McMaster University, Department of Computing and Software, Report CAS-14-07-ES, April 2014

A Comparison of Scalable Multi-Threaded Stack Mechanisms

J. Moore-Oliva E. Sekerinski S. Yao

McMaster University, Canada

chatgris@gmail.com, {emil, yaos4}@mcmaster.ca

Abstract

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, 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 paper, 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 is used to implement promising stack mechanisms and a suite of benchmarks is used to compare their performance and scalability under varying usage profiles.

Categories and Subject Descriptors D.4.2 [*MEMORY STRUCTURES*]: Design Styles; D.3.4 [*Processors*]: Compilers

Keywords multi-threaded programs, runtime memory organization, stack sharing

1. Introduction

The traditional call stack mechanism – where the stack and heap grow from opposite sides – is often taken for granted. This is because, until recently, it has been an effective solution to the problem of bookkeeping during program execution – it provides a simple and efficient way to keep track of variable values and control 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, operating systems take the strategy of allocating a stack area “large enough for anything”, and on overflow do not attempt to extend it.

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 [10, 11, 15]. This has led to the more complex scenario: when concurrency is increased with the use of threads, the default “large enough” call stack mechanism with predetermined maximal stack size 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.

The topic of this paper is the comparison of call stack mechanisms for highly concurrent multi-threaded programs, 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 a concurrent object-oriented language [omitted]. Every object being concurrent in principle can easily lead to programs with thousands of threads. In an experimental implementation of this language, stack sizes were intentionally set low and stack-gobbling features, most notably recursion, were disabled as a workaround. The experimental compiler which we use for the evaluation of promising stack mechanism is described in more detail in [21]. The compiler emits code for x86 [6]. While desktop processors have moved to 64 bit architectures, embedded processors are increasingly moving from 8 to 16 to 32 bit architectures. With the embedded market an order of magnitude larger than the desktop market, 32 bit architectures remain relevant.

2. Related Work

This section categorizes and discusses existing and proposed multi-threaded call stack mechanisms. To start with, we briefly discuss the existing single-threaded call stack mechanism.

2.1 Single-Threaded Call Stack

The fact that a procedure will not return until all other procedures it called have returned lends itself to a stack, where each new *activation frame* is placed on top of all existing frames. In a single-threaded program, there is only one call stack, typically growing 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. 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. 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 [8, 18, 19].

2.2 Single-Threaded Stack Mechanism Extensions

The traditional call stack mechanism works well for single-threaded processes for two reasons. Firstly, for any single-threaded program, only one call stack is required. Secondly, the MMU allows each process to use the entire address space as if it were the only process

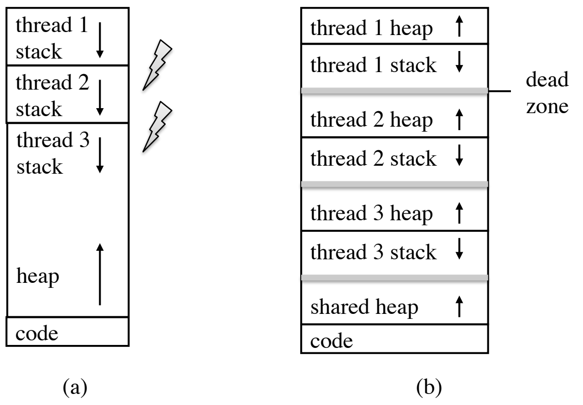


Figure 1. Single-threaded memory organization extended to multiple threads: (a) with single shared heap and (b) with multiple heaps for reducing heap contention.

running on the system; physical memory is not reserved for the process until it actually uses the space.

By contrast, 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 1 (a). 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 reminds of the practice without an MMU when programmers have to manually assign 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. With the number of cores on chips increasing and parallelism being projected as the way to increase performance in the future [10, 11, 15, 23], the traditional call stack is insufficient. We discuss several simple modifications that maintain each thread's call stack as a *contiguous* region of memory.

Solaris [19] 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 2 MB) is 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 cause a memory fault if accessed. *Windows* allocates by default 1 MB to each thread. *Mac OS X* allocates by default 8 MB to the *main thread* and 512 KB to *secondary threads*; *iOS* allocates by default 1MB to the main thread and also 512 to the secondary thread.

Oberon with active objects [12] can be viewed as a special case 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 128 KB, thereby supporting up to 16,384 call stacks simultaneously.

Concurrent Oberon [17] uses a call stack that is of 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 [26] attempts to address both heap contention and stack space in its proposed memory layout, depicted in Figure 1 (b). Each stack/heap block is created from an initial base address, from which the thread and heap stack grow in opposite directions. The patent does not specify how the initial base addresses are computed. Stack and heap overflow are detected via the use of *dead zones* 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".

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.

