MeldC Threads: Supporting Large-Scale Dynamic Parallelism

Wenwey Hseush
James C. Lee
Gail E. Kaiser

Columbia University
Department of Computer Science
500 W. 120th Street
New York, NY 10027
(212) 854-8123
Fax: (212) 666-0140
hseush@cs.columbia.edu

Technical Report
CUCS-010-92

24 March 1992

Abstract

We present a new thread model that supports large-scale dynamic parallelism, i.e., the number of co-existing threads is large and the life-time for each thread is short. We introduce the *interleaving stack* (IS), which melds the frames of multiple threads into one shared stack. The pure interleaving stack is not scalable due to severe internal fragmentation. We propose and evaluate an improvement, *circular interleaving stack* (CIS). Our simulation shows that in the domains of large-scale dynamic parallelism, CIS performs better than the traditional one-stack-per-thread (1SPT) mechanism with respect to memory utilization. The 1SPT mechanism gains better memory utilization as thread life-time increases. CIS and 1SPT show similar CPU utilization.

keywords: light-weight processes, object-oriented language, concurrency.

Hseush is supported in part by the Center for Telecommunications Research. Lee is supported in part by the Center for Advanced Technology. Kaiser is supported by National Science Foundation grants CCR-9106368, CCR-9000930 and CCR-8858029, by grants from AT&T, BNR, DEC, IBM and SRA, by the New York State Center for Advanced Technology in Computers and Information Systems and by the NSF Engineering Research Center for Telecommunications Research.

1. Introduction

Concurrent object-oriented programming models [1] promise the potential of expressing large-scale parallelism. In these models, each object is considered active and possesses its own thread(s) of control. Objects interact only by message passing. The encapsulation of data and control through objects and the orderly interaction through messages prepares a natural environment for applications that create a large number of concurrent objects. This trend is further pushed by high-speed networking and high-performance computing, which make feasible the cooperation of a large network of autonomous objects. Each opens a door to other objects, processing incoming messages and delegating computation to other cooperative objects. In such an ''open'' system, the number of interacting objects is unknown, and message arrivals are unpredictable. The expected parallelism is usually large scale and dynamic. *Large-scale parallelism* indicates that the number of concurrent threads in a network of objects is large. *Dynamic parallelism* implies that the creation and destruction of threads are dynamic, and the number of active threads is changing rapidly.

We identify the characteristics of large-scale and dynamic parallelism as likely to dominate future applications, and they are the key concerns to our design in building a concurrent object-oriented language, MELDC[1]. This paper presents a light-weight thread model (MELDCTHREADS) that supports large-scale dynamic parallelism, considering efficient resource utilization. The thread model is a generic framework that applies to object-oriented languages as well as non-object-oriented languages such as C.

Our approach starts with understanding thread behaviors by characterizing application domains and ends with designing a spectrum of light-weight thread mechanisms according to the characteristics. We justify them by performance evaluation using simulation study, which considers CPU and memory utilization. The measurement is based on two criteria: (1) CPU time for context switching as well as creating/destroying threads and (2) memory utilization

We introduce a family of thread mechanisms called *interleaving stacks* (IS). The idea is to meld the frames[2] of multiple threads into a single stack, instead of using one stack for each thread. The major problem with the IS mechanism is a scenario called *frame thrashing*, which may introduce substantial internal fragmentation. The pure IS mechanism is not scalable: memory utilization grows exponentially as the number of threads increases. However, two other members in this family, *circular interleaving stacks* (CIS) and *k-interleaving stack* (*k*-IS), show much better memory utilization than and similar CPU utilization to the traditional one-stack-per-thread (1SPT) approach in the domains of large-scale and dynamic parallelism. CIS and *k*-IS perform better when threads are short-lived and 1SPT has better

---

[1] Like C++, MELDC adopts the C language as its base language.

[2] A frame, also known as an activation record, represents the runtime local environment of a subroutine invocation.

memory utilization when threads are long-lived.

We are aware that memory capacity is increasing rapidly. We argue that in the domains of large-scale dynamic parallelism, two factors tend to eliminate the effect of increasing memory capacity. First, the increase of memory needs supersedes the increase of memory capacity. For example, a network management system manages a set of network elements (representing devices, workstations, etc.). Considering that the number of network elements is increasing rapidly, representing threads and objects for network elements requires a large amount of memory. Second, the increased number of threads complicates the interactions among them, and the interaction pattern becomes dynamic and unpredictable. This makes it difficult for programmers to predict the stack size for threads. The situation becomes worse without mechanisms to survive stack overflow.

High survivability can be achieved by either avoiding stack overflow or recovering from a stack fault when it occurs. Without assuming any semantic information about the applications, it is almost impossible to recover from a stack overflow. This is the reason that most thread packages [13, 6, 16, 11, 9, 10] ‘‘handle’’ stack overflow by aborting computation partially or entirely. We have attacked this problem by studying the concept of *thread mobility*, which characterizes the capability of relocating frames. A frame may not be relocatable if it exposes the addresses of its local variables to other program elements. We conducted a static semantic analysis on a range of C programs and found that less than 3% of C functions are not relocatable. Our approach is to relocate the relocatable frames for better memory utilization, without affecting program semantics, when a stack overflow fault occurs or available memory becomes small. The traditional 1SPT approach shows low survivability in the face of stack overflow, since it is necessary to relocate the *entire* stack for a thread and the probability that every frame on the stack is relocatable is low. The IS approach shows better survivability, since it can move an individual frame if it is relocatable. However, due to lack of space, we do not describe this analysis or our approach in this paper[3]

Real-world applications may combine both dynamic (short-lived threads) and static (long-lived threads) parallelism. A mechanism that interoperates both CIS and 1SPT is desirable. We are currently investigating this problem, but this is outside the scope of this paper. We also do not address issues concerning multiprocessors, which have been discussed in [2, 21, 14, 3, 8, 7], and also limit our discussion to language-level light-weight processes [16, 4, 7, 13, 11, 9], which do not require support from the operating system (e.g., [8, 18, 19, 5, 21, 17, 10, 20]).

[3] A full version of this paper, addressing survivability and garbage collection, and also a range of scheduling policies for threads, is available from the authors.

