for a random integer contained within the 20 level deep balanced portion of the binary tree. 30% of the time the program will search for the maximum value in the binary tree, triggering a spike in stack usage.

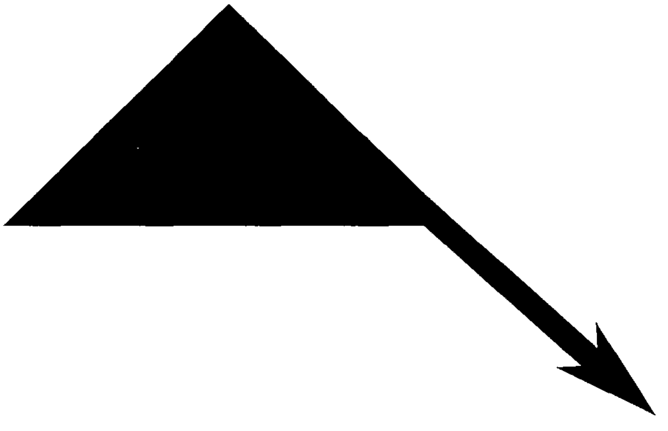


Figure 3.8: Unbalanced Binary Tree

This experiment aims to test performance of the various stack mechanisms in an environment that traditional stack mechanisms have difficulty performing under: a large number of highly variable sized stacks. The multi-threaded version of this experiment keeps the work per thread constant (100 searches) as the number of threads increase. The "cores" variation was run with 1, 2, 3, 4, 5, 6, 7 and 8 threads, and the "quantity" variation was run with 8, 16, 32 and 64 threads.

3.5.3 "Real World"

The goal of this thesis is to discover a stack mechanism that works as well as the existing stack mechanism for existing programs, but also allows for usage patterns that are difficult for the traditional stack mechanism. As the C-- compiler is primitive, it was difficult to find existing C code that would compile without significant modification. Instead, a simple set of functions with the call graph depicted in Figure 3.9 was written.

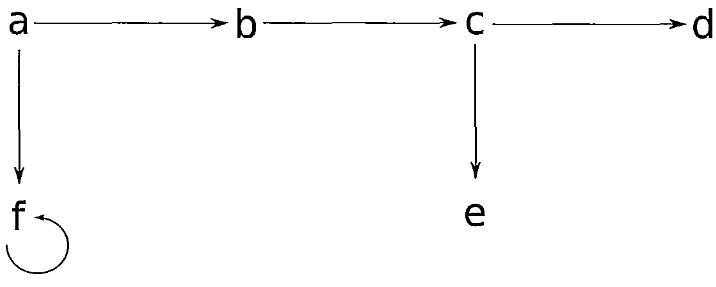


Figure 3.9: Real World

Every procedure performed work in the form of a simple for loop that incremented an integer, as shown in Listing 3.30. In an attempt to mirror common program behaviour, non-recursive calls did more work (10,000,000 units) than recursive calls (10,000 units). The amount of recursion was limited to a relatively shallow 100 recursive calls. The single-threaded version of this experiment altered the overall number of times the procedure a was called, while the multi-threaded version of this experiment kept the overall iterations at 10, but increased the number of threads that were run keeping the work per thread constant. The "cores" variation was run with 1, 2, 3, 4, 5, 6, 7 and 8 threads, and the "quantity" variation was run with 8, 32, 64, 128, 256 and 512 threads.

Listing 3.30: Real World Snippet

```
for ( i = 0; i < units; i = i + 1 ) {
      x = x + 1;
}
```

Chapter 4

Results

All figures in this section use a legend with shortened versions of the stack mechanism names given in Section 3.4. The shortened versions follow:

Heap Per Procedure Heap Allocation, Section 3.4.2.

Look-Ahead Linked Stack Chunks with Look-Ahead Overflow Detection, Section 3.4.3.

MMU Linked Stack Chunks with MMU Overflow Detection, Section 3.4.4.

Traditional Traditional Fixed-Size Stack, Section 3.4.1.

Per Procedure Heap Allocation (Section 3.4.2) was not compatible with the pthreads library, and was therefore not included in any of the multi-threaded tests.

All data points in the following figures are averages over 30 runs of the experiment. The confidence limits for a 95% confidence interval were so small that they were nearly indistinguishable when added to the following figures. Appendix B contains the raw data for the tests along with confidence values.

4.1 Summation

4.1.1 Single-Threaded

As can be seen in Figure 4.1, the overhead from a dynamic memory allocation call (malloc) for every procedure with the Heap mechanism is very significant. Viewing the same results on a smaller scale omitting the Heap mechanism in Figure 4.2 allows for better analysis of the remaining methods. Comparing Traditional to gcc and gcc -O2 shows that C--'s code generation in terms of performance for this simple procedure is somewhere between non-optimized gcc and optimized gcc. Using the Traditional mechanism as the performance baseline (the traditional fixed-size stack implemented in C--), Look-Ahead does not appear to add any significant overhead. The MMU mechanism outperforms the Traditional mechanism. The only plausible explanation so far theorized for this is that the use of JMP instruction for procedure calls is cheaper than the use of the CALL instruction for this usage pattern.

The stack mechanism implemented with gcc only experimented summing up to 50 million. On the larger sums, gcc ran out of stack space even with the stack size set to unlimited via bash's builtin shell command ulimit. All the stack mechanisms implemented in C-- and gcc -O2 were able to optimize stack usage such that summations of up to 100 million were possible.

4.1.2 Multi-Threaded "Cores"

Performance scaled mostly linearly up to four cores in the test machine for the Traditional and Look-Ahead mechanisms, with the overhead from the Look-Ahead mechanisms visible in Figure 4.3. While scalability for these two mechanisms was not perfectly linear, it did follow the general trend shown by gcc -O2. As noted in Section 4.1.1 gcc stack space utilization was not as efficient as C-- or gcc -O2, and was omitted from this test rather than run the test on a smaller scale.

The MMU mechanism, while starting out with better performance than the Traditional or Look-Ahead mechanisms demonstrated the worst scalability, and eventually the worst performance, as the number of threads exceeded the number of available cores. It appears this is due to the demultiplexing mechanism implemented in pthreads - a signal is only delivered to a process, and the search for the process's thread id is a linear search as can be seen in the source code for the pthreads library in Listing 4.1. However, further research would be required to gain a conclusive answer.

Listing 4.1: pthread_find_self

```
pthread_descr __pthread_find_self(void)
{
  char * sp = CURRENT_STACK_FRAME;
  pthread_handle h;
```

```
  /* __pthread_handles[0] is the initial thread,
     __pthread_handles[1] is the manager threads
     handled specially in thread_self(), so start
     at 2 */
  h = __pthread_handles + 2;
# ifdef _STACK_GROWS_UP
  while (! (sp >= (char *) h->h_descr
         && sp < h->h_descr->p_guardaddr)) h++;
# else
  while (! (sp <= (char *) h->h_descr
         && sp >= h->h_bottom)) h++;
# endif
  return h->h_descr;
}
```

4.1.3 Multi-Threaded "Quantity"

Trends observed in Section 4.1.2 were magnified in this experiment as can be seen in Figure 4.4. The Look-Ahead mechanism demonstrated worse performance compared to the Traditional mechanism, and the MMU mechanism continued to show the worst scalability of all the tested methods. All methods showed initial degradation in scalability that appeared to level off around 500 threads.