2.3 Stack Sharing

Hybrid stack sharing [27] creates a fixed number p of stacks in memory, and attempts to evenly distribute n threads among them using a round-robin approach, where $p \leq n$. On a context switch, if all stacks are used, the used portion of the exiting thread's stack is 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. Hybrid stack sharing improves upon the standard multi-threaded stack handling approach by introducing a constant amount of memory fragmentation, as there is a limited number of "large" stacks that take up address space (heap fragmentation aside). It addresses the issue that some threads may require a small and some may require a large stack.

Multitask 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 standard mechanism as 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. 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, total stack space is still of 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 an activation 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.

A *meshed stack* [14] is a call stack mechanism where all threads place their activation frames at the top of one common stack. When an activation 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 [14] gives only an overview referencing a thesis that is in preparation for further details.

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.

Stackless Python [24] is an unfortunately misleading name, but the call stack mechanism it uses is interesting nonetheless. Standard "stackful" 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 activation 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 [22] is a multi-threaded stack implementation for gcc [1]. To begin with, each thread gets its own contiguous stack space, just like the standard 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 standard 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 an activation frame cannot fit into the remaining space in a stack segment. Google's Go Language [9] implements this by initially allocating 4 KB for each thread ("goroutine") and having the linker insert a preamble at each procedure (function) call. When overflow is detected, a new stack page is allocated and linked to the previous stack page.

Capriccio [25] is a user-level thread package that uses a call stack mechanism that can be viewed as a refinement of Thread Segment Stacks. The major change that Capriccio makes is that it analyzes the call graph of a program 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 procedure 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.

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
Multitask 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

Table 1. Stack Implementation Summary

Capriccio, like Thread Segment Stacks, still suffers from a degree of call stack memory fragmentation. However, Behren et al. [25] 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 Summary

Table 1 reviews various features of the presented multi-threaded stack mechanisms.

Runtime Overhead The overhead above what the standard 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.

3. Experimental Setup

3.1 Criteria for Selection of Stack Mechanisms

Our 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.

The following methods do not meet the above criteria and were not selected for experimentation.

- All methods from Section 2.2 lack 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 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.
- Multitask stack sharing 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 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.

3.2 C--

The considered stack mechanisms 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 two reasons: First, 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 understanding beyond public interfaces to modify the instruction calling sequence. In particular, it was unknown if some optimizations relied on a contiguous stack frame. Secondly, we wanted to be under complete control of the optimizations that could distort timing measurements of procedure calls, like inlining and elimination of tail recursion, in order to isolate the effects of different calling mechanisms.

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. This C-like language is a subset of the C language and was given the unoriginal name of C--.

We give a brief overview of C--. More details can be found in [21]. The C-- compiler has no preprocessor, and accepts only one source file as input. The C-- compiler emits 32-bit x86 [6] instructions compatible with the open source assembler NASM [3]. 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. 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).

C-- supports the int, char, double, struct, void, and pointer data types. Several features of the of the C language were omitted simply because they were not needed for the experimentation: dynamic allocation of memory on the stack (stalloc), variable array declaration on the stack, variable declarations cannot have initializers, whole structs cannot be assigned (only individual struc members), function pointers, increment operators, decrement operators, and some other operators. C-- does not implement the auto, const, enum, goto, long, register, signed, static, switch, typedef, union, unsigned, and volatile keywords.

The compiler performs dataflow analysis, register allocation, peephole optimization, and includes a bottom-up rewriting systems (BURS) [13]. C-- has introduced a macro, stacksizeof(procedure), which like the C macro sizeof(type) returns the stack size for a given procedure.

4. Implemented Stack Mechanisms

4.1 Traditional Fixed-Size Stack with "Caller-cleanup"

This call stack mechanism does not meet the criteria outlined in Section 3. While we 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 independent 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 [5]. The caller is responsible for pushing arguments to the stack, as well as cleaning the stack on procedure exit.

```
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.

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

4.2 Traditional Fixed-Size Stack with "Callee-cleanup"

This mechanism also does not meet the criteria outlined earlier, but is closer to the "MMU" mechanism discussed later on and allows for a better comparison of the measurements.

Caller Instructions The caller is responsible for pushing arguments to the stack, but does not clean up the stack on exit.

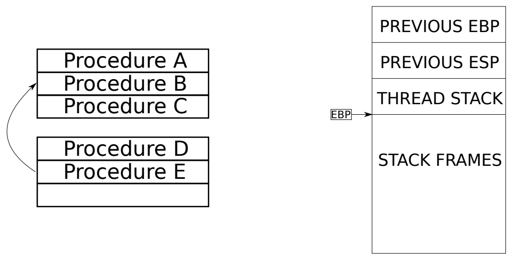
```
PUSH return_address
PUSH arg1
 ...
PUSH argn
JMP callee_name
```

Callee Instructions The callee is responsible for ensuring that the stack pointer has the same value on return as before the caller pushed the arguments on the stack.