2. Traditional Approach and Problems

The traditional approach to concurrent threads on a uniprocessor is to supply each thread with a stack, which stores the context of the thread, and to switch the contexts of ready threads in a rapid fashion, preemptively or non-preemptively[4]. We refer to this approach as *one-stack-per-thread* (1SPT). The stack structure supports the semantics of high-level procedural languages (*e.g.*, C and Pascal). In this paper, we are concerned primarily with C, since it is the base language of MELDC.

Most current architectures support the stack notion for efficient frame allocation/deallocation. The architectures provide a stack pointer (`sp`) register, which points to the top of the stack, and perhaps a frame pointer (`fp`) register, which references the current (top) frame. Allocation or deallocation of a frame requires only a few machine instructions to move `sp` up or down, respectively, and adjust `fp`. This architectural mechanism efficiently supports the execution of single-threaded programs. Accordingly, the C compiler generates the necessary code for calling and exiting functions.

To implement multi-threaded executions with the available architectural and compiler support, the natural approach is to prepare one stack for each thread, and schedule the executions of ready threads by context-switching among the stacks. Context switching includes saving the current `sp`, `fp` and program counter (`pc`) for the current thread and restoring the new ones from the next ready thread.

Current architectures and C compilers suggest two restrictions on memory resources regarding frame allocation and deallocation:

- **Contiguous stack space**: The stack memory is a contiguous memory space. A frame is physically adjacent to the frame of its calling function (if any).
- **Non-relocatable frames**: Frames/stacks are assumed to be non-relocatable. Hardware/compiler designers usually consider no semantic knowledge of applications, and relocating frames may break application semantics. Due to lack of space, we do not discuss this problem in this paper.

The requirement of contiguous stack memory is further imposed by the ‘‘grey area’’ of frames (considering the C language). The grey area stores function parameters[5] and return values, which conceptually can be considered part of the frame of the caller or part of the frame of the callee. The grey area is actually shared by the caller and the callee, where the function parameters are initialized by the caller and are referenced/updated by the callee. It is natural to implement the grey area as the overlapped area of two contiguous frames.

The traditional 1SPT approach has inefficient memory utilization from two sources:

1. Stack sizes are determined before thread creation time. This may require hints from programmers. An

[4]Preemptive vs. non-preemptive scheduling is independent of the results in this paper.

[5]Some architectures/compilers use registers for efficient parameter passing, when the number of parameters is small.

imprecise estimation implies inefficient memory utilization or possibility of stack overflow.

2. Even though the stack size (**SM**) for each thread is precisely determined, the memory used for the frames of all active threads at any time during execution is likely to be much less than the total memory ($\sum$ **SM**) of all stacks. Consider, for example, that whenever one thread consumes all its memory, others always have lots of available memory.

The problem of inefficient memory utilization becomes severe when the number of threads increases dramatically. One network management program written in MELDC employs about 20,000 threads. It would be nearly impossible to implement it using the 1SPT approach.

We first identify the restriction of contiguous stack space as the main source for poor memory utilization. The interleaving stack model is an approach to relaxing the contiguity condition with compiler or hardware support.

3. MELDCTHREADS Model

Our light-weight thread mechanisms are developed under a general threads model, which serves as a framework for reasoning about thread behavior as well as resource utilization. The model consists of three levels: application behavior, scheduling policies, and resource models.

3.1. Application Behavior

The behavior of application programs can be understood with three factors: (1) thread number, (2) thread depth, and (3) thread stability.

Thread Number

The thread number of an application program is the number of threads that potentially co-exist during execution. In general, application domains can be categorized into four classes: (1) large-scale parallelism (1000 threads or more), (2) medium-scale parallelism (20-1000 threads), (3) small-scale parallelism (less than 20 threads), and (4) unbounded parallelism, where the number of threads is non-deterministic. The numbers used to divide the scale of parallelism may not reflect the complexity in a precise way. However, the numbers serve as some sense of standards for the complexity introduced by the thread numbers.

Thread Depth

The term ‘‘depth’’ refers to the resources required for a thread; this makes the distinction between light-weight processes and heavy-weight processes. Our discussion on light-weight threads concerns mainly the memory resource for frames. The depth of a thread is the number of frames the thread currently has. Formally, thread depth is a distribution function over time. In general, we are interested in the maximum depth of threads.

Thread Stability

Thread stability indicates how stable the thread state remains. The thread state changes when a thread is created or destroyed. The stability is determined by thread creation rate and thread destruction rate. In our simulation, we use the inverse factors: time interval between a thread destruction and thread creation (we call it inter-creation time, even though it is not the time interval between two creations), and thread life-time. In general, there are two types of applications characterized by thread stability.

1. Static parallelism: In this type of application, most threads are created at initialization, and destroyed when the program exits. In between, threads might be created or destroyed occasionally and slowly. The number of threads is stable and predictable.

2. Dynamic parallelism: In this type of application, threads are created dynamically and are destroyed after a short time. The system has high thread creation and destruction rates, and the rates change from time to time. The number of threads is unstable and usually unpredictable.

3.2. Scheduling Policies

This component describes (1) the policies of scheduling threads and (2) the policies of scheduling housekeeping activities to maintain efficient memory utilization. We do not discuss any scheduling issues in this paper. We simply assume that the scheduling policy for ready threads is Round-Robin.

3.3. Resource Models

A resource model describes data structures as well as mechanisms for utilizing memory resources with respect to managing frames. A resource model has five components: a structure of memory resources, a mechanism for creating a thread, a mechanism for destroying a thread, a mechanism for allocating a frame, and a mechanism for freeing a frame. The memory allocated for stacks is referred to as *stack memory*.

The efficiency of memory utilization is measured by *thread memory size*. The thread memory is the memory previously allocated to active threads that cannot be claimed by newly created threads. For 1SPT, thread memory is the entire stack memory allocated to active threads. No new threads can use any memory previously allocated as stack memory, unless it is released back to heap space. For example, the thread memory size for 1000 active threads, 16K bytes for each, is 16M bytes. For the IS approach, the thread memory is not the entire stack memory, but the memory below `sp` (see the next section). A new thread allocates its first frame on top of the shared stack.

4. Interleaving Stack

We have investigated five resource models, 1SPT, IS, CIS, *k*-IS and the heap-based (HB) approach. Almost every thread package [13, 6, 16, 11, 9, 10] uses the 1STP approach except the SHARED tasks in the C++ task library [4], where tasks share a single stack (of some fixed size) and context-switching is done by copying the entire stack of the running task out to a save area and copying the resuming task from its save area back to the stack. The stack is shared in the sense of time-multiplexing; only one task occupies the stack at any moment. This is different from the IS approach, where the stack is shared in the sense of space-multiplexing; all active threads occupy different locations in the stack at the same time.