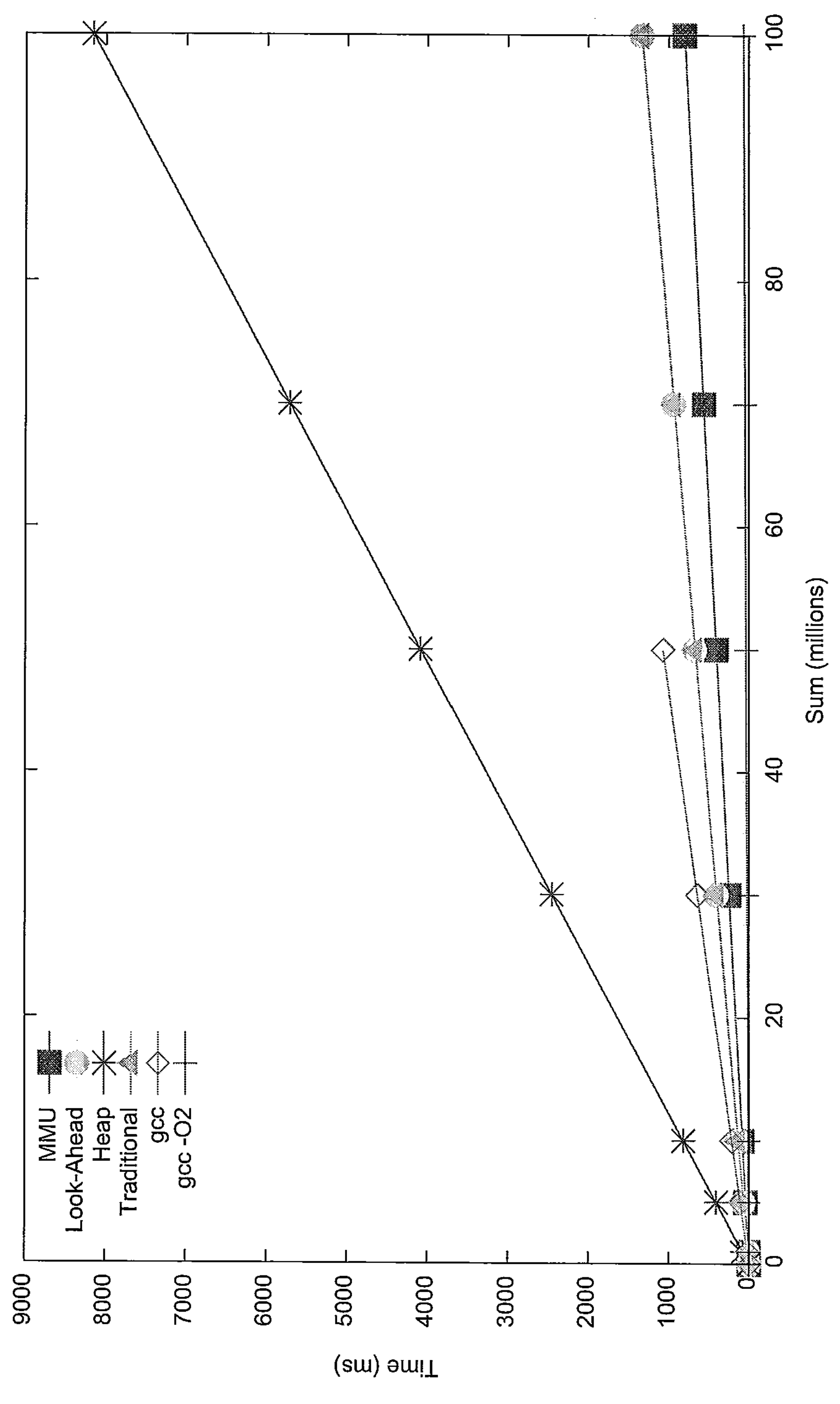


Figure 4.1: Summation Single-Threaded All

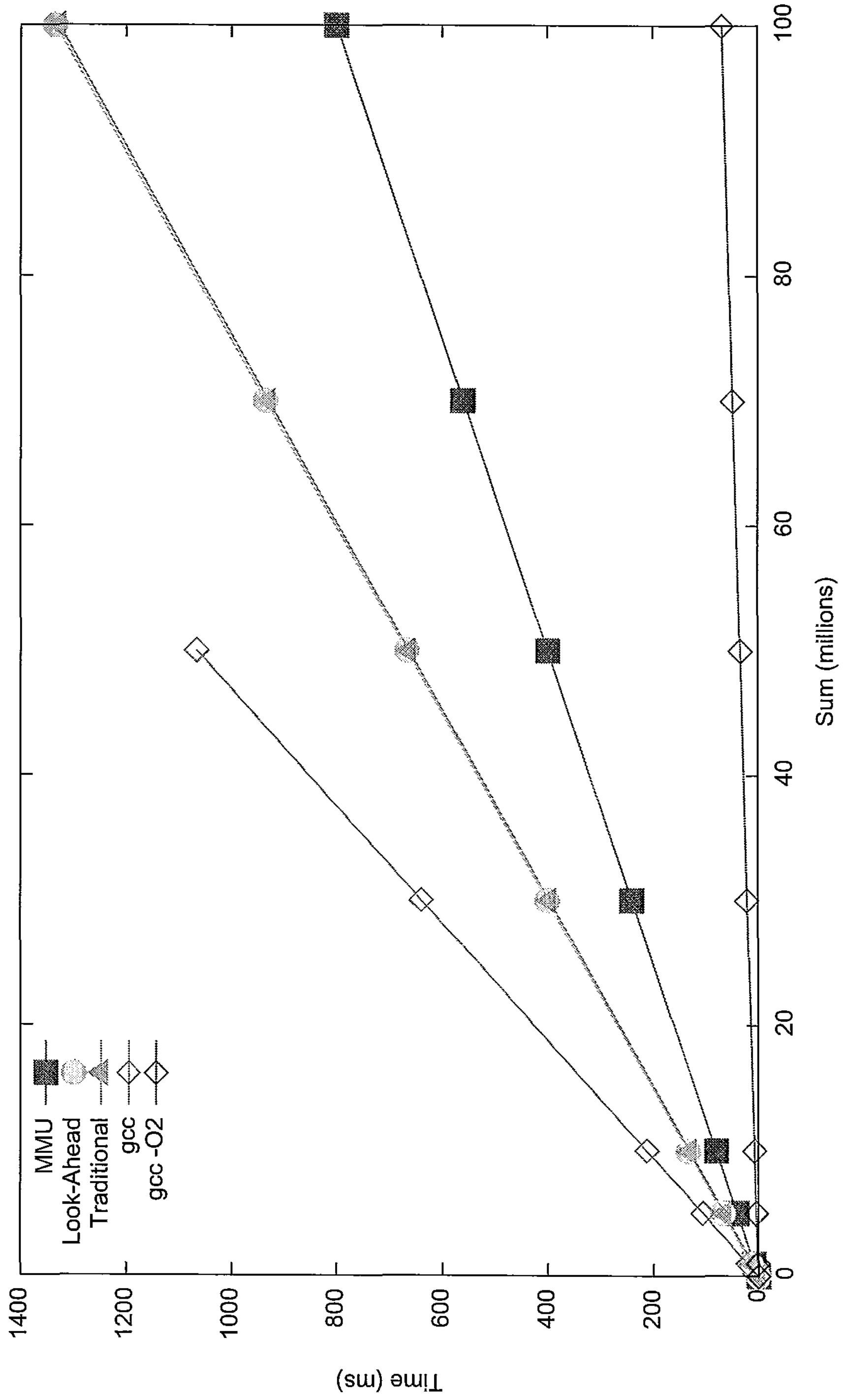


Figure 4.2: Summation Single-Threaded No Heap

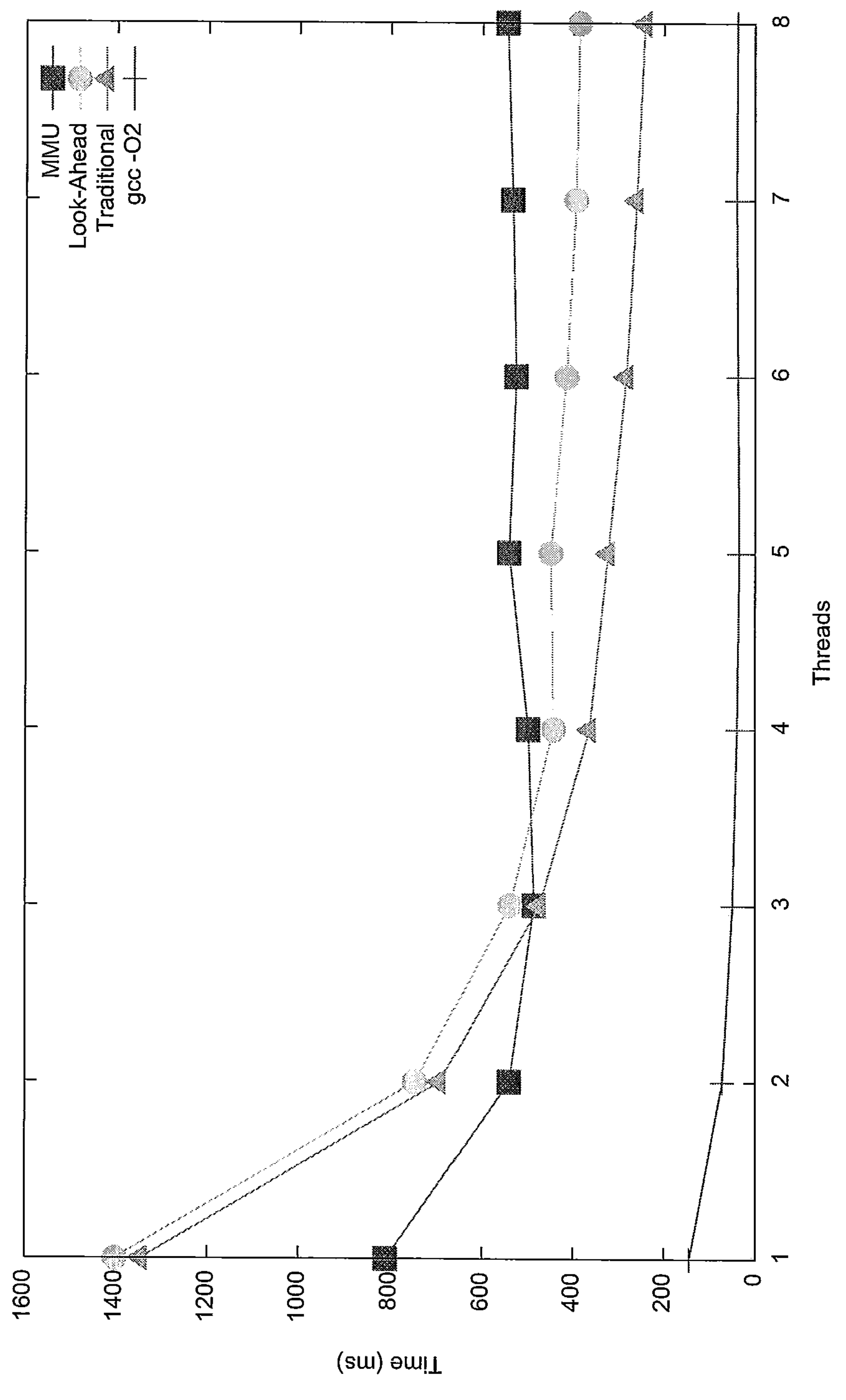


Figure 4.3: Summation Multi-Threaded Cores

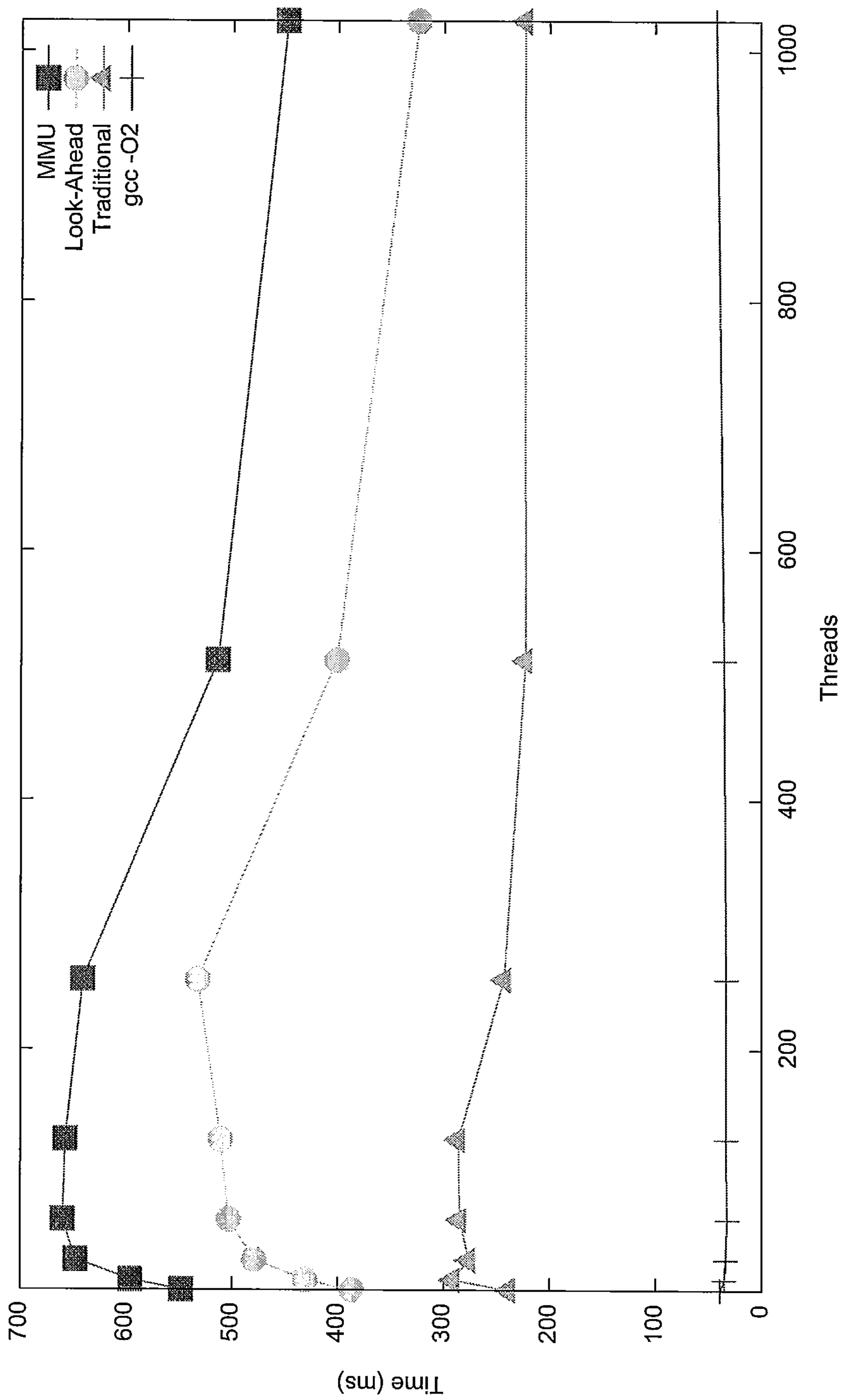


Figure 4.4: Summation Multi-Threaded Quantity

4.2 Unbalanced Binary Tree

4.2.1 Single-Threaded

As can be seen in Figure 4.5, the overhead from a malloc call for every procedure with the Heap mechanism continues to be very significant. Viewing the same results on a smaller scale omitting the Heap mechanism in Figure 4.6 allows for better analysis of the remaining methods. The trends observed in Section 4.1.1 continue to hold, which is not surprising given that this experiment is very similar (a heavily recursive procedure dominates the runtime for this test).

4.2.2 Multi-Threaded "Cores"

The Look-Ahead mechanism scaled identically to the Traditional mechanism, with overhead clearly visible in Figure 4.7. The MMU mechanism continued to show the poorest scaling (as discussed in Section 4.1.2), with the unexplained valley for 6 and 7 threads.

4.2.3 Multi-Threaded "Quantity"

None of the Traditional stack mechanisms were tested in this experiment, as the high concurrency combined with the tendency for threads to spike in their stack usage meant that a fixed size stack mechanism would not be able to share memory efficiently enough to run this experiment. As such, only the MMU and Look-Ahead mechanisms are visible in Figure 4.8. In this test, the MMU mechanism continued to show the poorest scaling (as discussed in Section 4.1.2), while the Look-Ahead mechanism continued to scale in a linear fashion.