```
callee_name : SUB ESP, callee_stack_size
               ...                #Body of procedure
              ADD ESP, callee_stack_size + arg_size
              JMP [ESP]
```

4.3 Per Procedure "Heap" Allocation

The call stack for a program is structured as a linked list allocated on the heap. Each procedure invocation has its own stack frame, just large enough to hold the callee's activation frame as well as a pointer to the caller's stack frame. The caller first allocates a new stack frame, then pushes the arguments on the new stack frame (while referring to its on stack frame), calls the callee, and finally deallocates the stack frame. Allocation is done by calling malloc [7], which requires its own stack space. Neither the current nor the new stack frame can be used for that, hence a per-thread stack region is reserved for this purpose. As the caller refers to three non-contiguous stack frames, the details of the caller's instruction are more involved and are explained in [21] The callee's instructions are identical to those of "Callee-cleanup".



(a) (b)

Figure 2. (a) Linked Stack Chunks for A → B→ C→ D→ E and (b) Stack Chunk for Look-Ahead Overflow Detection

4.4 Linked Stack Chunks with "Look-Ahead" Overflow Detection

The call stack for a program is structured as a linked list of *stack chunks*. Unlike "Heap", where each procedure has a region of memory dynamically allocated containing just one stack frame, this mechanism employs the use of stack chunks which may contain many stack frames, as depicted in Figure 2 (a). When a procedure call would cause a stack chunk to overflow, a new stack chunk as in Figure 2 (b) 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 [25] 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 of "Caller-cleanup".

```
      MOV EAX, ESP
      MOV EDX, ESP
      ADD EDX, (STACK_CHUNK_SIZE
                −LONGEST_PATH(callee_name)−16)
      CMP EDX, EBP
      JGE  .L1
      CALL STAMEX_OVERFLOW_HANDLER
.L1:  PUSH arg1
       ...
      PUSH argn
      CALL callee_name
      ADD ESP, args_size
      CMP EBP, ESP
      JNE  L2
      CALL STAMEX_UNDERFLOW_HANDLER
.L2:  ...
```

The overflow handler allocates a new stack chunk by calling malloc, saves the previous values of EBP and ESP, and reserves a new thread stack, as depicted in Figure 2 (b). Since a subsequent call to malloc (and free) requires its own stack space, a thread stack needs to be reserved for this purpose. The stack pointer ESP is set such that on return from the handler, the caller can push all the parameters on the stack in the possibly newly allocated chunk. The underflow handler is responsible for restoring the previous stack chunk and freeing the current stack chunk.

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. The instructions for this are identical to those for "Caller-cleanup".

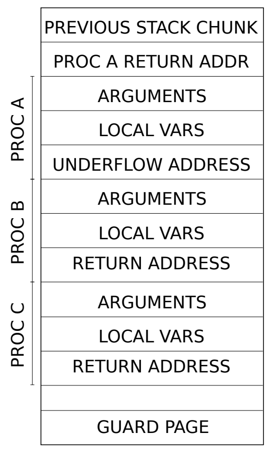


Figure 3. Stack Chunk for MMU Overflow Detection

4.5 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 2 (a). On overflow, a new stack chunk as depicted in Figure 3 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.

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). All other procedures in the stack chunk store their actual return address in the stack frame ('PROC B' in Figure 2) (a). 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.

Caller Instructions 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 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.

```
              MOV EAX, ESP
              MOV EDX, return_label
              MOV [ESP−callee_stack_size ],  EDX
              PUSH arg1
               ...
              PUSH argn
              JMP  callee_name
return_label :   ...
```

The C-- compiler assumes that the stack frame for a procedure is always smaller than the guard page, otherwise the attempt to write the return address to the stack may overwrite a location below the guard page rather than generate a SIGSEGV. In that case, extra instructions need to be inserted before attempting that: MOV [ESP−(callee_stack_size−guard_page_size)], EDX would hit the guard page if $\text{guard\_page\_size} < \text{callee\_stack\_size} \leq 2\times$ guard_page_size. If callee_stack_size is even larger, further instructions need to be inserted.

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.

```
callee_name :  SUB ESP,  callee_stack_size
                ...  #Body of procedure
               ADD ESP,  callee_stack_size + arg_size
               JMP [ESP−( callee_stack_size + arg_size )]
```

Stack Overflow and Underflow When SIGSEGV is generated, the signal handler will allocate a memory aligned chunk by calling memalign, will protect the guard page by calling mprotect, ensuring that any accesses will cause a SIGSEGV, link the previous 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. As the C calling conventions differ from the "MMU" calling conventions and as calls to memalign and mprotect need their own stack space, significant bookkeeping is needed; the details are in [21].

5. Experiments

In order to isolate the overhead of procedure call mechanisms from other computations, three programs with little computation but extensive calls were selected as usage profiles, each with different characteristics: Summation, Unbalanced Binary Tree, and Quicksort. Some experiments have 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" [4]). A "quantity" variation tests scalability across a number of threads which greatly exceeds available cores in the system. In multi-threaded experiments, the stack address space of 1 GB is divided equally among each thread, so to keep the total used memory constant for avoiding impacts of the virtual memory management. For stack chunks, the size is 8 pages or 32 kilobytes, excluding the space for the guard page if applicable.

Deep Summation. This program sums the numbers from 1 to n recursively.