The HB approach allocates individual frames from the heap space. HB does not need stack memory and `sp`, but only `fp`. No thread package uses the HB approach to allocate individual frames, due to inefficient CPU utilization. The HB approach is the ideal approach to efficient memory utilization, since it allocates frames directly from heap space, where sophisticated data structures are maintained. Our intention is not to design or evaluate any heap-based algorithm, but to treat the HB approach as the optimal memory utilization for comparison. In our performance study, we simply use `malloc/free` functions from the C library.

An interleaving stack is a stack shared by all threads, which shuffle their frames on it. A similar model was proposed by Bobrow and Wegbreit [12]. They also shuffle multiple stacks into one, but require that each new frame be allocated directly above its caller due to the grey area of parameters between caller and callee. If there is sufficient free space immediately above a calling frame due to previous internal fragmentation, the new frame is allocated in the hole; otherwise, the calling frame is moved to the top of the shared stack prior to completing the call (it is not necessary to also move the caller's caller, and so on). This approach requires complex and expensive storage management. We avoid this by removing the grey area from our compiler techniques, so we can always allocate the new frame on the top of the interleaving stack.

The IS resource model is described as follows.

Resources:

One contiguous stack memory space (**SM**) is reserved at the beginning of program execution. The global register `sp` divides **SM** into two areas. The area below `sp` is the interleaving stack (IS), and the area above `sp` is the free memory space for frame allocation. Each frame maintains one link ($\mathbf{L_{prev}}$) and one flag. The link points to the previous frame (if any), which is the frame adjacent to and below it[6]. Note that this frame may belong to a different thread. The flag indicates the status of the frame, which can be either ''allocated'' or ''empty''; empty frames are the memory space not currently allocated to any thread. Initially, `sp` points to the bottom of **SM** (no frame is in the IS area).

---

[6] Every frame in the interleaving stack family needs to maintain another link, $\mathbf{L_{caller}}$, which points to the frame of the caller function.

Allocation:

Allocating a frame is to move the `sp` up according to the size of the frame. The new frame is marked ‘‘allocated’’ and the link $\mathbf{L_{prev}}$ is set. The new frame becomes the top frame of the stack.

Deallocation:

Freeing a frame is to mark the frame ‘‘empty’’. If it is currently the top frame of the stack, move the `sp` down through the links $\mathbf{L_{prev}}$ repeatedly until it reaches a frame marked ‘‘allocated’’.

Creation:

Creating a thread is do nothing except allocating a new frame (see allocation) for the first function to be executed.

Destruction:

Destroying a thread is simply to free every frame of the thread (see deallocation).

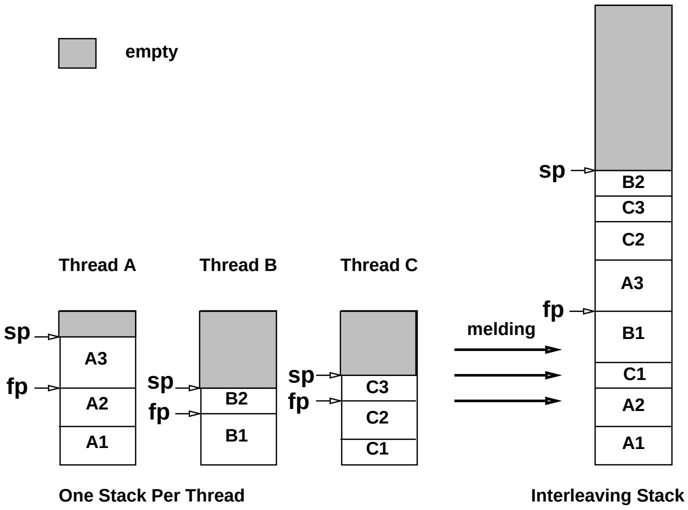

Figure 4-1: An Example of Interleaving Stack

One example is shown in figure 4-1 with three threads A, B and C running (a) with the 1SPT mechanism and (b) with the interleaving stack. Thread A currently has 3 frames, B has 2 frames and C has 3 frames. Frames A3, B2 and C3 are the top frames for A, B and C, respectively. In the 1SPT approach, `sp` points to the free memory space adjacent to the top frame on the stack and `fp` points to the top frame on the stack. In the IS approach, `sp` also points to the free memory space adjacent to the top frame on the stack. However, `fp` does not necessarily point to the top frame on the stack: it points to the top frame of one of threads (A3, B2 or C3), the currently executing one. Conceptually, the interleaving stack can be viewed as the construction of the three individual stacks by shuffling them into one.

The IS approach is based on a significant change to current compiler techniques, which assume the contiguity condition that a frame is physically adjacent to the frame of its calling function. The key to relaxing the condition is to preserve the integrity of frames by eliminating the ‘‘grey area’’ and assume no adjacency between the caller’s frame and the callee’s frame. Input parameters are kept in the callee’s frame, instead of being in an overlapped area between the caller's frame and the callee's frame. Parameters to a function can be passed through registers or by copying them directly into the callee's frame. The return value can be passed through a register or by copying it directly into the caller's frame. Each frame needs to maintain a link ($\mathbf{L_{caller}}$) to its caller's frame in order to pass a return value and resume its caller function. `fp` serves the purpose of referencing input parameters as well as local variables, and `sp` is used only for frame allocation/deallocation.

4.1. Problems: Internal Fragmentation and Frame Thrashing

When a frame that is not the top frame on the interleaving stack is freed, the memory space is not immediately reclaimed for reuse, since `sp` is not moved down until every frame above the frame becomes empty. This leaves the memory space unavailable, even though the frame is not allocated to any thread. The scenario is called *internal fragmentation*. The thread memory of an interleaving stack is the memory space from the bottom of the stack to the place pointed to by `sp`.