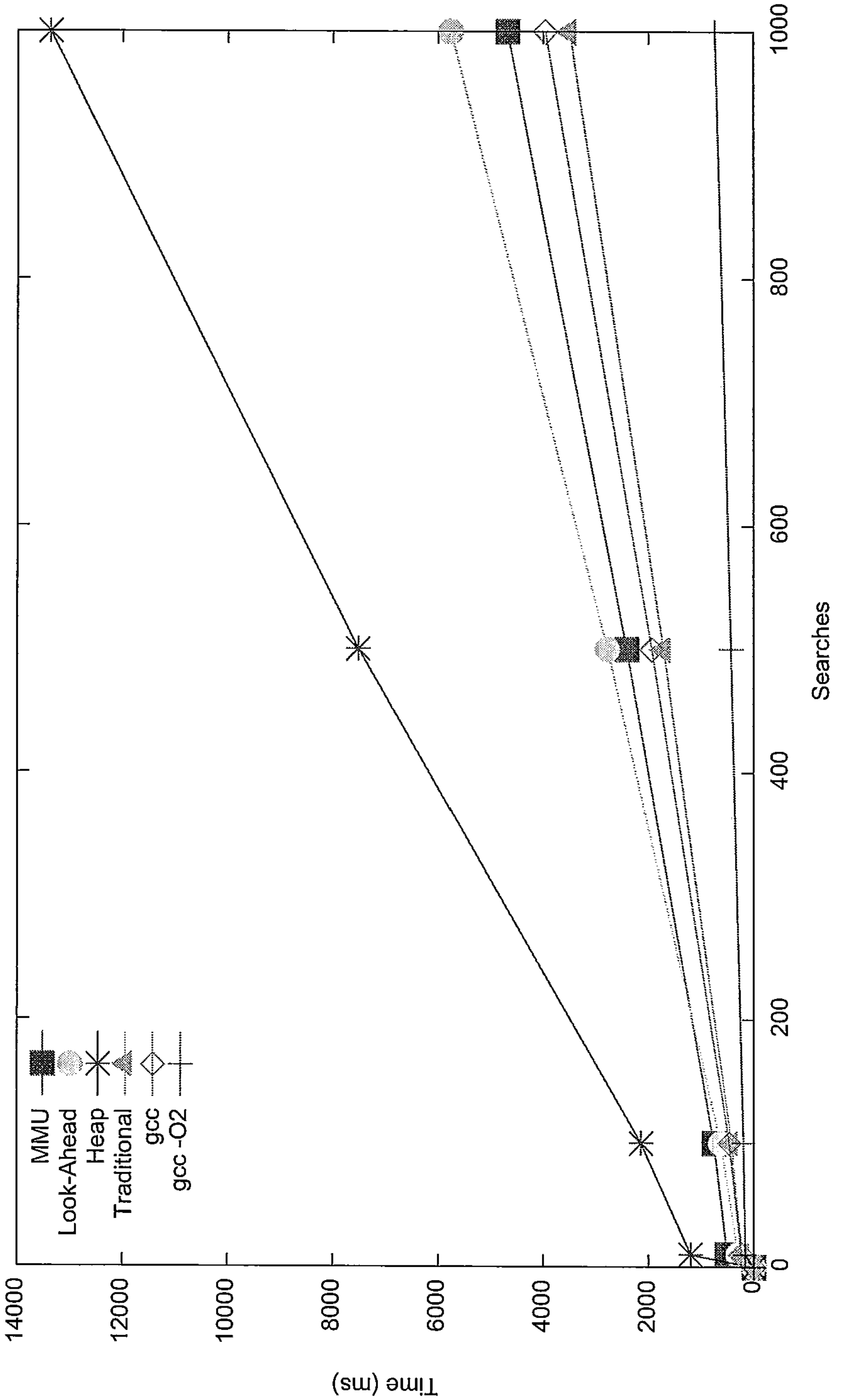


Figure 4.5: Unbalanced Binary Tree Single-Threaded All

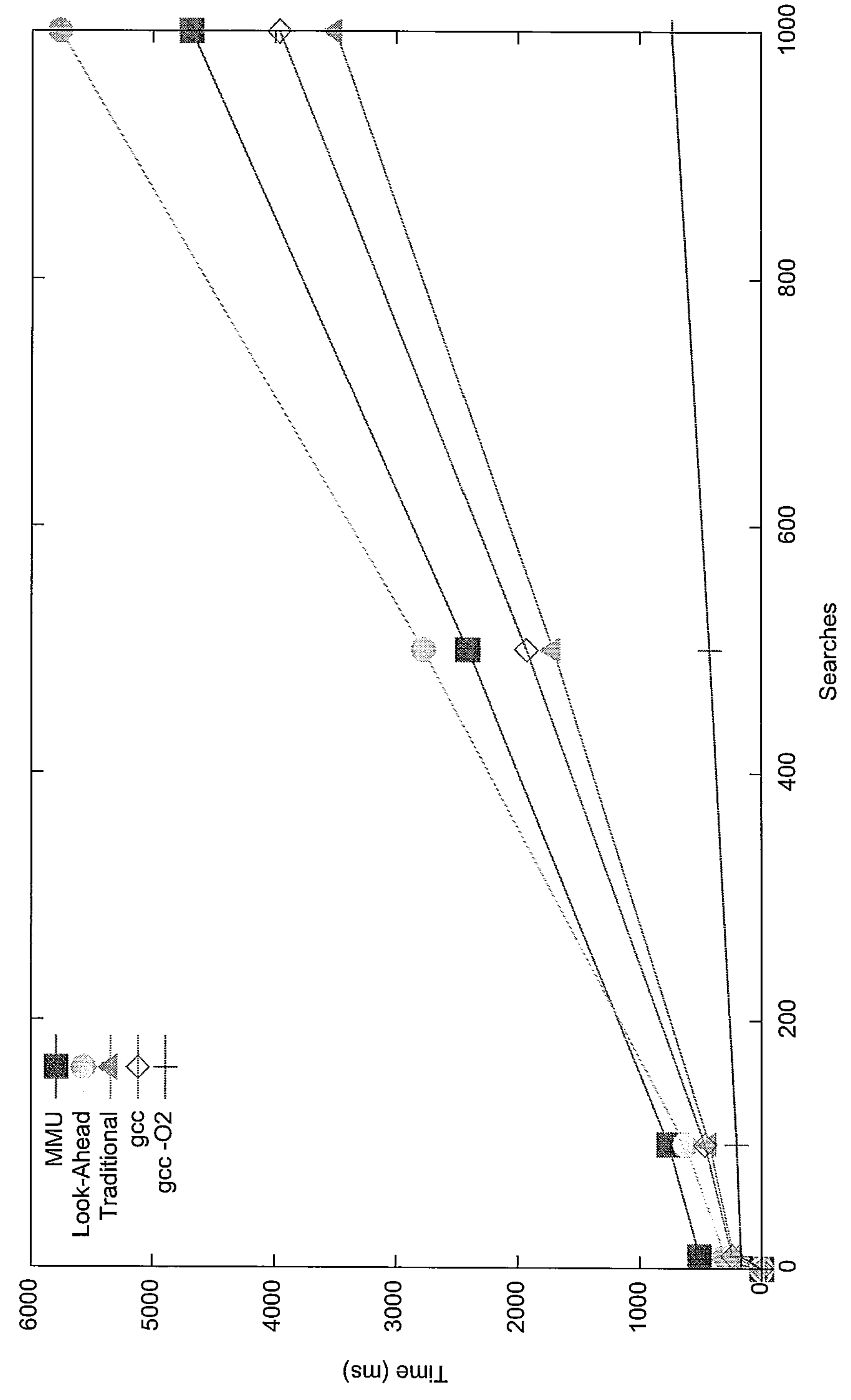


Figure 4.6: Unbalanced Binary Tree Single-Threaded No Heap

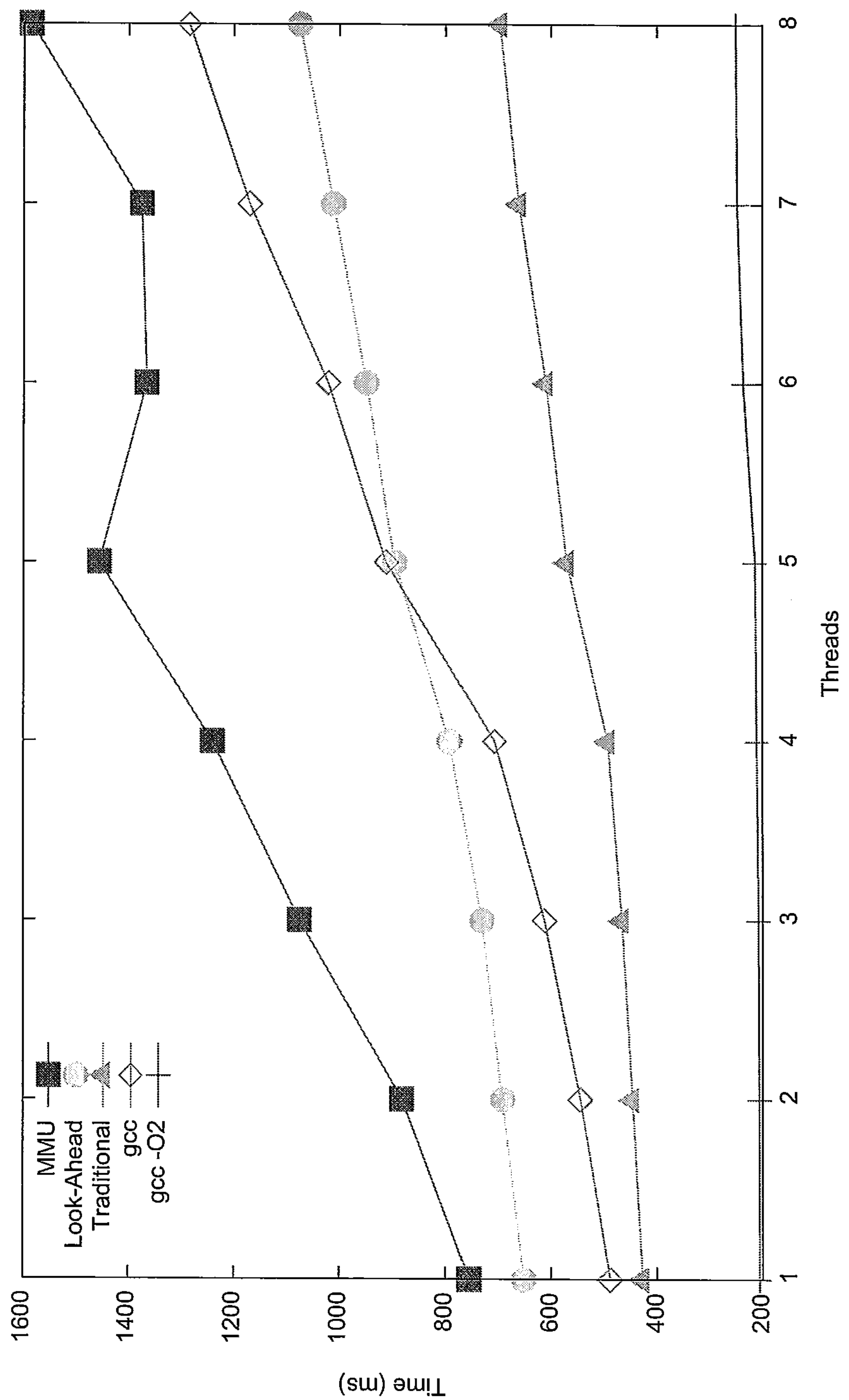


Figure 4.7: Unbalanced Binary Tree Multi-Threaded Cores

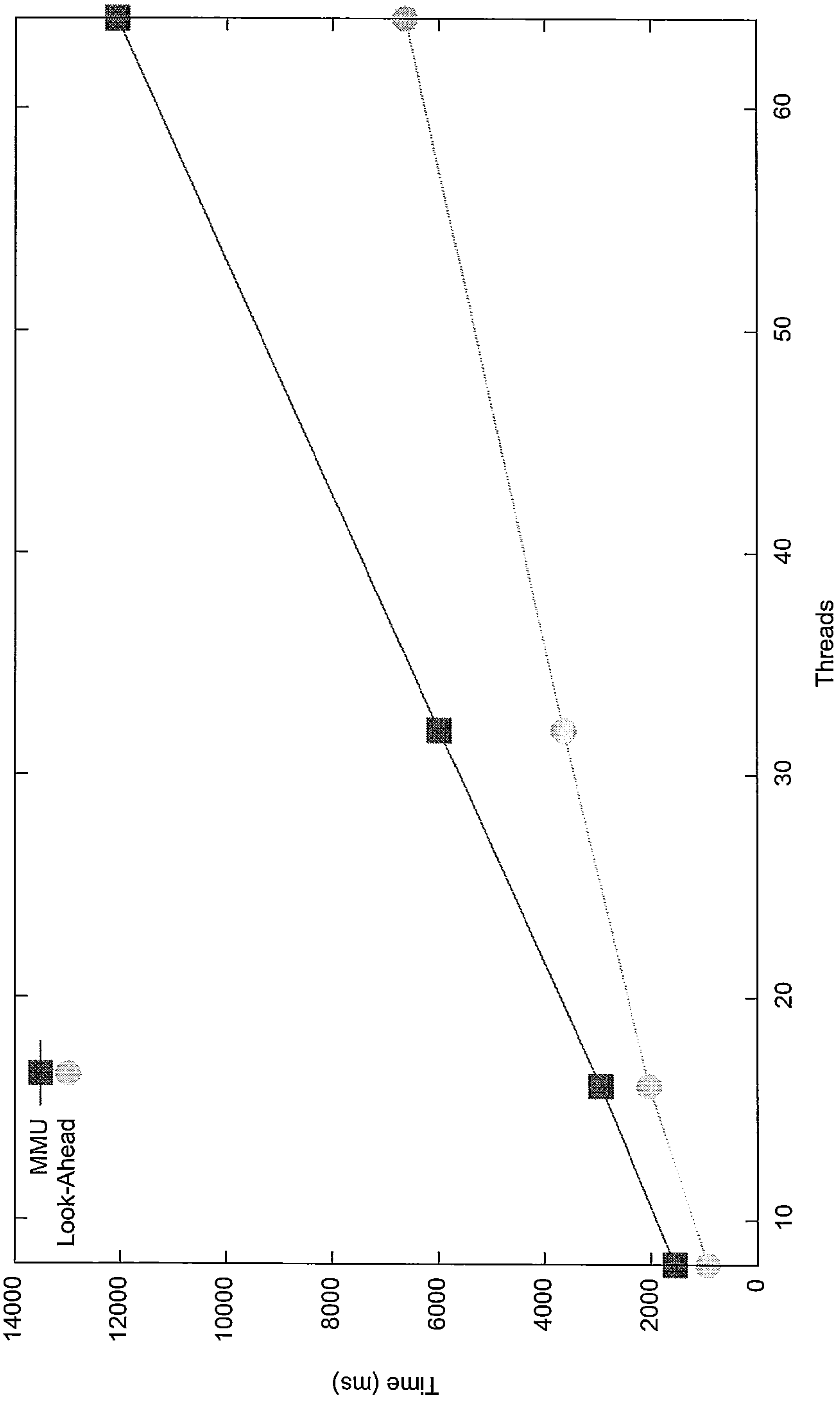


Figure 4.8: Unbalanced Binary Tree Multi-Threaded Quantity

4.3 "Real World"

4.3.1 Single-Threaded

Due to the simplistic nature of the "work" in this experiment (a simple for loop increments an integer, as discussed in Section 3.5.3), gcc -O2 could not be tested as it folded the loop into a constant amount of work. As can be seen in Figure 4.9, the non-optimized gcc performed worse than all the C-- stack mechanisms. Omitting non-optimized gcc in Figure 4.10, all mechanisms perform equally well, even including the Heap mechanism which performed poorly in the above experiments. This appears to indicate that in existing "real world" usage, the overhead from all of these stack mechanisms is insignificant.

4.3.2 Multi-Threaded "Cores"

Figures 4.11 and 4.12 show linear scaling across the 4 cores in the test system. Non-optimized gcc continues to perform worse than all the C-- mechanisms, but all implemented C-- stack mechanisms continue to perform equally well.

4.3.3 Multi-Threaded "Quantity"

Figures 4.13 and 4.14 show linear scaling, with non-optimized gcc performing worse than all the C-- mechanisms, but all implemented C-- stack mechanisms continue to perform equally well.