```
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

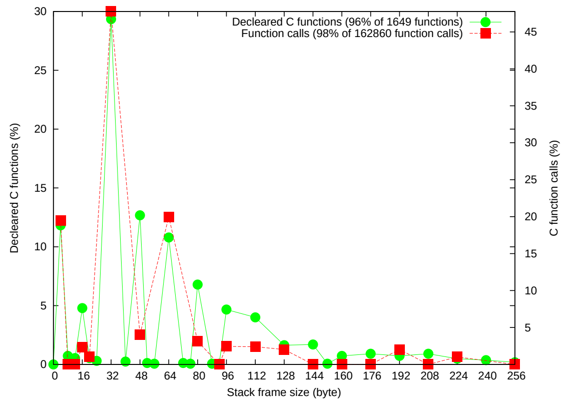


Figure 4. Distribution of stack frame sizes of Gnuplot

procedure with small stack frame and that contains a minimum of computation. In the multi-threaded version of this experiment, each thread sums the numbers from 1 to n / 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.

Big Summation. This experiment aims to test the effect the stack frame size has on the various stack implementations. Programs allocate stack frame of various sizes. To understand what the typical distribution of stack frame sizes is, we analyzed Gnuplot 4.6.0 (we analyzed three other well-known open-source programs as well to be sure that Gnuplot is representative). Figure 4 shows the relative frequency of declared C functions for stack frame sizes from 4 to 256 bytes and the relative number of calls in a typical run. It turns out that 98% of function calls are to functions with a stack frame of 256 bytes or less and about 30% are to functions with a stack frame size of 32 bytes. The average is about 50 bytes.

The function summation above has a stack frame of 8 bytes (4 for the return address and 4 for the parameter). We have modified it by allocating local variables to increase the stack frame size to 16, 32, 48, 64, 80, 96, 112 and 128 bytes.

Unbalanced Binary Tree. This experiment is an implementation of an ordered 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. The program will search 70% of the time for a random integer contained within the 20 level deep balanced portion of the binary tree and 30% of the time the for the maximum value in the binary tree, triggering a spike in stack usage.

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.

Quicksort The implementation is taken from [16]. This experiment is meant to be representative for programs that don't have a deep calling structure but instead contain some computation (here the comparisons and swaps). We compare various stack mechanisms for sorting 10^6, 10^7, and 10^8 random integers with a single-threaded version only. As the calling structure is so shallow that all stack frames are in the first chunk of MMU and Look-Ahead, there would be no contention between threads in a multi-threaded version.

6. Results

The experiments were run on the following processors:

Pentium 4 launched in November 2000, 3.2GHz, containing 42 million transistors, is based on the NetBurst architecture featuring a very deep instruction pipeline to achieve a high clock speed.

Core 2 Duo launched in May 2007, 1.8GHz, containing 291 million transistors, has 2 (physical) processor cores.

Sandy Bridge I7 launched in January 2011, 3.4GHz, containing 1.16 million transistors, has a 14-17 stages pipeline and has 4 physical cores and 8 logical cores through hyper-threading technology.

Haswell I7 launched in August 2013, 3.4GHz, containing 1.4 billion transistors, has a 14-19 stages instruction pipeline and has 4 physical cores and 8 logical cores through hyper-threading technology. It improves the back end of the pipeline: the instruction decode queue is not statically partitioned between the two threads at each core can service.

All measurements were with Gentoo 3.10.7 in single-user mode. As a locally-compiled system, Gentoo builds optimized code for the underlying architecture. The results reported here are the average over sixty individual runs of the experiments. The difference between the maximum and minimal value, as reported in Table 2 for one set of experiments, were small enough, particularly for larger running times, that only the average value is reported.

6.1 Impact of Processor Architecture

In the first experiment we analyze the impact of the processor architecture on the relative efficiency of the procedure calling mechanism. We use single-threaded Deep Summation with different depths of recursion, as Deep Summation makes most use of the stack; the results are reported in Figure 5. As expected, the running time is linear to the sum being calculated. However, while on older processors MMU, Look-Ahead, Caller-cleanup, and Callee-cleanup perform nearly identical, the newer the processor the better MMU and Callee-cleanup perform: Look-Ahead and Caller-cleanup perform half the cleanup in the callee and half the cleanup in the caller, resulting in one more instruction. A possible explanation is that the deep pipeline of Pentium 4 can cope with that better than newer processors.