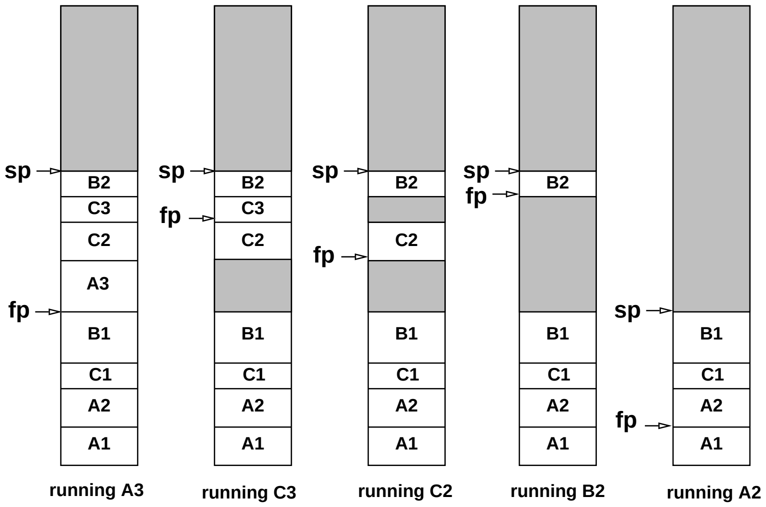

Figure 4-2: Fragmentation in Interleaving Stack

An example is shown in figure 4-2 with the scheduling order, A3, C3, C2, B2, A2, etc. Internal fragmentation becomes more and more severe until the completion of B2 (`sp` is moved down).

A scenario called *frame thrashing* may bring a severe problem to the interleaving stack, because it causes substantial internal fragmentation. This happens when frame allocations (by function calls or thread creations) occur too often, such that the probability of freeing the top frame on the stack is low. It leaves a large number of empty frames in the middle of the stack, which cannot be reused until after they are reclaimed. For example, a thread T1 creates thread T2, then terminates itself. T2 creates T3 and ends. T3 creates T4 and ends, and so on. This application runs out of stack memory with huge internal fragments (figure 4-3).

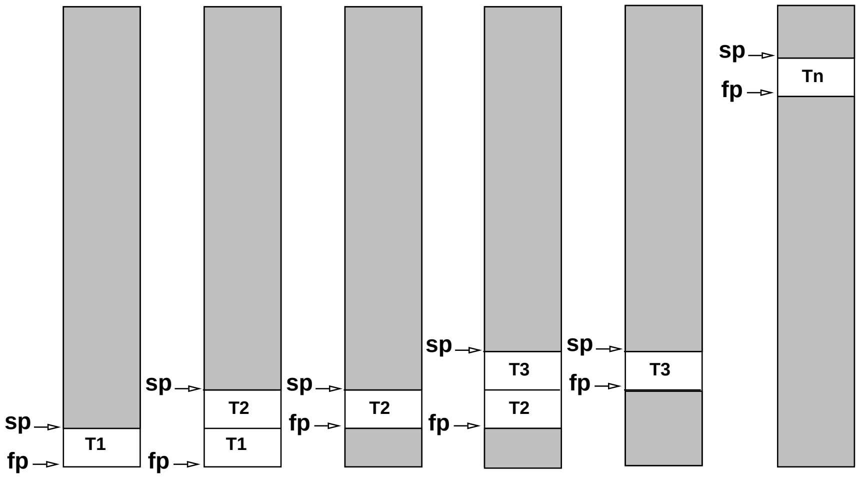

Figure 4-3: Frame Thrashing

The pure interleaving stack is thus not scalable. While the number of threads increases, the probability of freeing the top frame on the stack decreases. This results in very poor memory utilization.

4.2. Sliding Allocation Window and Splitting Windows

We observe that the allocated frames tend to be gathered within a small portion of memory that is moving toward the upper boundary of the stack. We refer to this phenomenon as *sliding allocation window*. Recall that the first allocated frame on the top is referred to as the top frame. We refer to the first allocated frame from the bottom of the stack as the bottom frame. The *allocation window* is the memory space from the bottom frame to the top frame.

In the domains of large-scale dynamic parallelism, the life-time of threads is typically short. When scheduled fairly, an old thread is likely to terminate before a new thread does. This implies that the frames of an old thread are likely to be freed before those of a new thread. Thus, threads behave in an FIFO-like (First-In-First-Out-like) fashion; the first thread to be created is likely to be the first thread to be destroyed. Since a frame of an old thread is likely to be located below one of a new thread, we might see a scenario where new frames are allocated on the top of the stack while the old frames are deallocated from the bottom of the stack. A second type of behavior, LIFO (Last-In-First-Out), determines frame allocation/deallocation for each individual thread; the last frame allocated for a thread is freed first.

The FIFO-like behavior and the LIFO behavior dominate frame allocation/deallocation for threads in the interleaving stack. When the LIFO behavior supersedes the FIFO-like behavior, the stack grows and shrinks, and the window does not move. When the FIFO-like behavior supersedes the LIFO behavior, the allocation window moves towards the upper boundary of the stack.

The behavior of sliding allocation window manifests why the pure interleaving stack is not scalable. Even though the size of the window may be small, the window may move fast and hit the boundary of the stack, causing a stack overflow.

However, if there exists one thread (called a persistent thread) that lives from the beginning of the program to the end, the window is as large as the thread memory (the memory below `sp`). In this case, the window can be viewed as being divided into two subwindows, one persistently near the bottom of the stack and the other moving towards the boundary of the stack. We refer to this scenario as *splitting windows*.

The circular interleaving stack is designed to handled the case without persistent threads, and the *k*-interleaving stack is designed to handle the case with a few persistent stacks. For the case with many persistent threads, the traditional 1SPT may be the best approach.

4.3. Circular Interleaving Stack

CIS treats the stack memory space as a circular memory, and circulates the allocation window around the memory space. The memory required to support CIS is the maximum allocation window size. We add a new register, `bsp`, which always points to the bottom frame of the allocation window. The resource model is described as follows.

Resources:

One contiguous stack memory space (**SM**) is reserved and treated as a circular memory space. Two global registers `sp` and `bsp`, which are initially set to the bottom of **SM**, point to the top of the stack and the bottom of the stack, respectively. The area between `sp` and `bsp` is the IS area. The rest of **SM** is the free memory space for frame allocation. Each frame in IS maintains two links ($\mathbf{L_{prev}}$ and $\mathbf{L_{next}}$) and one flag. The link $\mathbf{L_{prev}}$ points to the previous frame, if any, and the link $\mathbf{L_{next}}$ points to the next frame. The flag indicates the status of the frame (allocated or empty).