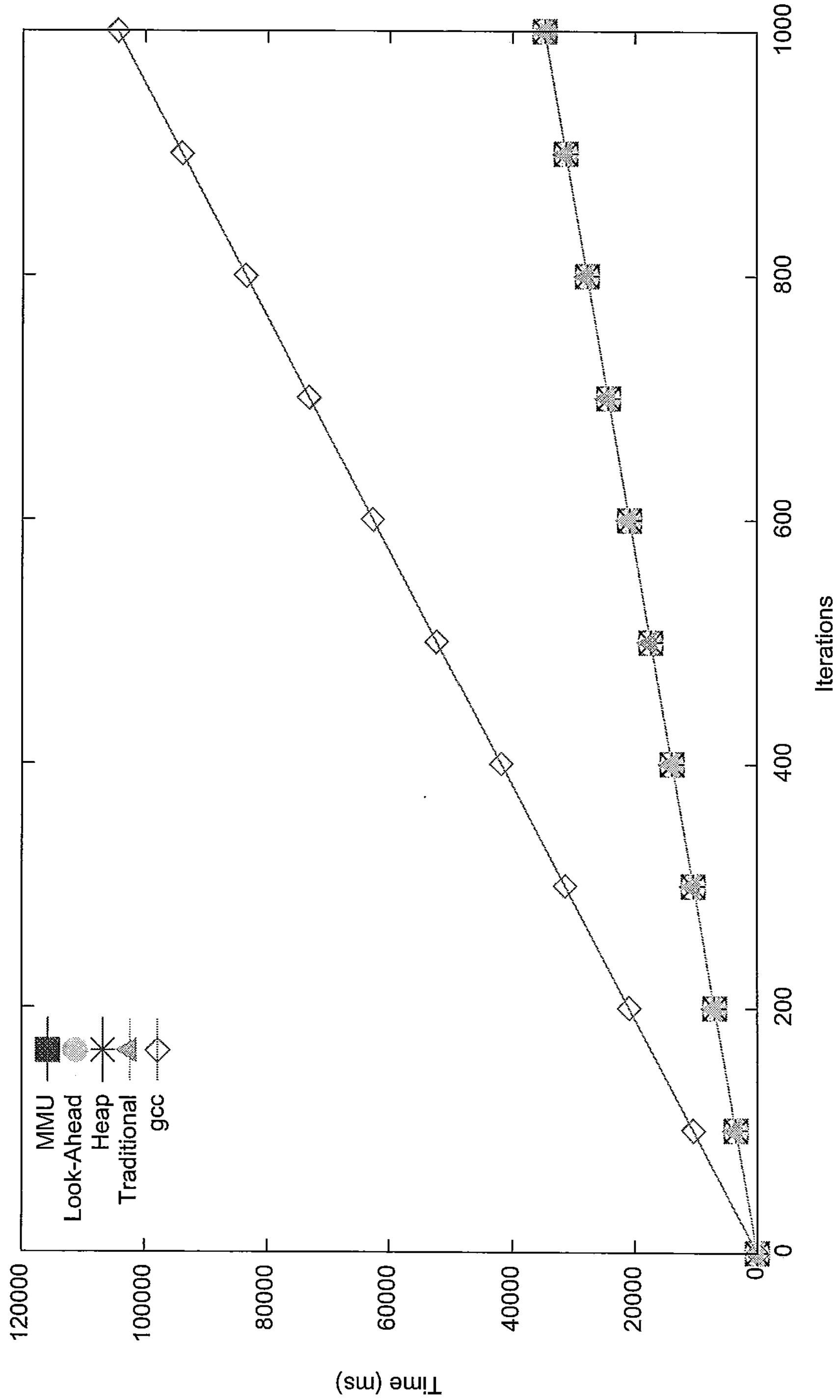


Figure 4.9: “Real World” Single-Threaded All

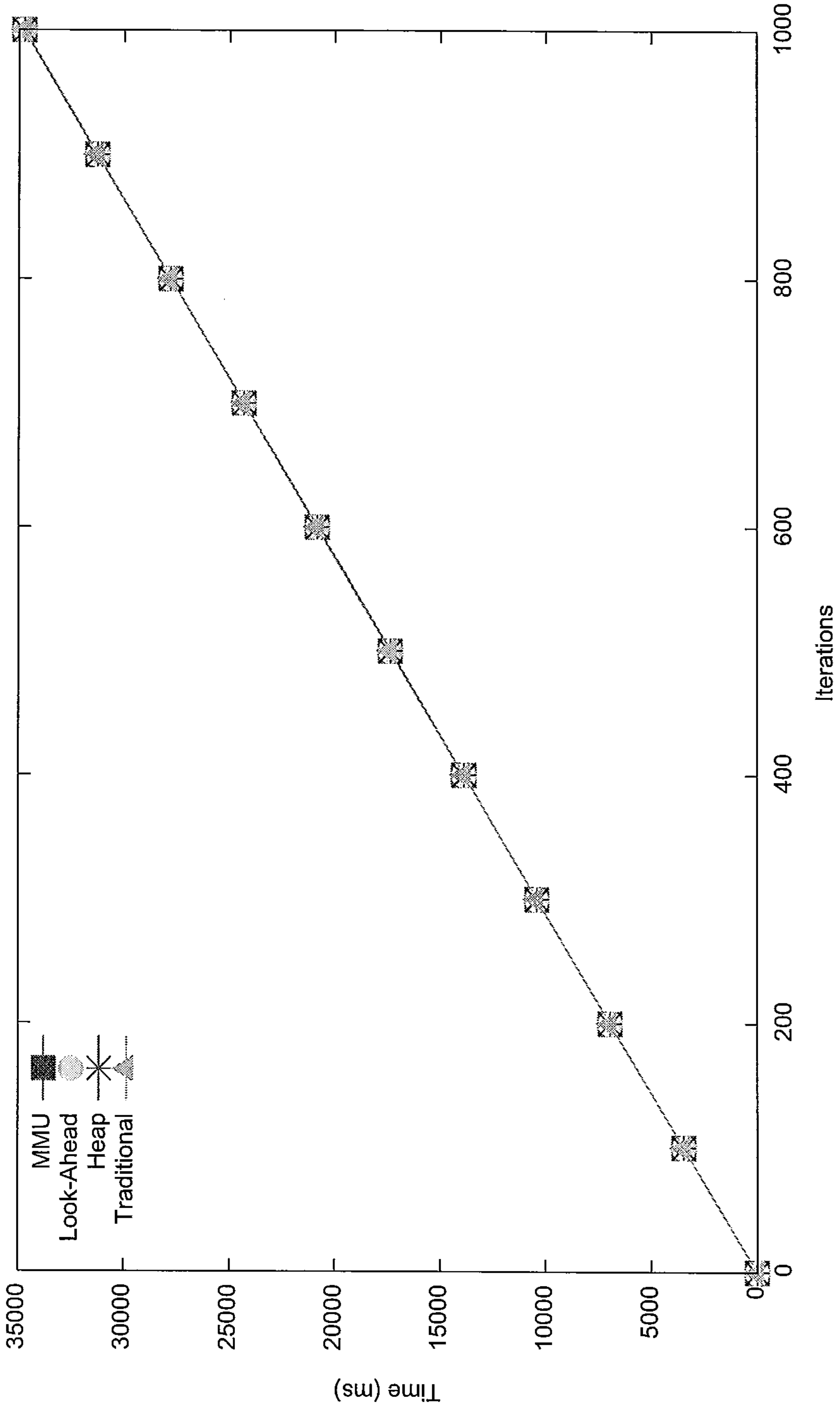


Figure 4.10: “Real World” Single-Threaded C--

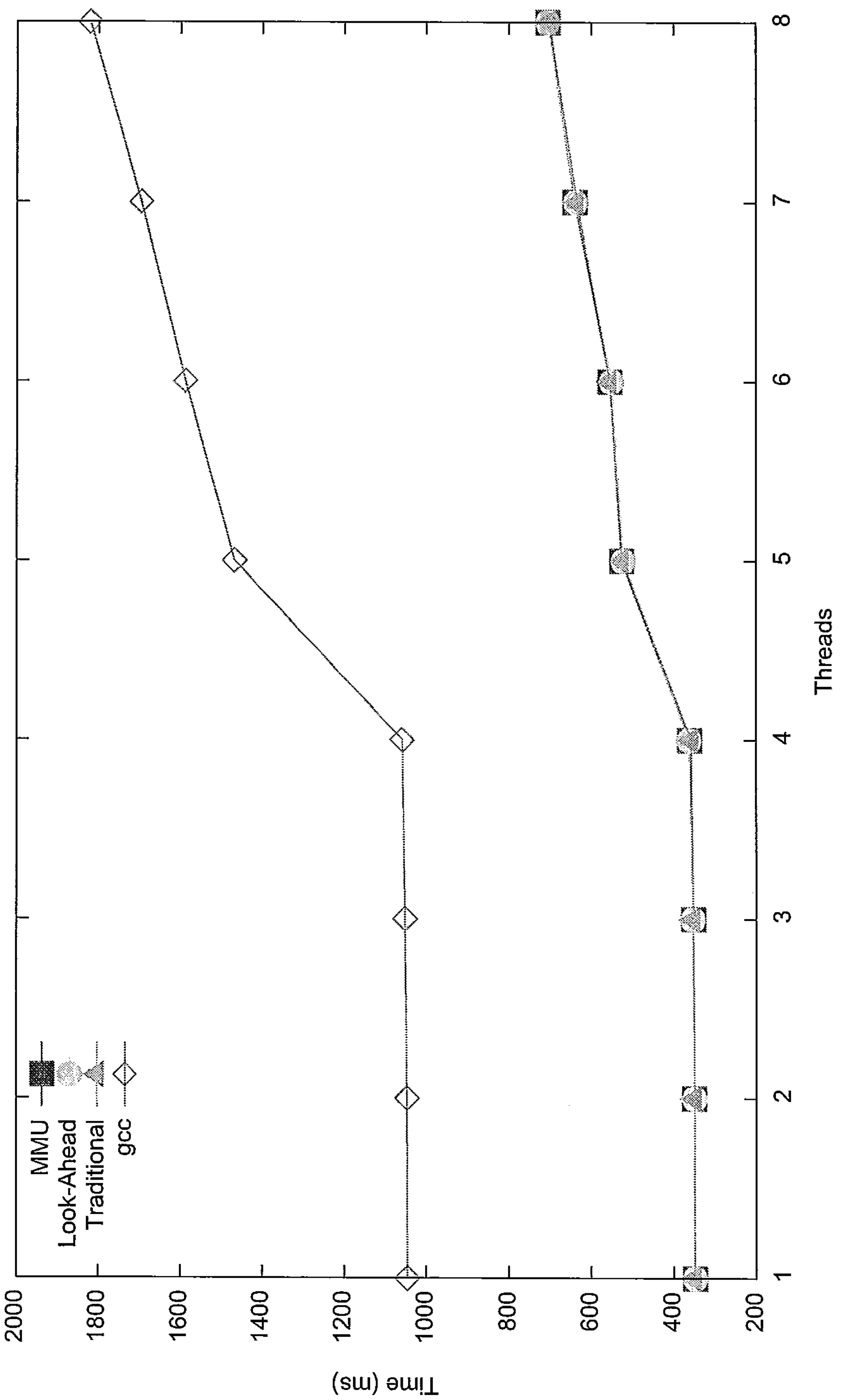


Figure 4.11: "Real World" Multi-Threaded Cores

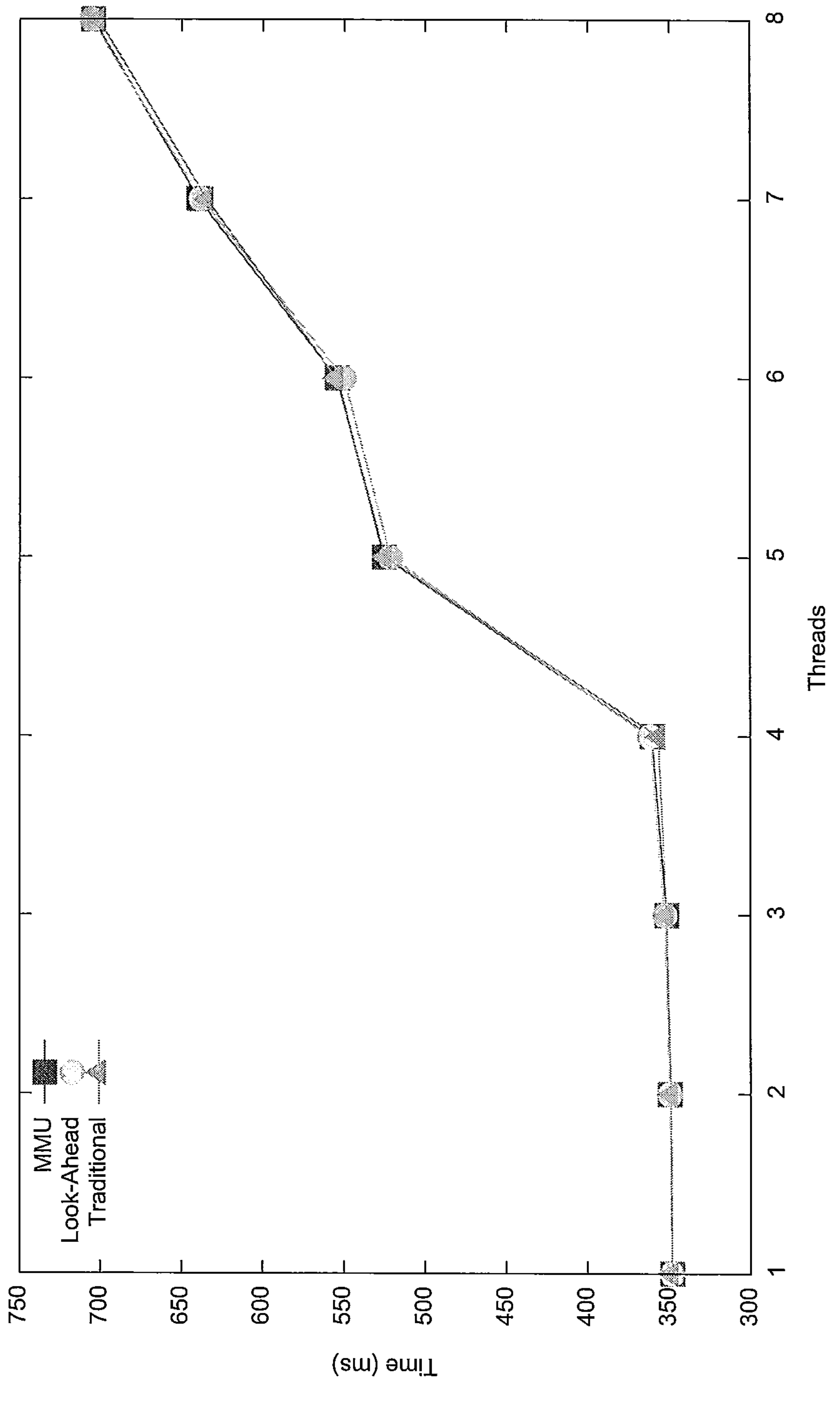


Figure 4.12: "Real World" Multi-Threaded Cores C--

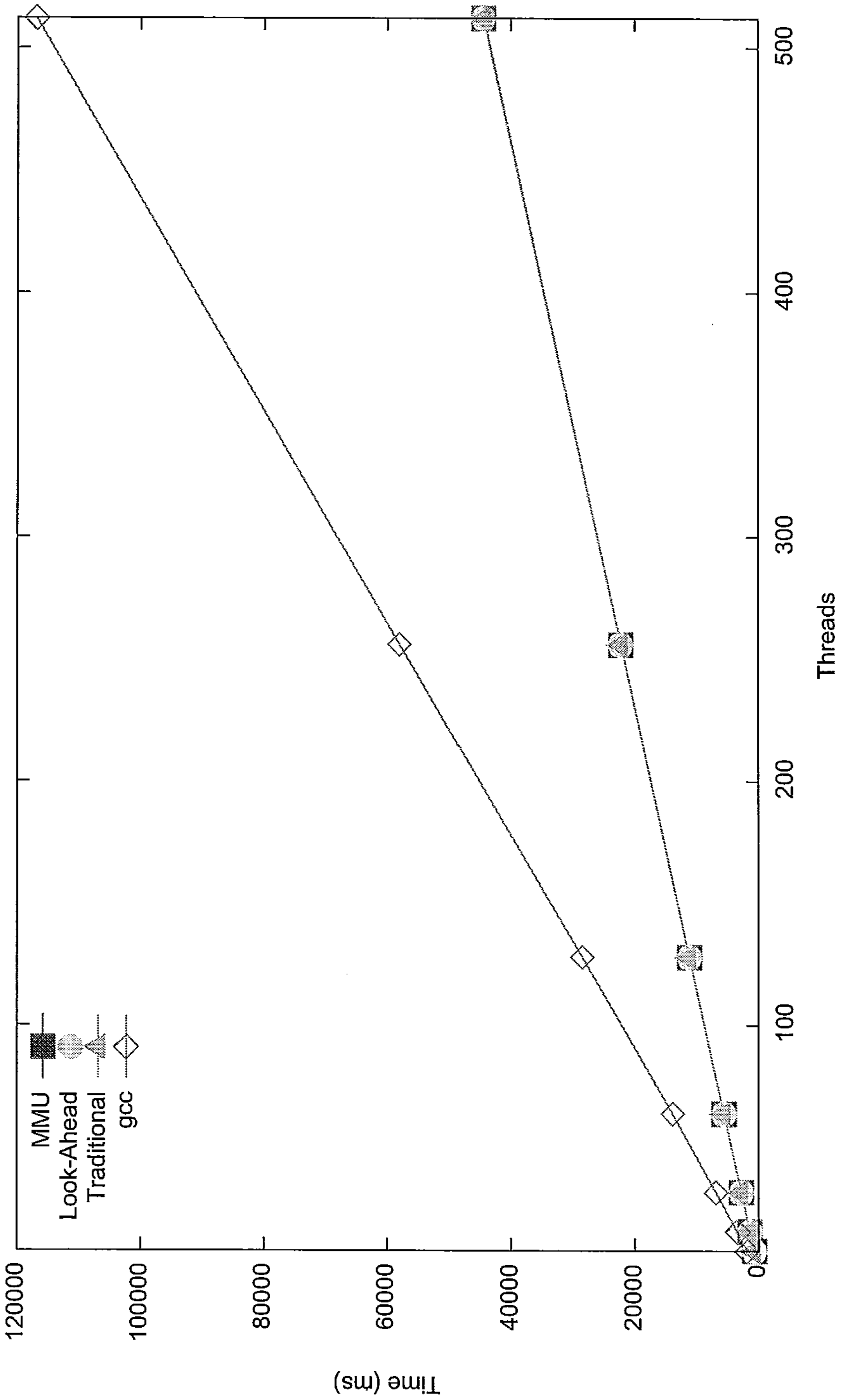


Figure 4.13: "Real World" Multi-Threaded Quantity

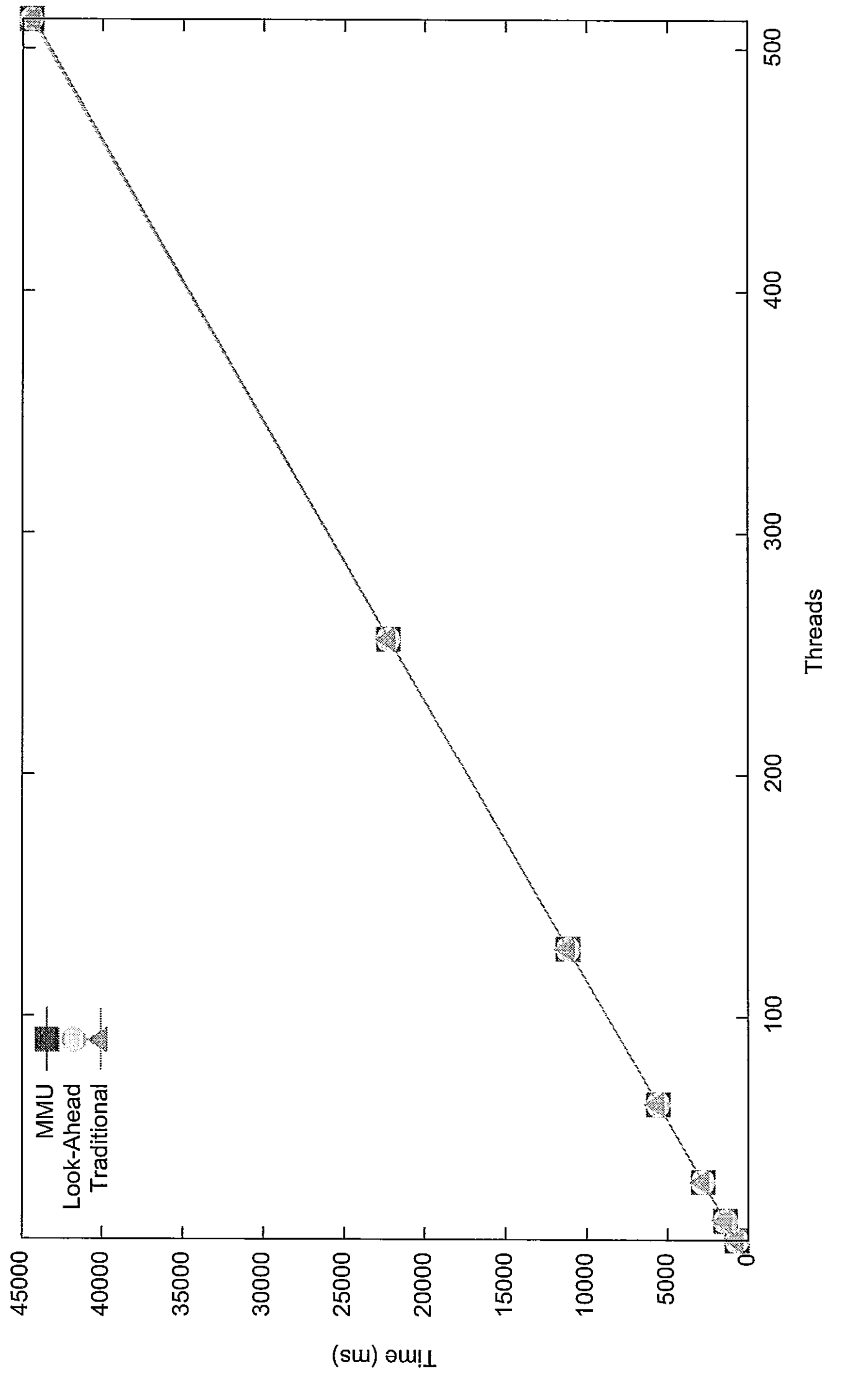


Figure 4.14: "Real World" Multi-Threaded Quantity C--

Chapter 5

Conclusions

In single-threaded applications, the MMU mechanism appears to show the best performance out of all the non-traditional stack mechanisms. However, the MMU mechanism does not scale well for reasons discussed in Section 4.1.2, and the traditional stack mechanism is already adequate for existing single-threaded applications. During heavily recursive usage patterns the Look-Ahead mechanism shows the best scalability, but demonstrates a fixed amount of overhead due to the fact that the call graph optimizations are of no use in a recursive call pattern. In existing "real world" usage patterns, the overhead from all the stack mechanisms appears to be insignificant. However, such "real world" usage patterns may very well exist due to the fact that heavy recursion in multi-threaded programs is problematic for the reasons discussed in Section 2.1.2. All else being equal in "real world" usage patterns, the Look-Ahead mechanism is suggested as the best replacement for the Traditional stack mechanism in multi-threaded applications for its scalability even during heavily recursive usage patterns that a dynamic stack mechanism allows.

Chapter 6

Future Work

As discussed in Section 4.1.2, the MMU mechanism displayed poor scalability for what was surmised to be the linear de-multiplexing of signals implemented in the pthreads library. The MMU mechanism has the potential to outperform the Look-Ahead mechanism assuming that using hardware to detect overflow should be faster than explicit conditional checks, especially during heavily recursive usage patterns where call graph optimizations cannot reduce the number of conditional checks. Further investigation with focus on a new threading library, and possibly some operating system kernel routines would be required to determine the source of the observed scalability issues.