Caller-cleanup and Callee-cleanup use a fixed stack size of 1GB and are not scalable, whereas MMU and Look-Ahead allocate chunks of 8 pages (plus 1 guard page for MMU). MMU and Callee-cleanup have similar caller and callee sequences, however, the extra overhead for allocating chunks makes MMU slower than Callee-cleanup: for summing up to 100 millions, MMU and Look-Ahead allocate approximately 24420 chunks (due to internal fragmentation, Look-Ahead needs 5 more chunks than MMU). Surprisingly, MMU is more efficient than Caller-cleanup, the standard gcc convention, which itself is marginally faster than Look-Ahead.

6.2 Impact of Usage Profile in Single-Threaded Runs

All the remaining experiments were carried out on Haswell I7, the newest processor available to us.

Deep Summation. As can be seen in Figure 6, the overhead from a dynamic memory allocation call (malloc) for every procedure with the Heap mechanism is significant. The figure also gives the times of gcc without optimization ("gcc") and gcc with optimization ("gcc -O2"). Our C-- compiler with Caller-cleanup performs somewhere in between. The figure also magnifies the observations from the first experiment.

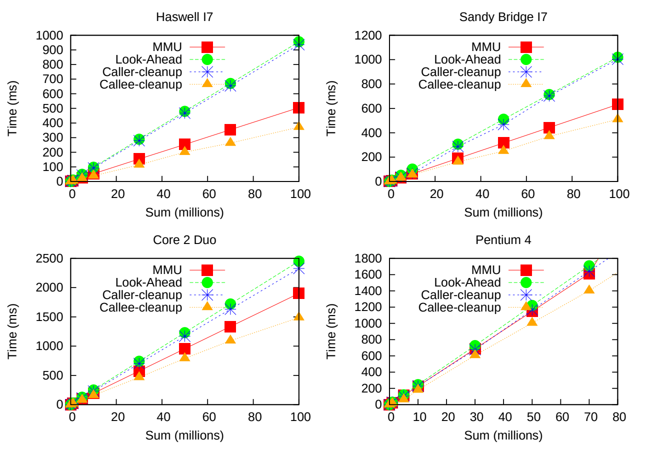


Figure 5. Single-threaded Deep Summation on Haswell I7, Sandy bridge I7, Core 2 Duo and Pentium 4

Threads	1	2	3	4	5	6	7	8
Maximum	451	551	630	720	824	927	1060	1193
Average	430	529	615	701	806	911	1042	1156
Minimum	417	502	586	659	778	898	1014	1125

Table 2. The original times in ms of the Unbalanced Binary Tree multi-threaded Look-Ahead experiment over 60 runs

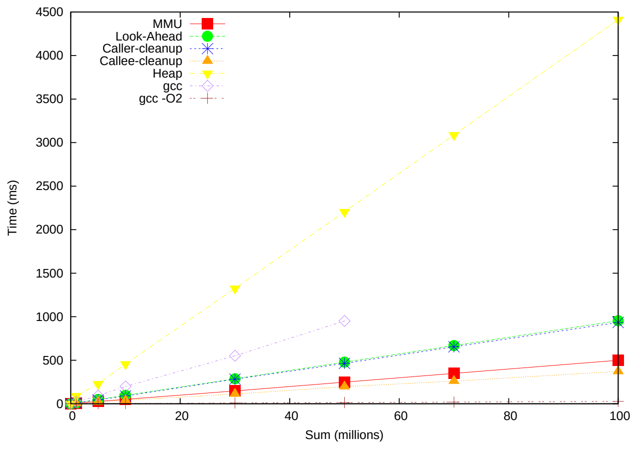


Figure 6. Deep Summation Single Threaded

Unbalanced Binary Tree. As can be seen in Figure 7, the overhead from a malloc call for every procedure with the Heap mechanism continues to be very significant. The trends observed in Summation continue to hold, the only difference is that the Caller-cleanup mechanism runs faster than the MMU: the overhead from dynamic memory allocation for every stack chunk is more significant significant than the overhead of Caller-cleanup compared to Callee-cleanup.

Quicksort. As can be seen in Figure 8, the maximal call depth (including auxiliary functions) is so shallow that all computation remains within the first chunk. With MMU, a guard page is not hit