Frame Allocation:

Allocating a frame is to move `sp` up or around the stack memory **SM** (if it hits the upper boundary of **SM**), according to the frame size. $\mathbf{L_{prev}}$ of the new frame and $\mathbf{L_{next}}$ of the previous frame are set and the new frame is marked ‘‘allocated’’.

Frame Deallocation:

Freeing a frame is to mark the frame ‘‘empty’’. If the frame is currently the top frame of the stack, move `sp` down through the links $\mathbf{L_{prev}}$ repeatedly until it reaches `bsp` or an allocated frame. If it is currently the bottom frame of the stack, move `bsp` up through the links $\mathbf{L_{next}}$ repeatedly until it reaches `sp` or an allocated frame.

Thread Creation:

Creating a thread is simply to allocate a new frame for the first function to be executed.

Thread Destruction:

Destroying a thread is to free every frame of the thread.

4.4. *k*-Interleaving Stack

We describe the *k*-interleaving stack resource model as follows.

Resources:

k contiguous stack memory spaces ($\mathbf{SM}_i$, $1 \le i \le k$) are reserved at the beginning of a program execution. Each stack (IS_i, $1 \le i \le k$) maintains a stack pointer ($\mathtt{sp}_i$), which is set to the bottom of $\mathbf{SM}_i$. A global register (c) indicates the current stack, set to 1 initially. The structure of each stack and frame are the same as in the pure interleaving stack, except that each frame also maintains an additional variable (d), indicating on which stack it is located.

Frame Allocation:

First, test whether the available memory left in IS_c is large enough for the new frame. If not, change the register c to the next stack ($c = mod(c, k) + 1$). Repeat the process at most *k-1* times until a stack with enough available memory is found. If found, move $\mathtt{sp}_c$ up and update the links ($\mathbf{L}_{\mathbf{prev}}$ and $\mathbf{L}_{\mathbf{next}}$), the flag and the variable d.

Frame Deallocation:

First, mark the frame ‘‘empty’’. Test $\mathtt{sp}_d$ to see whether it is currently the top frame of IS_d. If yes, move $\mathtt{sp}_d$ down through the links $\mathbf{L}_{\mathbf{prev}}$ repeatedly until it reaches a frame marked ‘‘allocated’’.

Thread Creation:

Creating a thread is simply to allocate a new frame for the first function to be executed.

Thread Destruction:

Destroying a thread is to free every frame of the thread.

The *k*-IS is not considered in the simulation study, but is introduced for completeness. This approach is needed for applications that combine a few long-lived threads with mostly short-lived threads.

5. Performance Evaluation

Our study shows how application behavior affects CPU and memory utilization when different thread mechanisms are used. We compare three mechanisms, 1SPT, CIS and HB. The CPU utilization is studied by running a sample program with the MELDC kernel. However, the memory utilization is studied by simulation due to the lack of benchmark concurrent programs.

5.1. CPU Utilization

Table 5-1 shows the CPU performance of the thread mechanisms. The first sample MELDC program (called **Create/Destroy**) starts with a master thread, which repeatedly creates a child thread and waits for the child thread to be terminated one million times. The child threads do nothing but die. The program includes 1,000,000 thread creations and destructions, and 1000 context switches. The second sample MELDC program (called **CS**) does the same as the first program, except it performs 1,000,000 context switches. We ran these programs on a Sun Sparc 2 with four implementations in the MELDC kernel, two for 1SPT (1SPT and 1SPT-emu describe below), one for CIS (CIS-emu) and one for HB.

	HB	1SPT	1SPT-emu	CIS-emu
Create/Destroy	85.48	63.89	67.91	57.21
CS	218.57	96.91	175.70	162.63

Figure 5-1: CPU Utilization (μsec.)

CIS-emu employs an emulated stack architecture instead of the system stack architecture (see Section 6). For fair comparison, we also built an emulated version of 1SPT (1SPT-emu). The additional performance overhead for the emulated version is due to (1) using global variables rather than registers (`sp`, `fp` and `bsp`), and (2) additional data movement between the system stack and the emulated stack. The first source of overhead applies to both thread creation/destruction and context switching. However, data movement is only required for context switching, not thread creation/destruction. The cost for an architecture and/or compiler-supported version of CIS can be calculated by eliminating this overhead. The HB implementation uses `malloc`/`free` for frame allocation and deallocation. The 1SPT and 1SPT-emu implementations use `malloc`/`free` with a free stack list for stack allocation and deallocation. The CIS-emu implementation allocates a large shared stack at program initialization time.

For the **Create/Destroy** program, the major cost is thread creation and destruction, since the cost of context switching is small (1000 versus 1,000,000). The cost of data movement is negligible. HB is the most expensive since it allocates a frame from the heap space every time a thread is created. CIS performs better than 1SPT and 1SPT-emu, since it does not require any memory allocation from heap space except one shared stack. 1SPT and 1SPT-emu need to allocate a stack from the heap space or the free stack list every time a thread is created.

For the **CS** program, context switching plays an important role in CPU performance. This implies that data movement is not negligible. 1STP-emu and CIS-emu perform much worse than 1SPT because of the cost of data movement (around 78μsec). By eliminating this overhead, we expect the cost for a fully supported version of CIS to be around 88μsec.

5.2. Memory Utilization

The simulation is based on the M/M/1//M model [15] (see figure 5-2), where (1) the interarrival time (inter-creation time) distribution for thread creation is exponential (M), (2) the service time (thread life-time) distribution is exponential (M), (3) the number of servers (CPU) is one, and (4) the customer population (maximum potential thread number) is finite (M).

In our simulation, the CPU serves multiple threads in a time-sharing fashion. We assume a unit of CPU time as a cycle. The CPU processes one thread for one cycle before it switches to another thread. Round-robin scheduling is used. A thread with five cycles requires five rounds to finish and each round takes as many cycles as the number of ready threads. For each life cycle, a thread can allocate zero, one

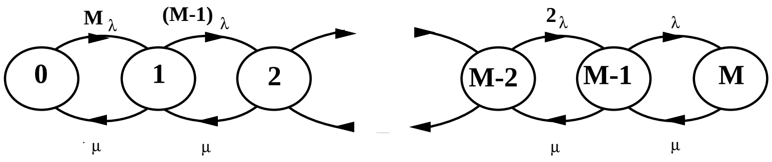

Figure 5-2: State-transition Diagram for M/M/1//M.

or more frames by making function calls, or return from function calls. We assume every frame has the same memory size.