It should also be noted that the C-- compiler has very little in the way of optimizations, and does not implement the full C language. Implementing the most promising stack mechanisms into an existing professional compiler framework such as LLVM would allow for better comparisons of more complex real world programs, and assuming that the experiments continued to show similar runtime performance for the dynamic stack mechanisms, these dynamic stack mechanisms could be utilized in real world applications.

Bibliography

[1] "GNU Compiler Collection." http://gcc.gnu.org.

[2] "The LLVM Compiler Infrastructure." http://llvm.org.

[3] "The Netwide Assembler." http://www.nasm.us.

[4] "pthread_attr_setstacksize man page, The Linux *man-pages* project," November 2008. http://www.kernel.org/doc/man-pages/online/pages/man3/pthread_attr_setstacksize.3.html.

[5] "pthreads man page, The Linux *man-pages* project," November 2008. http://www.kernel.org/doc/man-pages/online/pages/man7/pthreads.7.html.

[6] "Frequently Asked Questions: Intel® Multi-Core Processor Architecture," October 2010. http://software.intel.com/en-us/articles/frequently-asked-questions-intel-multi-core-processor-architecture/.

[7] "GCC's Stdcall Calling Convention," October 2010. http://gcc.gnu.org/onlinedocs/gnat_ugn_unw/Stdcall-Calling-Convention.html.

[8] "GNU C Library," November 2010. http://www.gnu.org/software/libc/.

[9] "malloc man page, The Linux *man-pages* project," October 2010. http://www.kernel.org/doc/man-pages/online/pages/man3/malloc.3.html.

[10] AMD®, "Multi-Core Processors — The Next Evolution in Computing," November 2010. http://www.amd.com/us/Documents/33211A_Multi-Core_WP_en.pdf.

[11] X.-L. Cui, "An Experimental Implementation of Action-Based Concurrency," Master's thesis, McMaster University, Hamilton, January 2009.

[12] A. R. Disteli and P. Reali, "Combining Oberon with Active Objects," in *Proceedings of the Joint Modular Languages Conference on Modular Programming Languages*, pp. 221–235, Springer-Verlag New York, 1997.

[13] A. Dunkels and O. Schmidt, "Protothreads : Lightweight Stackless Threads in C," Technical Report, Swedish Institute of Computer Science, March 2005.

[14] G. Hogen and R. Loogen, "A New Stack Technique for the Management of Runtime Structures in Distributed Implementations." Informatik-Berichte 93-3, RWTH Aachen, 1993.

[15] IEEE, "IEEE Standard 1003.1c-1995 thread extensions." IEEE 1995, ISBN 1-55937-375-X.