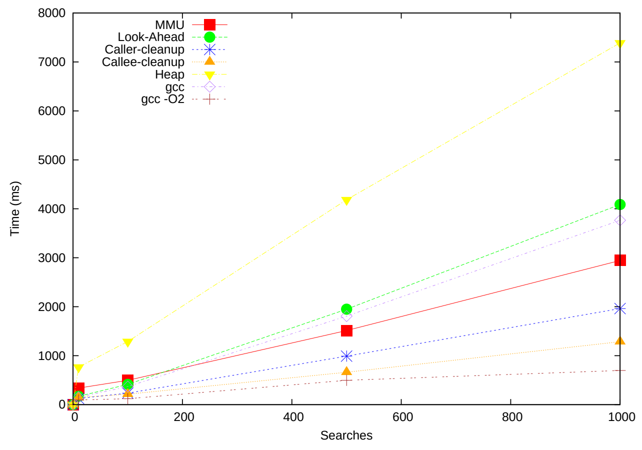


Figure 7. Unbalanced Binary Tree Single Threaded

Elements	10^6	10^7	10^8
Callee-cleanup	152 ms	1,686 ms	19,086 ms
MMU	156 ms	1,737 ms	19,618 ms
Caller-cleanup	157 ms	1,755 ms	20,007 ms
Look-Ahead	159 ms	1,775 ms	20,257 ms
Number of calls	14,703,523	160,540,046	1,855,875,685
Maximal depth	19	21	25

Figure 8. Quicksort Single Threaded

and MMU performs consistently better than Look-Ahead. However, the difference is at most 3%.

6.3 Impact of Usage Profile in Multi-Threaded Runs

Each multi-threaded experiment has two variations: The ”cores” variation tests one to eight threads to test scalability over the available cores. A ”quantity” variation tests scalability across a number of threads which greatly exceeds available cores in the system.

Deep Summation. The times reported in Figure 9 are the total running time for all threads for summing from 1 to n / number_of_threads. The MMU mechanism, while starting out with better performance than Caller-cleanup and Look-Ahead, demonstrates the worst scalability, and eventually the worst performance, as the number of threads exceed the number of available cores. To isolate the cause, we introduced a *stack chunk reuse* mechanism: rather than deallocating stack chunks, they are placed in queue for future use. On allocation, first chunks from that queue are used before a new chunks is allocated through memalign and mprotect. Calls to mprotect take more than 100 times than calls to memalign for allocating page-aligned memory, which itself takes about twice as long as malloc. MMU need mprotect and memalin. Calls to mprotect cause the processor’s TLB to be flushed, thus incur a heavy penalty. The new mechanism is called “MMU-with-reuse”, the old mechanism is renamed to “MMU-without-reuse”. As can be seen in Figure 10, the MMU-with-reuse mechanism has a better performance than Look-Ahead when the number of threads exceeds 200. The reason is that the concurrency is so high that some threads manage to start their cleanup phase while the others are still in their growth phase, in this case, the stack chunks are able to be reused, meaning there are fewer calls to malloc and mprotect. To magnify this effect, when summation is repeated 10 times, MMU outperforms Caller-cleanup and Look-Ahead, see Figure 11.

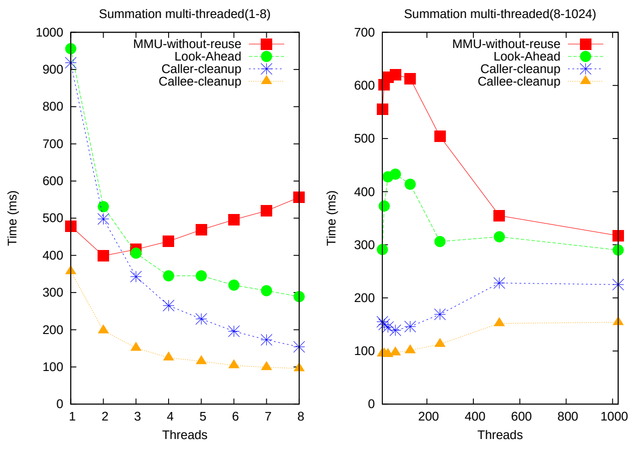


Figure 9. Summation multi-threaded (MMU without reuse)