The memory utilization in the simulation reacts to four independent factors: (1) *N*, the number of threads that potentially exist (final customer population), (2) *L*, thread life-time, (3) *I*, inter-creation time (a modified version of interarrival time) between a thread being destroyed and a new thread being created, and (4) *D*, thread depth. The average number of threads is a function of *N*, *L* and *I*. To simplify the simulation, we assume each thread has three phases: a growing phase, in which it allocates frames; an oscillating phase, in which a frame is freed and then allocated for each cycle; and a shrinking phase, in which it deallocates frames. When a thread is created, its life-time and thread depth are determined by two independent random functions, which are exponentially distributed. When it is destroyed, the time until the next thread is created is determined according to another exponential distribution function.

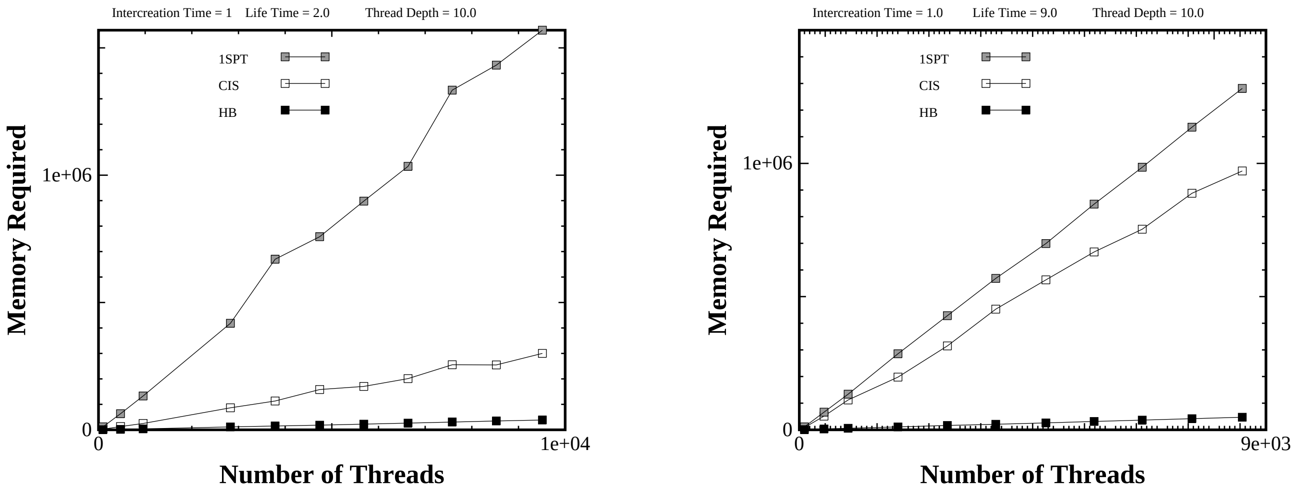

Figure 5-3: Memory Usage for 1SPT, CIS and HB

Figure 5-3 shows the memory usage for 1SPT, CIS and HB when the number of threads increases. The first case (on the left) has an average inter-creation time of 1 cycle, an average thread life-time of 2 cycles and an average thread depth of 10. The second case (on the right) has the same parameters except for an average life-time of 9 cycles. We demonstrate two points (1) when threads are short-lived, CIS performs better than 1SPT with respect to memory utilization; and (2) CIS is scalable.

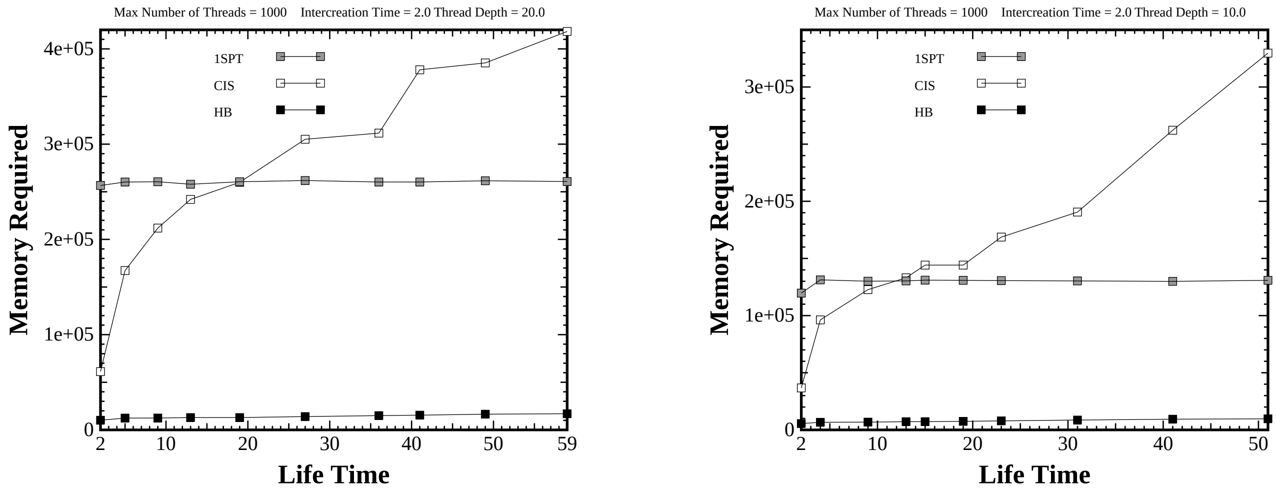

Figure 5-4: Memory Usage for 1SPT, CIS and HB when Thread Life-time Changes

Figure 5-4 shows the memory usage for 1SPT, CIS and HB when the average life-time of threads increases. The first case has 1000 for the maximum number of threads, 2 cycles for average inter-creation time, and 20 for average thread depth. The memory usage of CIS increases while average life-time increases. However, the memory usage of 1SPT remains almost the same. These two curves cross around 20 for life-time. 1SPT is not affected by the life-time of threads, as long as thread depth is fixed; two threads with the same thread depth and different life-time use the same amount of stack memory.