[16] Intel®, "Intel Technology Journal, Volume 06, Issue 01, Hyper Threading Technology," Technical Report, February 2002.

[17] Intel®, "Intel® 64 and IA-32 Architectures Software Developer's Manual," June 2009.

[18] S. Lalis and B. A. Sanders, "Adding Concurrency to the Oberon System," in *Proceedings of the International Conference on Programming Languages and System Architectures*, pp. 328–344, Springer-Verlag New York, 1994.

[19] Microsoft®, "Windows Development," November 2010. `http://msdn.microsoft.com/en-us/library/ms686774(v=VS.85).aspx`.

[20] B. Middha, M. Simpson, and R. Barua, "MTSS: Multi Task Stack Sharing for Embedded Systems," in *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pp. 191–201, ACM New York, 2005.

[21] M. Pizka, "Thread Segment Stacks," in *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications*, June 1999.

[22] Sun Microsystems®, "Multithreaded Programming Guide, Solaris," Technical Report, September 2008.

[23] H. Sutter and J. Larus, "Software and the Concurrency Revolution," *Queue*, vol. 3, pp. 54–62, September 2005.

[24] C. Tismer, "Continuations and Stackless Python Or "How to change a Paradigm of an existing Program"," in *Proceedings of the 8th International Python Conference*, January 2000.

[25] R. von Behren, J. Condit, and E. Brewer, "Why Events Are A Bad Idea (for high-concurrency servers)," in *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, pp. 19–24, USENIX Association, 2003.

[26] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: Scalable Threads for Internet Services," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pp. 268–281, ACM New York, 2003.

[27] M. F. Wilding and D. A. Wood, "Heap and Stack Layout for Multithreaded Processes in a Processing System." Patent, November 2008. US 7447829.

[28] K.-F. Wong and B. Dageville, "Supporting Thousands of Threads Using a Hybrid Stack Sharing Scheme," in *Proceedings of the 1994 ACM Symposium on Applied Computing*, pp. 493–498, ACM New York, 1994.

Appendix A

Experiment Listings

A.1 Simple Threads Library

Each thread implementation from Section 3.4 has a different thread library. Each thread library is a simple wrapper around pthreads with a different thread entry routine for each thread implementation. These thread libraries are compiled with gcc in order to interface with all the system header files, using an interface that is compatible with C--. Note that since Per Procedure Heap Allocation was not compatible with pthreads, there is no thread library for that stack implementation.

A.1.1 Traditional Fixed-Size Stack

Listing A.1: Traditional Fixed-Size Stack Thread Library

```
#include <pthread.h>
#include <stdlib.h>

extern void * STAMEX_CALLBACK_TRAMPOLINE( void * arg );

struct STAMEX_CALLBACK {
    void * arg;
    int fp;
};

int sthread_create( void * arg, int fp
```

```
                        , int fpStackSize ) {
    pthread_t tid;
    pthread_attr_t attr;
    struct STAMEX_CALLBACK * sc =
        malloc( sizeof( struct STAMEX_CALLBACK ) );

    if ( pthread_attr_init( &attr ) != 0 ) {
        perror( "pthread_attr_init\n" );
        exit(1);
    }

    if ( fpStackSize > 0 ) {
        if ( pthread_attr_setstacksize(
                &attr, fpStackSize ) != 0 ) {
            perror( "pthread_attr_setstacksize\n" );
            exit(1);increased
        }
    }

    sc->arg = arg;
    sc->fp = fp;
    pthread_create( &tid
                  , &attr
                  , STAMEX_CALLBACK_TRAMPOLINE
                  , sc );

    pthread_attr_destroy( &attr );

    return tid;
}

void * sthread_mutex_create() {
    pthread_mutex_t * m
        = malloc( sizeof( pthread_mutex_t ) );

```

```
    pthread_mutex_init( m, NULL );

    return m;
}

void sthread_mutex_destroy( void * m ) {
    pthread_mutex_destroy( m );
    free( m );
}

void sthread_exit( void * val ) {
    pthread_exit( val );
}

int sthread_join( int id, void ** retval ) {
    return pthread_join( id, retval );
}

void sthread_mutex_lock( void * m ) {
    pthread_mutex_lock( (pthread_mutex_t *)m );
}

void sthread_mutex_unlock( void * m ) {
    pthread_mutex_unlock( (pthread_mutex_t *)m );
}
```

A.1.2 Linked Stack Chunks with Look-Ahead Overflow Detection

Listing A.2: Linked Stack Chunks with Look-Ahead Overflow Detection Thread Library

```
#include <pthread.h>
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
```

```
#include "sthread.h"

extern void * STAMEX_CALLBACK_TRAMPOLINE( void * arg );
extern int STAMEX_PAGE_SIZE;

struct STAMEX_CALLBACK {
    void * arg;
    int fp;
};

int sthread_create( void * arg, int fp
                  , int fpStackSize ) {
    pthread_t tid;
    struct STAMEX_CALLBACK * sc =
        malloc( sizeof( struct STAMEX_CALLBACK ) );
    pthread_attr_t attr;
    int stacksize;

    if ( pthread_attr_init( &attr ) != 0 ) {
        perror( "pthread_attr_init\n" );
        exit(1);
    }

    if ( pthread_attr_setstacksize(
            &attr, STAMEX_PAGE_SIZE * 4 ) != 0 ) {
        perror( "pthread_attr_setstacksize\n" );
        exit(1);
    }

    sc->arg = arg;
    sc->fp = fp;
    pthread_create( &tid
                  , &attr
                  , STAMEX_CALLBACK_TRAMPOLINE
```

```
                        , sc );

    pthread_attr_destroy( &attr );

    return tid;
}

void sthread_exit( void * val ) {
    pthread_exit( val );
}

int sthread_join( int id, void ** retval ) {
    return pthread_join( id, retval );
}

void * sthread_mutex_create() {
    pthread_mutex_t * m
        = malloc( sizeof( pthread_mutex_t ) );

    pthread_mutex_init( m, NULL );

    return m;
}

void sthread_mutex_destroy( void * m ) {
    pthread_mutex_destroy( m );
    free( m );
}

void sthread_mutex_lock( void * m ) {
    pthread_mutex_lock( (pthread_mutex_t *)m );
}

void sthread_mutex_unlock( void * m ) {
    pthread_mutex_unlock( (pthread_mutex_t *)m );
```

```
}
```

A.1.3 Linked Stack Chunks with MMU Overflow Detection

Listing A.3: Linked Stack Chunks with MMU Overflow Detection Thread Library

```
#include <pthread.h>
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

#include "sthread.h"

extern void * STAMEX_CALLBACK_TRAMPOLINE( void * arg );
extern int STAMEX_PAGE_SIZE;
extern int STAMEX_STACK_SIZE;

struct STAMEX_CALLBACK {
    void * arg;
    int fp;
    int fpStackSize;
};

int sthread_create( void * arg, int fp
                    , int fpStackSize ) {
    pthread_t tid;
    struct STAMEX_CALLBACK * sc =
        malloc( sizeof( struct STAMEX_CALLBACK ) );
    pthread_attr_t attr;
    int stacksize;

    if ( pthread_attr_init( &attr ) != 0 ) {
        perror( "pthread_attr_init\n" );
        exit(1);
    }

```

```
    if ( pthread_attr_setguardsize(
            &attr, STAMEX_PAGE_SIZE ) != 0 ) {
        perror( "pthread_attr_setguardsize\n" );
        exit(1);
    }

    if ( pthread_attr_setstacksize(
            &attr, STAMEX_PAGE_SIZE
                + STAMEX_STACK_SIZE ) != 0 ) {
        perror( "pthread_attr_setstacksize\n" );
        exit(1);
    }

    sc->arg = arg;
    sc->fp = fp;
    sc->fpStackSize = fpStackSize;
    pthread_create( &tid
                  , &attr
                  , STAMEX_CALLBACK_TRAMPOLINE
                  , sc );

    pthread_attr_destroy( &attr );

    return tid;
}

void sthread_exit( void * val ) {
    pthread_exit( val );
}

int sthread_join( int id, void ** retval ) {
    return pthread_join( id, retval );
}

void * sthread_mutex_create() {
```

```
    pthread_mutex_t * m
        = malloc( sizeof( pthread_mutex_t ) );

    pthread_mutex_init( m, NULL );

    return m;
}

void sthread_mutex_destroy( void * m ) {
    pthread_mutex_destroy( m );
    free( m );
}

void sthread_mutex_lock( void * m ) {
    pthread_mutex_lock( (pthread_mutex_t *)m );
}

void sthread_mutex_unlock( void * m ) {
    pthread_mutex_unlock( (pthread_mutex_t *)m );
}
```

A.1.4 GCC

In order to be able to use a single source file with gcc and C--, a simple thread library wrapper was created for gcc.

Listing A.4: GCC Thread Library

```
#include <pthread.h>
#include <stdlib.h>

int sthread_create( void * arg, int fp, int fpStackSize ) {
    pthread_t tid;
    pthread_attr_t attr;

    if ( pthread_attr_init( &attr ) != 0 ) {
        perror( "pthread_attr_init\n" );
```

```
        exit(1);
    }

    if ( fpStackSize > 0 ) {
        if ( pthread_attr_setstacksize(
                &attr, fpStackSize ) != 0 ) {
            perror( "pthread_attr_setstacksize\n" );
            exit(1);
        }
    }

    pthread_create( &tid
                  , &attr
                  , fp
                  , arg );

    pthread_attr_destroy( &attr );

    return tid;
}

void * sthread_mutex_create() {
    pthread_mutex_t * m
        = malloc( sizeof( pthread_mutex_t ) );

    pthread_mutex_init( m, NULL );

    return m;
}

void sthread_mutex_destroy( void * m ) {
    pthread_mutex_destroy( m );
    free( m );
}

```

```
void sthread_exit( void * val ) {
    pthread_exit( val );
}

int sthread_join( int id, void ** retval ) {
    return pthread_join( id, retval );
}

void sthread_mutex_lock( void * m ) {
    pthread_mutex_lock( (pthread_mutex_t *)m );
}

void sthread_mutex_unlock( void * m ) {
    pthread_mutex_unlock( (pthread_mutex_t *)m );
}
```

A.2 Summation

The following listings are for the experiments detailed in Section 3.5.1.

A.2.1 Single Threaded

Listing A.5: Single Threaded Summation

```
extern int atoi( char * nptr );
extern void free( void * ptr );
extern void * malloc( int size );
extern int printf( char * fmt, ... );

int summation( int n ) {
    int ret;

    if ( n == 0 ) {
        return 0;
    }
```

```

    ret = n + summation( n - 1 );

    return ret;
}

int main( int argc, char ** argv ) {
    int arg;
    int * tids;
    void * ret;

    if ( argc <= 1 ) {
        printf( "Usage: <number>\n" );
        return 1;
    }

    arg = atoi( argv[1] );

    printf( "Data: %d %d\n"
          , arg / 1000000, summation( arg ) );

    return 0;
}
```

A.2.2 Multi-Threaded

Listing A.6: Multi-Threaded Summation

```
extern int atoi( char * nptr );
extern void free( void * ptr );
extern void * malloc( int size );
extern int printf( char * fmt, ... );
extern int sthread_create( void * arg, int fp
                         , int fpStackSize );
extern int sthread_join( int tid, void ** retval );

```

```c
int summation( int n ) {
    int ret;

    if ( n == 0 ) {
        return 0;
    }

    ret = n + summation( n - 1 );

    return ret;
}

void * thread_entry( void * arg ) {
    int ret;
    ret = summation( (int)arg );

    return (void *)ret;
}

int main( int argc, char ** argv ) {
    int i;
    int arg;
    int numthreads;
    int stacksize;
    int * tids;
    void * ret;

    if ( argc != 3 && argc != 4 ) {
        printf( "Usage:␣<numthreads>␣<summation>␣"
                "[stacksize]" );
        return 1;
    }

    numthreads = atoi( argv[1] );
    arg = atoi( argv[2] );
```

```
    if ( argc == 3 ) {
        stacksize = stacksizeof(thread_entry);
    } else {
        stacksize = atoi( argv[3] );
    }
    tids = malloc( numthreads * sizeof(int) );

    for ( i = 0; i < numthreads; i = i + 1 ) {
        tids[i] = sthread_create(
                              (void *)arg, thread_entry
                            , stacksize );
    }

    for ( i = 0; i < numthreads; i = i + 1 ) {
        sthread_join( tids[i], &ret );
    }

    printf( "Data: %d\n", numthreads );

    free( tids );

    return 0;
}
```

A.3 Unbalanced Binary Tree

The following listings are for the experiments detailed in Section 3.5.2.