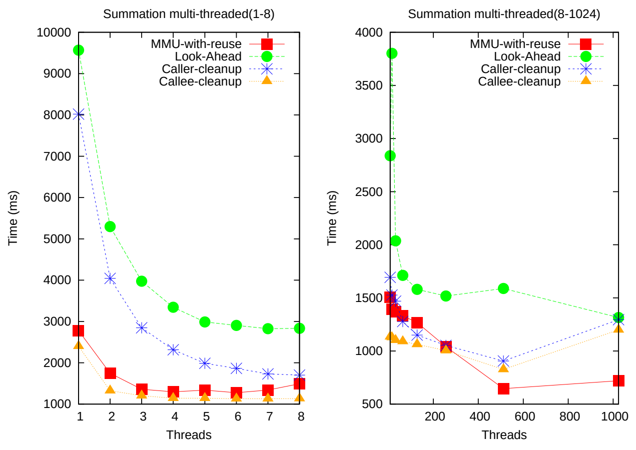


Figure 11. Summation multi-threaded (MMU with reuse), 10 repeats

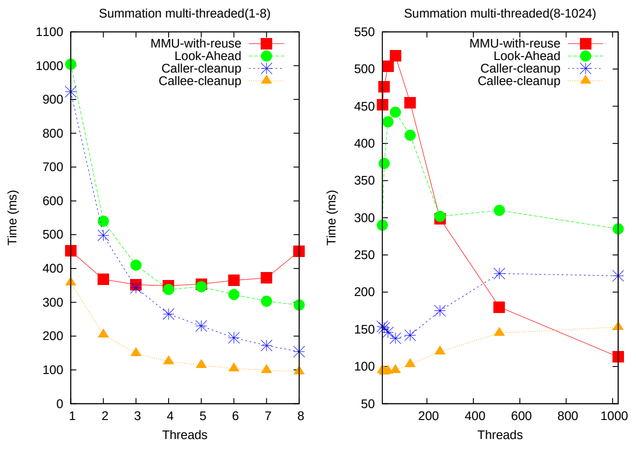


Figure 10. Summation multi-threaded (MMU with reuse)

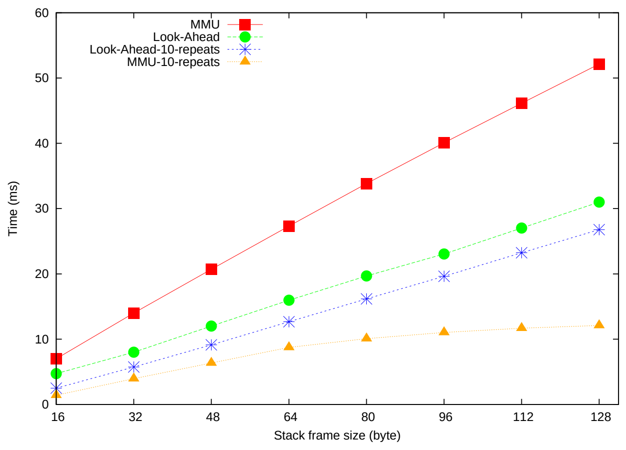


Figure 12. Big Summation multi-threaded (MMU with reuse)

Big Summation. As evident from Figure 12, there is almost a linear increase in the time with the increase for the stack frame size due to the need for allocating memory, despite the same computation taking place. For a single run, MMU performs worse than Look-Ahead because of the overhead of calling mprotect. However, if the runs are repeated 10 times and chunks are reused, MMU outperforms Look-Ahead significantly.

Unbalanced Binary Tree. The time reported in Figure 13 is the total time for all threads to finish. MMU-with-reuse scales identically to Caller-cleanup and Look-Ahead. MMU-with-reuse continues to show better scaling than Look-Ahead.

Neither the Caller-cleanup nor Callee-cleanup were tested when the number of thread exceeds 8, as the high concurrency combined with the tendency of 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.

The spike in Look-Ahead in Figure 13 results from mutex contention. When threads simultaneously allocate and free memory, there could be contention for mutexes used in malloc. To scalably handle memory allocation in multi-threaded applications, glibc creates memory arenas if mutex contention is detected. Each arena is a large region of memory from which memory is allocated. The number of arenas usually equals to the number of cores. When the number of thread exceeds the number of cores, mutex contention increases.

7. Conclusions

Two stack calling mechanisms were identified that satisfy the given criterions, MMU and Look-Ahead. To summarize, MMU tends to perform worse that Look-Ahead, if (1) there is a deep recursion without repeats (so the overhead of mprotect-ing does not amortize), (2) the stack frame size is large (so the guard page is more frequently hit), and (3) a larger number of short-lived threads all start at the same time (so the overhead of mprotect-ing does not amortize). However, none of these three situations are typical. Thus, one may conclude that MMU performs better in practice, particularly for languages that allow arrays to be allocated only on the heap and thus have small stack frames. However, (extrapolating Quicksort) the differences are not big and there can be situations where MMU degrades. Still, (extrapolating Quicksort again), the differences between fixed stack size (Callee-cleanup and Caller-cleanup) and automatic stack size management (MMU and Look-Ahead) are so small that there no reason to keep programmers guessing the stack size at thread creation.

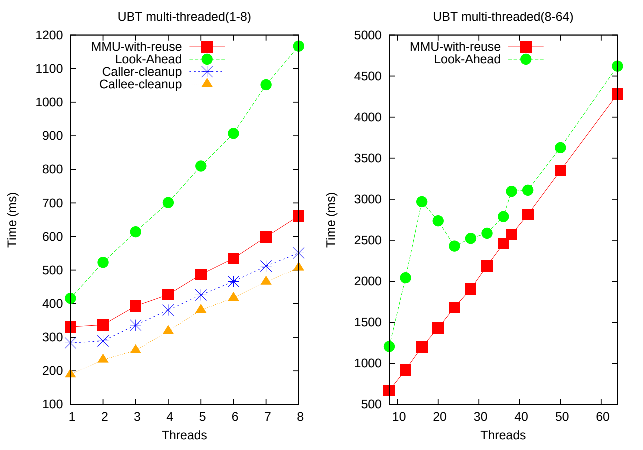


Figure 13. Unbalanced Binary Tree multi-threaded

While we tried to make the comparison as fair as possible, there are limits: for Look-Ahead, a call-graph analysis is used; no analysis of similar complexity is done for MMU. Also, Look-Ahead suffers more from internal fragmentation as stack frames for a number of procedures are allocated at once; we did not measure that effect. On the other hand, MMU uses Callee-cleanup, whereas Look-Ahead uses Caller-cleanup; one could modify Look-Ahead to use the faster Callee-cleanup sequence as well.