We show two formulae as follows, indicating that 1SPT is affected only by the depth of threads and CIS is affected by the life-time of threads as well as the depth of threads. Assuming that every frame has the same memory size, A_{frame}, the maximum memory that a thread can allocate in one life cycle is A_{cycle}. Let N_{max} be the maximum number of threads that can ever co-exist, D_{max} the maximum thread depth and L_{max} (in cycles) the maximum life time for threads. A_{cycle} is a function of the depth of threads, since increasing the depth of a thread increases the memory that the thread can allocate in one life cycle, assuming its life-time remains the same.

The memory required for 1SPT is as follows:

$$M_{1SPT} = N_{max} \times D_{max} \times A_{frame}$$

The maximum number of stacks is N_{max} and the size of a stack must be at least as large as $D_{max} \times A_{frame}$. Usually, the stack size is expected to be more, since it is pre-determined before the time of thread creation and may be over-estimated for safety reasons. However, the stack size may vary from one thread to another. We eliminate these two factors in our analysis.

The stack memory required for CIS is as follows:

$$M_{CIS} \le N_{max} \times L_{max} \times A_{cycle}$$

The stack memory needed is the maximum allocation window size. Consider a new thread *t* is created and its first frame *f* is allocated on the stack. In $N_{max} \times L_{max}$ cycles, thread *t* and any thread that was created before *t* must be destroyed, and frame *f* and any frame below it must be released for reuse. Within those cycles, the stack can grow at most $N_{max} \times L_{max} \times A_{cycle}$, since at most A_{cycle} can be allocated in one cycle. The maximum window size is bounded by $N_{max} \times L_{max} \times A_{cycle}$.

The second case has the same parameters, except 10 for average thread depth. The two curves cross around 13 for life-time. When decreasing average thread depth, memory usage also decreases for both CIS and 1SPT, since both D_{max} and A_{cycle} drop. Memory usage for 1SPT tends to decrease linearly (see the first formula); in this case, memory usage drops from 260,000 to 130,000, when thread depth drops 50%. However, the drop for CIS is not significant. The reason is that as the depth of a thread decreases with fixed life-time, the growing phase decreases and its oscillating phase increases. This longer oscillating phase magnifies internal fragmentation for CIS. The crossing point moves to the left, while the thread depth decreases.

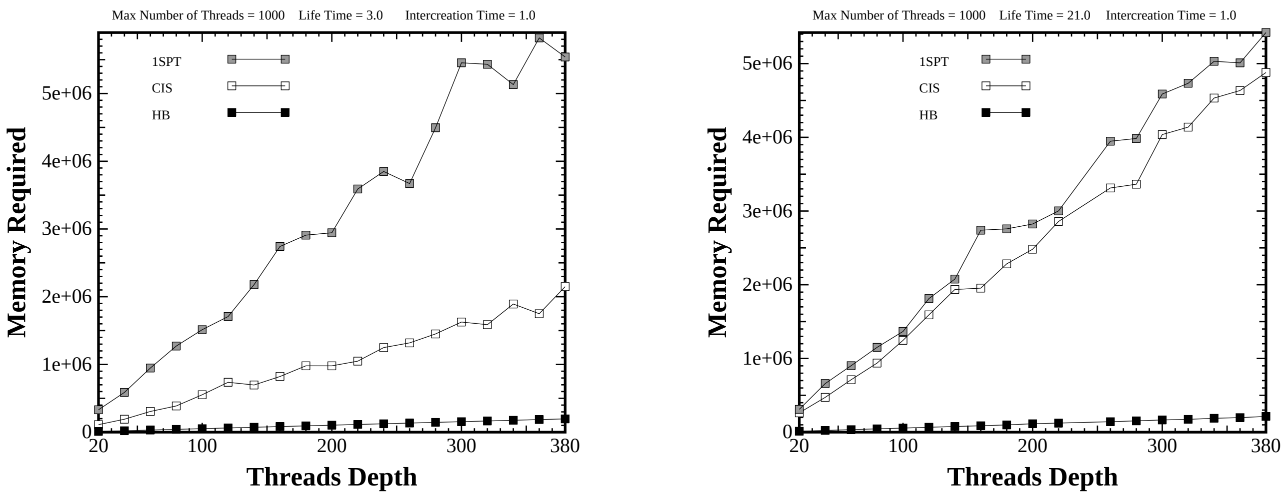

Figure 5-5: Memory Usage for 1SPT, CIS and HB when Thread Depth Changes

Figure 5-5 shows the memory usage for 1SPT, CIS and HB when the average depth of threads increases. The first case has 1000 for the maximum number of threads, an average thread life-time of 3 cycles and an average inter-creation time of 1 cycle. CIS shows better memory utilization than 1SPT, since average life-time is short. The second case has the same parameters except an average thread life-time of 21 cycles. CIS shows almost equal memory utilization to 1SPT, since average life-time is reasonably increased. For both 1SPT and CIS, the memory usage increases while the thread depth increases. This demonstrate that CIS is scalable when the depth of threads increases.

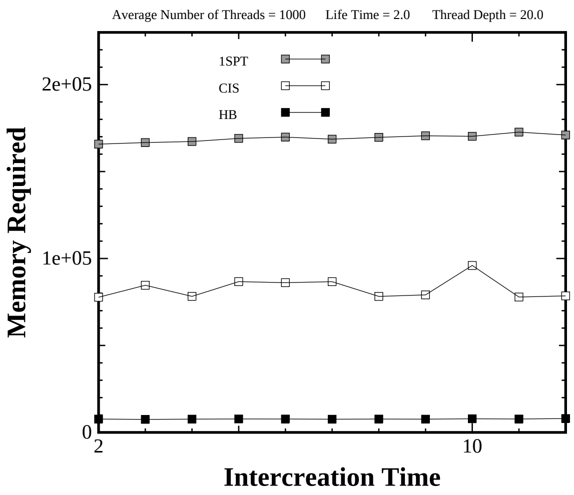

Figure 5-6: Memory Usage for 1SPT, CIS and HB when Inter-Creation Time Changes

Figure 5-6 shows the memory usage for 1SPT, CIS and HB when the average inter-creation time of threads increases. We show the case with an *average* number of threads of 1000, an average thread life-time of 2 cycles and an average thread depth of 20. The reason that we use the average number of threads instead of the maximum number of threads is to eliminate the effect that the memory usage decreases due to the decrease of average number of threads. It shows that inter-creation time does not affect the memory usage for both CIS and 1SPT as long as the average number of threads is fixed.