A.3.1 Single Threaded

Listing A.7: Single Threaded Unbalanced Binary Tree

```
extern double atof( char * s );
extern int atoi( char * s );
extern void free( void * ptr );
extern void * malloc( int size );
```

```
extern int printf( char * fmt, ... );
extern int rand_r( int * seed );

struct ubt {
    struct ubt_node * root;
};

struct ubt_node {
    struct ubt_node * left;
    struct ubt_node * right;
    int value;
};

struct ubt_node * ubt_create_node_balanced(
    int l, int r ) {
    struct ubt_node * ret;
    int m;

    ret = NULL;

    if ( l <= r ) {
        ret = malloc( sizeof( struct ubt_node ) );
        m = (l+r)/2;
        ret->value = m;
        ret->left = ubt_create_node_balanced( l, m-1 );
        ret->right = ubt_create_node_balanced( m+1, r );
    }

    return ret;
}

struct ubt * ubt_create_balanced( int max ) {
    struct ubt * ret;

    ret = malloc( sizeof( struct ubt ) );
```

```
    ret->root = ubt_create_node_balanced( 0, max );

    return ret;
}

void ubt_node_free( struct ubt_node * n ) {
    if ( n == NULL ) {
        return;
    }

    ubt_node_free( n->left );
    ubt_node_free( n->right );

    free( n );
}

void ubt_free( struct ubt * p ) {
    printf( "start free\n" );
    if ( p->root != NULL ) {
        ubt_node_free( p->root );
    }

    printf( "mid free\n" );

    free( p );

    printf( "end free\n" );
}

struct ubt_node * ubt_node_max( struct ubt * p ) {
    struct ubt_node * ret;

    ret = p->root;

```

```
    while ( ret->right != NULL ) {
        ret = ret->right;
    }

    return ret;
}

int ubt_node_search( struct ubt_node * n, int val ) {
    int ret;

    if ( n == NULL ) {
        ret = 0;
    } else

    if ( val == n->value ) {
        ret = 1;
    } else

    if ( val < n->value ) {
        ret = ubt_node_search( n->left , val );
    } else {
        ret = ubt_node_search( n->right , val );
    }

    return ret;
}

int ubt_search( struct ubt * p, int val ) {
    return ubt_node_search( p->root , val );
}

int main( int argc , char ** argv ) {
    int balanced_size;
    int unbalanced_size;
    int searches;
```

```
    double r;
    double rand_max;
    int randseed;
    int i;
    double percent_spike;
    struct ubt * t;
    struct ubt_node * max;

    if ( argc != 5 ) {
        printf( "Usage: <balanced_size> <unbalanced_size> "
                "<percent_spike> <searches>\n" );
        return 1;
    }

    balanced_size = atoi( argv[1] );
    unbalanced_size = atoi( argv[2] );
    percent_spike = atof( argv[3] );
    searches = atoi( argv[4] );

    rand_max = 2147483647;
    randseed = 128191227;

    t = ubt_create_balanced( balanced_size );

    max = ubt_node_max( t );

    for ( i = 0; i < unbalanced_size; i = i + 1 ) {
        max->right = malloc( sizeof( struct ubt_node ) );
        max->right->value = i + balanced_size + 1;
        max->right->left = NULL;
        max->right->right = NULL;

        max = max->right;
    }

```

```
    for ( i = 0; i < searches; i = i + 1 ) {
        r = rand_r( &randseed );
        r = r / rand_max;

        if ( r < percent_spike ) {
            ubt_search( t, balanced_size
                                + unbalanced_size );
        } else {
            r = rand_r( &randseed );
            r = r / rand_max * balanced_size;

            ubt_search( t, (int)r );
        }
    }

    ubt_free( t );

    printf( "Data: %d\n", searches );

    return 0;
}
```

A.3.2 Multi-Threaded

Listing A.8: Multi-Threaded Unbalanced Binary Tree

```
extern int atoi( char * s );
extern double atof( char * s );
extern void free( void * ptr );
extern void * malloc( int size );
extern int printf( char * fmt, ... );
extern int rand_r( int * seed );
extern int sthread_create( void * arg, int fp
                           , int fpStackSize );
extern int sthread_join( int tid, void ** retval );

```

```
int balanced_size;
double percent_spike;
int unbalanced_size;

struct ubt * t;

struct ubt {
    struct ubt_node * root;
};

struct ubt_node {
    struct ubt_node * left;
    struct ubt_node * right;
    int value;
};

struct ubt_node * ubt_create_node_balanced(
    int l, int r ) {
    struct ubt_node * ret;
    int m;

    ret = NULL;

    if ( l <= r ) {
        ret = malloc( sizeof( struct ubt_node ) );
        m = (l+r)/2;
        ret->value = m;
        ret->left = ubt_create_node_balanced( l, m-1 );
        ret->right = ubt_create_node_balanced( m+1, r );
    }

    return ret;
}

struct ubt * ubt_create_balanced( int max ) {
```

```
    struct ubt * ret;

    ret = malloc( sizeof( struct ubt ) );

    ret->root = ubt_create_node_balanced( 0, max );

    return ret;
}

void ubt_node_free( struct ubt_node * n ) {
    if ( n == NULL ) {
        return;
    }

    ubt_node_free( n->left );
    ubt_node_free( n->right );

    free( n );
}

void ubt_free( struct ubt * p ) {
    if ( p->root != NULL ) {
        ubt_node_free( p->root );
    }

    free( p );
}

struct ubt_node * ubt_node_max( struct ubt * p ) {
    struct ubt_node * ret;

    ret = p->root;

    while ( ret->right != NULL ) {
        ret = ret->right;
```

```
    }

    return ret;
}

int ubt_node_search( struct ubt_node * n, int val ) {
    int ret;

    if ( n == NULL ) {
        ret = 0;
    } else

    if ( val == n->value ) {
        ret = 1;
    } else

    if ( val < n->value ) {
        ret = ubt_node_search( n->left , val );
    } else {
        ret = ubt_node_search( n->right , val );
    }

    return ret;
}

int ubt_search( struct ubt * p, int val ) {
    return ubt_node_search( p->root , val );
}

int thread_entry( void * arg ) {
    int i;
    int randseed;
    double r;
    double rand_max;
    int searches;
```

```

    rand_max = 2147483647;
    randseed = 128191227;
    searches = (int)arg;

    for ( i = 0; i < searches; i = i + 1 ) {
        r = rand_r( &randseed );
        r = r / rand_max;

        if ( r < percent_spike ) {
            ubt_search( t, balanced_size
                              + unbalanced_size );
        } else {
            r = rand_r( &randseed );
            r = r / rand_max * balanced_size;

            ubt_search( t, (int)r );
        }
    }
}

int main( int argc, char ** argv ) {
    int searches;
    int stacksize;
    int threads;
    int * tids;
    int i;
    struct ubt_node * max;

    if ( argc != 6 && argc != 7 ) {
        printf( "Usage: <balanced size> <unbalanced size>"
                "<percent_spike> <searches> <threads>"
                " [stacksize]\n" );
        return 1;
    }
```

```

    balanced_size = atoi( argv[1] );
    unbalanced_size = atoi( argv[2] );
    percent_spike = atof( argv[3] );
    searches = atoi( argv[4] );
    threads =  atoi( argv[5] );

    printf( "balanced_size %d\n", balanced_size );
    printf( "unbalanced_size %d\n", unbalanced_size );
    printf( "percent_spike %lf\n", percent_spike );
    printf( "searches %d\n", searches );
    printf( "threads %d\n", threads );


    if ( argc == 6 ) {
        printf( "Using stacksizeof\n" );
        stacksize = stacksizeof(thread_entry);
    } else {
        printf( "Using given stack size\n" );
        stacksize = atoi( argv[6] );
    }

    printf( "stacksize %d\n", stacksize );

    tids = malloc( threads * sizeof(int) );

    t = ubt_create_balanced( balanced_size );

    max = ubt_node_max( t );

    for ( i = 0; i < unbalanced_size; i = i + 1 ) {
        max->right = malloc( sizeof( struct ubt_node ) );
        max->right->value = i + balanced_size + 1;
        max->right->left = NULL;
        max->right->right = NULL;
```

```

            max = max->right;
        }

        for ( i = 0; i < threads; i = i + 1 ) {
            tids[i] = sthread_create( (void *)searches
                                    , thread_entry
                                    , stacksize );
        }

        for ( i = 0; i < threads; i = i + 1 ) {
            sthread_join( tids[i], NULL );
        }

        ubt_free( t );

        free( tids );

        printf( "Data: %d\n", threads );

        return 0;
}
```

A.4 “Real World”

The following listings are for the experiments detailed in Section 3.5.3.

A.4.1 Single Threaded

Listing A.9: Single Threaded “Real World”

```
extern int printf( char * fmt, ... );
extern int atoi( char * s );

int overall_iterations;
int recursive_count;
```

```
int work_per_non_recursive_call;
int work_per_recursive_call;

int a( int arg ) {
    int i;
    int end;

    end = work_per_non_recursive_call;
    for ( i = 0; i < end; i = i + 1 ) {
        arg = arg + 1;
    }

    arg = b( arg );

    arg = f( arg, recursive_count );

    return arg;
}

int b( int arg ) {
    int i;
    int end;

    end = work_per_non_recursive_call;
    for ( i = 0; i < end; i = i + 1 ) {
        arg = arg + 1;
    }

    arg = c( arg );

    return arg;
}

int c( int arg ) {
    int i;
```

```
    int end;

    end = work_per_non_recursive_call;
    for ( i = 0; i < end; i = i + 1 ) {
        arg = arg + 1;
    }

    arg = d( arg );

    arg = e( arg );

    return arg;
}

int d( int arg ) {
    int i;
    int end;

    end = work_per_non_recursive_call;
    for ( i = 0; i < end; i = i + 1 ) {
        arg = arg + 1;
    }

    return arg;
}

int e( int arg ) {
    int i;
    int end;

    end = work_per_non_recursive_call;
    for ( i = 0; i < end; i = i + 1 ) {
        arg = arg + 1;
    }

```

```
    return arg;
}

int f( int arg, int count ) {
    int i;
    int end;

    if ( count <= 0 ) {
        return arg;
    }

    count = count - 1;

    end = work_per_recursive_call;
    for ( i = 0; i < end; i = i + 1 ) {
        arg = arg + 1;
    }

    return f( arg, count );
}

int main( int argc, char ** argv ) {
    int i;
    int end;
    int arg;

    if ( argc != 5 ) {
        printf( "Usage: <overall_iterations> "
                "<recursive_count> "
                "<work_per_non_recursive_call> "
                "<work_per_recursive_call>\n" );
        return 1;
    }

    overall_iterations = atoi( argv[1] );
```

```
    recursive_count = atoi( argv[2] );
    work_per_non_recursive_call = atoi( argv[3] );
    work_per_recursive_call = atoi( argv[4] );

    printf( "overall_iterations:␣%d\n"
          , overall_iterations );
    printf( "recursive_count:␣%d\n", recursive_count );
    printf( "work_per_non_recursive_call:␣%d\n"
          , work_per_non_recursive_call );
    printf( "work_per_recursive_call:␣%d\n"
          , work_per_recursive_call );

    arg = 0;
    end = overall_iterations;
    for ( i = 0; i < end; i = i + 1 ) {
        arg = a( arg );
    }

    printf( "Data:␣%d\n", overall_iterations );

    return 0;
}
```

A.4.2 Multi-Threaded

Listing A.10: Multi-Threaded "Real World"

```
extern int atoi( char * s );
extern void free( void * ptr );
extern void * malloc( int size );
extern int printf( char * fmt, ... );
extern int sthread_create( void * arg, int fp
                          , int fpStackSize );
extern int sthread_join( int tid, void ** retval );

int overall_iterations;
```

```
int recursive_count;
int work_per_non_recursive_call;
int work_per_recursive_call;

int a( int arg ) {
    int i;
    int end;

    end = work_per_non_recursive_call;
    for ( i = 0; i < end; i = i + 1 ) {
        arg = arg + 1;
    }

    arg = b( arg );

    arg = f( arg, recursive_count );

    return arg;
}

int b( int arg ) {
    int i;
    int end;

    end = work_per_non_recursive_call;
    for ( i = 0; i < end; i = i + 1 ) {
        arg = arg + 1;
    }

    arg = c( arg );

    return arg;
}

int c( int arg ) {
```

```
    int i;
    int end;

    end = work_per_non_recursive_call;
    for ( i = 0; i < end; i = i + 1 ) {
        arg = arg + 1;
    }

    arg = d( arg );

    arg = e( arg );

    return arg;
}

int d( int arg ) {
    int i;
    int end;

    end = work_per_non_recursive_call;
    for ( i = 0; i < end; i = i + 1 ) {
        arg = arg + 1;
    }

    return arg;
}

int e( int arg ) {
    int i;
    int end;

    end = work_per_non_recursive_call;
    for ( i = 0; i < end; i = i + 1 ) {
        arg = arg + 1;
    }
```

```
    return arg;
}

int f( int arg, int count ) {
    int i;
    int end;

    if ( count <= 0 ) {
        return arg;
    }

    count = count - 1;

    end = work_per_recursive_call;
    for ( i = 0; i < end; i = i + 1 ) {
        arg = arg + 1;
    }

    return f( arg, count );
}

void * thread_entry( void * targ ) {
    int arg;
    int end;
    int i;

    arg = 0;
    end = overall_iterations;
    for ( i = 0; i < end; i = i + 1 ) {
        arg = a( arg );
    }

    return (void *)arg;
}
```

```

int main( int argc, char ** argv ) {
    int end;
    int i;
    void * ret;
    int stacksize;
    int threads;
    int * tids;

    if ( argc != 6 && argc != 7 ) {
        printf( "Usage: <threads> <overall_iterations> "
                "<recursive_count> "
                "<work_per_non_recursive_call> "
                "<work_per_recursive_call> "
                "[stacksize]\n" );
        return 1;
    }

    threads = atoi( argv[1] );
    overall_iterations = atoi( argv[2] );
    recursive_count = atoi( argv[3] );
    work_per_non_recursive_call = atoi( argv[4] );
    work_per_recursive_call = atoi( argv[5] );

    if ( argc == 6 ) {
        stacksize = stacksizeof(thread_entry);
    } else {
        stacksize = atoi( argv[6] );
    }

    printf( "threads: %d\n", threads );
    printf( "overall_iterations: %d\n"
          , overall_iterations );
    printf( "recursive_count: %d\n", recursive_count );
    printf( "work_per_non_recursive_call: %d\n"
```

```
            , work_per_non_recursive_call );
    printf( "work_per_recursive_call: %d\n"
           , work_per_recursive_call );

    tids = malloc( threads * sizeof(int) );

    for ( i = 0; i < threads; i = i + 1 ) {
        tids[i] = sthread_create( NULL, thread_entry
                                , stacksize );

        printf( "main: created %d\n", tids[i] );
    }

    for ( i = 0; i < threads; i = i + 1 ) {
        sthread_join( tids[i], &ret );
        printf( "Joined thread %d with ret value %d\n"
              , tids[i], ret );
    }

    free( tids );

    printf( "Data: %d\n", threads );

    return 0;
}
```

Appendix B

Experiment Data

This section contains raw data for all experiments with a 95% confidence interval.