A surprise to use was that Callee-cleanup offers a measurable advantage over Caller-cleanup, the standard C calling convention.

It should also be noted that the C-- compiler has no optimizations that could reduce or eliminate procedure calls. Implementing the most promising stack mechanisms into an existing professional compiler framework would allow for better comparisons of more complex real world programs. Finally, our experiments involved only processors of the Intel x86 family. It remains to be seen if our conclusions apply to other processor families as well.

Acknowledgments

Measurements were run through the Datamill Project of the Real-Time Embedded Software Group at the University of Waterloo. We thank the Datamill project members for their support. We are also grateful for the financial support from China Scholarship Council.

References

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

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

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

[4] *Intel Technology Journal, Special Issue on Hyper Threading Technology*, volume 6, 1, February 2002.

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

[6] Intel 64 and IA-32 Architectures Software Developer's Manual, May 2011. URL http://download.intel.com/design/processor/manuals/253665.pdf.

[7] malloc man page, The Linux *man-pages* project, May 2012. URL http://man7.org/linux/man-pages/man3/malloc.3.html.

[8] pthread_create man page, The Linux *man-pages* project, August 2012. URL http://man7.org/linux/man-pages/man3/pthread_create.3.html.

[9] The go programming language, November 2013. URL http://golang.org.

[10] AMD. Multi-Core Processing with AMD, November 2013. URL http://www.amd.com/us/products/technologies/multi-core-processing/Pages/multi-core-processing.aspx.

[11] ARM. ARM11MPCore Processor, November 2013. URL http://www.arm.com/products/processors/classic/arm11/arm11-mpcore.php.

[12] A. R. Disteli and P. Reali. Combining Oberon with active objects. In *Proceedings of the Joint Modular Languages Conference on Modular Programming Languages*, pages 221–235. Springer-Verlag New York, 1997.

[13] D. Grune, H. E. Bal, C. J. H. Jacobs, and K. G. Langendoen. *Modern Compiler Design*. John Wiley & Sons, NY, 2001.

[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. URL http://sunsite.informatik.rwth-aachen.de/Publications/AIB/1993/1993-03.ps.gz.

[15] Intel. Multi-Core Processor Architecture Explained, August 2013. URL http://software.intel.com/en-us/articles/multi-core-processor-architecture-explained.

[16] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall Inc., 1988.

[17] S. Lalis and B. A. Sanders. Adding concurrency to the Oberon system. In *Proceedings of the International Conference on Programming Languages and System Architectures*, pages 328–344. Springer-Verlag New York, 1994. ISBN 0-387-57840-4.

[18] Microsoft. Windows Development, October 2013. URL http://msdn.microsoft.com/en-us/library/windows/desktop/ms686774(v=vs.85).aspx.

[19] S. Microsystems. Multithreaded Programming Guide, September 2008. URL http://docs.oracle.com/cd/E19253-01/816-5137/816-5137.pdf.

[20] B. Middha, M. Simpson, and R. Barua. MTSS: Multitask stack sharing for embedded systems. *ACM Transactions on Embedded Computing Systems*, 7(4, Article 46), July 2008. .

[21] J. I. Moore-Oliva. A comparison of scalable multi-threaded stack mechanisms. Master's thesis, McMaster University, Hamilton, Ontario, Canada, December 2010. URL http://digitalcommons.mcmaster.ca/opendissertations/4172/.

[22] M. Pizka. Thread segment stacks. In H. R. Arabnia, editor, *Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications - PDPTA*, Las Vegas, NV, June 1999. CSREA Press. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.44.3522.

[23] H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, September 2005. ISSN 1542-7730.

[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. URL http://www.python.org/workshops/2000-01/proceedings/papers/tismers/spcpaper.pdf.

[25] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 268–281, New York, NY, USA, 2003. ACM.

[26] M. F. Wilding and D. A. Wood. Heap and stack layout for multithreaded processes in a processing system. US Patent 744782, November 2008.

[27] K.-F. Wong and B. Dagevill. Supporting thousands of threads using a hybrid stack sharing scheme. In *Proceedings of the 1994 ACM Symposium on Applied Computing*, pages 493–498. ACM New York, 1994. ISBN 0-89791-647-6.