B.1 Summation

Sum (Millions)	Mean Time (ms)	Variance
1	20.500	1.021
5	105.533	1.147
10	211.867	1.558
30	638.700	5.969
50	1065.300	6.918

Figure B.1: Summation Single Threaded gcc

Sum (Millions)	Mean Time (ms)	Variance
1	1.000	0.000
5	4.000	0.000
10	7.000	0.000
30	21.000	0.000
50	34.000	0.000
70	48.000	0.000
100	68.033	0.367

Figure B.2: Summation Single Threaded gcc -O2

Sum (Millions)	Mean Time (ms)	Variance
1	82.000	1.395
5	407.900	3.471
10	817.100	9.568
30	2450.333	21.894
50	4084.167	30.973
70	5716.067	45.797
100	8160.633	48.272

Figure B.3: Summation Single Threaded Heap

Sum (Millions)	Mean Time (ms)	Variance
1	13.933	0.509
5	67.167	0.761
10	133.600	1.131
30	400.367	3.051
50	666.800	3.181
70	935.200	9.031
100	1334.667	9.939

Figure B.4: Summation Single Threaded Look-Ahead

Sum (Millions)	Mean Time (ms)	Variance
1	8.000	0.000
5	40.000	0.000
10	80.000	0.000
30	239.567	1.141
50	399.700	1.505
70	560.067	2.037
100	800.433	3.363

Figure B.5: Summation Single Threaded MMU

Sum (Millions)	Mean Time (ms)	Variance
1	10.567	2.921
5	63.900	2.905
10	130.700	2.696
30	395.933	3.116
50	662.500	3.239
70	929.033	3.633
100	1327.633	3.313

Figure B.6: Summation Single Threaded Traditional

Threads	Mean Time (ms)	Variance
1	144.033	0.367
2	72.500	1.021
3	49.433	3.641
4	39.667	3.886
5	35.100	8.409
6	37.633	18.298
7	40.600	13.602
8	38.467	4.466

Figure B.7: Summation Multi-Threaded gcc -O2 "Cores"

Threads	Mean Time (ms)	Variance
1	1402.667	11.349
2	746.467	4.075
3	538.600	3.056
4	442.700	4.779
5	448.400	14.464
6	414.567	7.850
7	393.800	11.520
8	385.133	43.065

Figure B.8: Summation Multi-Threaded Look-Ahead "Cores"

Threads	Mean Time (ms)	Variance
1	810.200	3.267
2	540.600	6.185
3	485.700	10.494
4	498.733	11.848
5	540.767	39.278
6	525.733	62.863
7	533.433	28.533
8	544.800	15.129

Figure B.9: Summation Multi-Threaded MMU "Cores"

Threads	Mean Time (ms)	Variance
1	1345.500	4.177
2	691.767	3.899
3	472.700	3.659
4	363.133	2.569
5	322.033	7.219
6	282.333	16.364
7	259.900	9.554
8	241.633	13.499

Figure B.10: Summation Multi-Threaded Traditional "Cores"

Threads	Mean Time (ms)	Variance
8	39.133	7.357
16	35.333	8.441
32	33.667	4.576
64	32.533	2.970
128	33.533	1.019
256	33.167	3.296
512	34.933	2.932
1024	42.267	2.105

Figure B.11: Summation Multi-Threaded gcc -O2 "Quantity"

Threads	Mean Time (ms)	Variance
8	387.133	52.292
16	430.700	19.265
32	479.133	14.456
64	503.433	16.111
128	511.167	13.150
256	532.367	32.082
512	401.400	38.146
1024	323.433	18.824

Figure B.12: Summation Multi-Threaded Look-Ahead "Quantity"

Threads	Mean Time (ms)	Variance
8	547.900	20.713
16	595.967	10.189
32	647.800	6.943
64	659.933	6.973
128	657.500	7.904
256	641.067	14.332
512	513.767	17.155
1024	447.867	16.102

Figure B.13: Summation Multi-Threaded MMU “Quantity”

Threads	Mean Time (ms)	Variance
8	239.067	13.636
16	291.033	30.854
32	276.467	30.873
64	284.900	21.561
128	286.533	17.909
256	242.767	8.447
512	222.800	5.974
1024	223.300	6.672

Figure B.14: Summation Multi-Threaded Traditional “Quantity”

B.2 Unbalanced Binary Tree

Searches	Mean Time (ms)	Variance
10	240.267	1.662
100	464.333	8.588
500	1928.067	26.863
1000	3961.833	50.180

Figure B.15: Unbalanced Binary Tree Single Threaded gcc

Searches	Mean Time (ms)	Variance
10	170.300	0.936
100	204.733	2.412
500	425.667	1.695
1000	734.533	1.558

Figure B.16: Unbalanced Binary Tree Single Threaded gcc -O2

Searches	Mean Time (ms)	Variance
10	1196.867	12.112
100	2142.967	35.997
500	7518.667	188.614
1000	13374.000	91.121

Figure B.17: Unbalanced Binary Tree Single Threaded Heap

Searches	Mean Time (ms)	Variance
10	297.933	1.388
100	627.200	2.907
500	2774.033	11.593
1000	5767.567	22.559

Figure B.18: Unbalanced Binary Tree Single Threaded Look-Ahead

Searches	Mean Time (ms)	Variance
10	509.267	7.323
100	761.800	10.497
500	2408.300	8.422
1000	4685.267	22.665

Figure B.19: Unbalanced Binary Tree Single Threaded MMU

Searches	Mean Time (ms)	Variance
10	230.233	5.911
100	426.000	9.650
500	1707.467	35.937
1000	3502.100	36.331

Figure B.20: Unbalanced Binary Tree Single Threaded Traditional

Threads	Mean Time (ms)	Variance
1	488.133	5.292
2	544.533	7.636
3	612.167	14.207
4	707.167	17.004
5	912.233	28.081
6	1021.000	13.989
7	1169.667	21.249
8	1284.200	22.821

Figure B.21: Unbalanced Binary Tree Multi-Threaded gcc "Cores"

Threads	Mean Time (ms)	Variance
1	206.067	2.037
2	206.433	1.363
3	207.867	3.604
4	211.300	5.061
5	214.300	7.116
6	236.067	31.533
7	247.867	15.405
8	250.233	5.981

Figure B.22: Unbalanced Binary Tree Multi-Threaded gcc -O2 "Cores"

Threads	Mean Time (ms)	Variance
1	651.833	4.777
2	692.200	8.293
3	731.467	23.447
4	792.333	25.982
5	896.167	92.520
6	949.700	154.352
7	1011.800	210.585
8	1074.733	320.055

Figure B.23: Unbalanced Binary Tree Multi-Threaded Look-Ahead "Cores"

Threads	Mean Time (ms)	Variance
1	755.067	5.528
2	881.800	25.740
3	1075.367	27.338
4	1240.533	224.903
5	1455.433	461.089
6	1365.333	552.162
7	1375.033	117.046
8	1585.767	288.890

Figure B.24: Unbalanced Binary Tree Multi-Threaded MMU "Cores"

Threads	Mean Time (ms)	Variance
1	427.467	10.206
2	446.267	9.267
3	465.933	5.750
4	492.633	5.739
5	570.100	5.784
6	609.600	15.664
7	661.300	18.945
8	696.700	18.349

Figure B.25: Unbalanced Binary Tree Multi-Threaded Traditional "Cores"

Threads	Mean Time (ms)	Variance
8	928.267	307.654
16	2034.967	156.759
32	3643.567	355.707
64	6647.233	604.172

Figure B.26: Unbalanced Binary Tree Multi-Threaded Look-Ahead "Quantity"

Threads	Mean Time (ms)	Variance
8	1551.667	150.341
16	2935.000	93.381
32	5994.067	146.873
64	12077.200	177.223

Figure B.27: Unbalanced Binary Tree Multi-Threaded MMU "Quantity"

B.3 "Real World"

Searches	Mean Time (ms)	Variance
100	10457.000	48.615
200	20929.767	135.479
300	31378.200	251.054
400	41814.800	55.112
500	52300.133	267.031
600	62713.567	174.954
700	73190.567	389.076
800	83609.233	129.726
900	94040.300	315.154
1000	104528.567	120.227

Figure B.28: "Real World" Single Threaded gcc

Searches	Mean Time (ms)	Variance
100	3476.467	7.562
200	6953.233	16.444
300	10421.000	0.000
400	13895.667	12.301
500	17368.000	0.000
600	20841.467	1.019
700	24316.133	3.062
800	27790.933	3.836
900	31264.800	4.424
1000	34738.933	5.452

Figure B.29: "Real World" Single Threaded Heap

Searches	Mean Time (ms)	Variance
100	3474.833	6.416
200	6950.067	12.465
300	10427.600	21.749
400	13893.433	2.511
500	17366.633	3.141
600	20840.033	3.595
700	24313.133	4.075
800	27787.133	4.588
900	31259.967	5.206
1000	34734.633	4.701

Figure B.30: "Real World" Single Threaded Look-Ahead

Searches	Mean Time (ms)	Variance
100	3476.667	8.088
200	6953.967	16.315
300	10424.667	20.470
400	13905.367	33.922
500	17380.033	40.352
600	20858.067	47.043
700	24327.167	50.376
800	27799.333	50.484
900	31279.433	67.400
1000	34754.367	74.726

Figure B.31: "Real World" Single Threaded MMU

Searches	Mean Time (ms)	Variance
100	3476.567	8.128
200	6951.700	14.198
300	10426.133	20.295
400	13904.367	36.792
500	17373.067	38.841
600	20843.300	20.366
700	24324.900	45.668
800	27805.133	61.237
900	31262.633	29.471
1000	34734.100	5.121

Figure B.32: "Real World" Single Threaded Traditional

Threads	Mean Time (ms)	Variance
1	1045.867	4.242
2	1047.233	2.566
3	1051.500	8.198
4	1058.867	10.555
5	1468.300	78.435
6	1589.000	75.048
7	1696.533	113.923
8	1820.600	91.148

Figure B.33: "Real World" Multi-Threaded gcc "Cores"

Threads	Mean Time (ms)	Variance
1	348.000	0.000
2	349.467	3.829
3	352.533	3.365
4	361.667	8.540
5	521.433	27.587
6	550.233	46.686
7	638.167	24.527
8	705.700	18.537

Figure B.34: "Real World" Multi-Threaded Look-Ahead "Cores"

Threads	Mean Time (ms)	Variance
1	348.000	0.000
2	349.500	3.063
3	351.433	3.363
4	360.400	8.560
5	525.367	20.994
6	554.267	43.461
7	639.767	26.784
8	704.733	23.889

Figure B.35: "Real World" Multi-Threaded MMU "Cores"

Threads	Mean Time (ms)	Variance
1	348.400	1.000
2	348.600	1.944
3	351.467	4.825
4	356.367	11.971
5	524.700	14.050
6	554.967	41.483
7	634.533	20.803
8	701.533	11.378

Figure B.36: "Real World" Multi-Threaded Traditional "Cores"

Threads	Mean Time (ms)	Variance
8	1829.700	132.315
16	3427.800	57.301
32	6877.433	126.508
64	13909.633	254.544
128	28432.800	354.881
256	58041.567	689.085
512	116923.833	2528.484
1024	234874.033	2753.601

Figure B.37: "Real World" Multi-Threaded gcc "Quantity"

Threads	Mean Time (ms)	Variance
8	704.833	19.286
16	1409.000	20.242
32	2798.567	24.240
64	5573.467	14.975
128	11126.733	18.944
256	22244.167	26.807
512	44474.700	27.193
1024	88925.367	30.809

Figure B.38: "Real World" Multi-Threaded Look-Ahead "Quantity"

Threads	Mean Time (ms)	Variance
8	704.500	11.548
16	1403.333	17.473
32	2794.200	20.224
64	5576.367	22.285
128	11127.167	20.792
256	22240.267	21.299
512	44461.800	21.452
1024	88917.333	15.617

Figure B.39: "Real World" Multi-Threaded MMU "Quantity"

Threads	Mean Time (ms)	Variance
8	702.000	11.527
16	1405.067	17.470
32	2791.433	13.459
64	5572.200	20.713
128	11123.067	35.482
256	22189.600	78.179
512	44319.367	174.388
1024	88587.433	354.017

Figure B.40: "Real World" Multi-Threaded Traditional "Quantity"

Appendix C

Glossary of Acronyms

gcc Gnu Compiler Collection

IEEE Institute of Electrical and Electronics Engineers

LLVM The Low Level Virtual Machine (Compiler Infrastructure)

NASM Netwide Assembler

POSIX Portable Operating System Interface [for Unix]

Pthreads POSIX Threads

MMU Memory Management Unit