6. Implementation Status

We have implemented a spectrum of thread mechanisms as part of the MELDC kernel, including 1SPT, HB, IS, CIS and *k*-IS[7]. In the current implementation of the IS family, we do not take advantage of the system stack supported by the hardware architecture. Instead, we build the stack with a contiguous memory space and three global variables (`sp`, `fp` and `bsp`), and emulate the IS behavior. The MELDC compiler translates a MELDC program into a C program that operates on the interleaving stack for function invocation/return as well as thread creation/destruction. This implementation is inefficient with respect to CPU utilization. However, it provides high portability since it requires no modification of the C compiler, and also provides a quick prototype for us to investigate the behavior of interleaving stack[8]. The IS mechanisms using the system stack are required to modify the C compiler because of the ‘‘grey

[7]The desired mechanism is selected as a compiler option, with no changes to code needed.

[8]MELDC runs on both Sparc and Mips architectures.

area’’ as well as frame allocation/deallocation.

We have commenced work on an efficient implementation for CIS by modifying the gcc compiler.[9] One additional register is used as `bsp`. The modified compiler will accept a superset of the C language, C*, which is a multi-threaded C language. In C*, the structure of a frame is visible to programmers (a C struct for each C function); the members of the structure include input parameters, local variables and frame attributes (caller’s link, allocation flag, relocatable status, links to previous and next frames, etc.). The language supports primitives to (1) allocate a frame for a function with input parameters and caller information, (2) run a frame if it is the top frame of a thread, (3) resume a frame if it is the top frame of a thread, (4) yield control (preemptively and non-preemptively) and (5) invoke a garbage collection activity, based on thread mobility. Freeing frames is done implicitly, either when a function returns or a thread is terminated. The performance overhead is limited to maintaining the register and frame attributes.

7. Conclusion

We presented the MELDCTHREADS model to support large-scale dynamic parallelism. Large-scale dynamic parallelism indicates that the number of threads that co-exist is large and the life--time of each thread is short. We introduced the concept of interleaving stack (IS), which melds frames of multiple threads into one shared stack. The pure IS mechanism is not scalable. We designed two useful versions, the circular interleaving stack (CIS) and the *k*-interleaving stack (*k*-IS). The CIS mechanism is evaluated and compared with the traditional one-stack-per-thread (1SPT) mechanism. Our study shows (1) CIS has better memory utilization than 1SPT in the domain of large-scale dynamic parallelism and (2) CIS provides about the same CPU utilization as 1SPT for both thread creation/destruction and context switching. In an application domain where most threads have long life, 1SPT performs better than CIS with respect to memory utilization.

References

[1] Gul Agha, Carl Hewitt, Peter Wegner and Akinori Yonezawa (editors). *ECOOP-OOPSLA Workshop on Object-Based Concurrent Programming*. ACM Press, 1990. Special issue of *OOPS Messenger*, 2(2), April 1991.

[2] Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Computers* 38(12):1631-1643, December, 1989.

[9]Copyright (c) Free Software Foundation, Inc.

[3] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy.
Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism.
ACM Transactions on Computer Systems 10(1):53-79, February, 1992.

[4] *The Task Library, UNIX System-V AT&T C++ Language System Release 2.0*
AT&T, 1989.
Library Manual Select Code 307-145.

[5] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, A. Krakowiak, M. Meysembourg, P. Le Dot, H. Nguyen Van, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville, and G. Vandome.
Architecture and Implementation of Guide, and Object-Oriented distributed system.
Computing Systems 4(1):31-67, 1991.

[6] Brian N. Bershad.
The Presto User's Manual.
Technical Report 88-01-04, Department of Computer Science, University of Washington, January, 1988.

[7] B. Bershad, E. Lazowska, and H. Levy.
Presto: A System for object-oriented parallel programming.
Sofrware: Practice and Experience. 18(8):713-732, August, 1988.

[8] David L. Black.
Scheduling Support for Concurrency and Parallelism in the Mach Operating System.
IEEE Computer 23(5):35-43, May, 1990.

[9] Peter A. Buhr and Richard A. Stroobosscher.
The usystem: Providing Light-Weight Concurrency on Shared-Memory Multiprocessor Computers Running Unix.
Sofrware: Practice and Experience. 20(9):929-963, September, 1990.

[10] Daniel S. Conde, and Felix S. Hsu.
ULTRIX Threads.
In *Proceedings of the Summer 1989 USENIX Conference, Maryland*, pages 257-268. June, 1989.

[11] Eric C. Cooper and Richard P. Draves.
C Threads.
Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie Mellon University, June, 1988.

[12] Daniel G. Bobrow and Ben Wegbreit.
A Model and Stack Implementation of Multiple Environments.
Communications of the ACM 16(10):591-603, 1973.

[13] Thomas W. Doeppner, Jr.
A Threads Tutorial.
Technical Report CS-87-06, Department of Computer Science, Brown University, March, 1987.

[14] John E. Faust and Henry M. Levy.
The Performance of an Object-Oriented Thread Package.
In *Conference on Object-Oriented Programming: Systems, Languages and Applications*. October, 1990.

[15] Leonard Kleinrock.
Queueing Systems, Volume 1: Theory.
A Wiley-Interscience Publication, John Wiley & Sons, New York, 1975.

[16] *Lightweight Processes, Sun Programming Utilities and Libraries*
Sun Microsystems, 1990.
Part Number 800-3847-10.

[17] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc and Evangelos P. Markatos.
First-Class User-Level Threads.
In *Thirteen ACM Symposium on Operating System Principles*, pages 110-121. October, 1991.
Special Issue: Operating Systems Review, Volume 25, number 5.

[18] Richard F. Rashid.
Threads of A New System.
Unix Review :37-49, August, 1986.

[19] Michael L. Scott, Thomas J. LeBlanc, and Brian D. Marsh.
Multi-Model Parallel Programming In Psyche.
In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 70-78. March, 1990.

[20] Shigekazu Inohara, Kazuhiko Kato, Atsunobu Narita and Takashi Masuda.
A Thread Facility Based on User/Kernel Cooperation in the Xero Operating System.
In George J. Knafl (editor), *The Fifteenth Annual International Computer Software & Applications Conference*, pages 398-405. IEEE, September, 1991.

[21] Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite, Jr.
Firefly: A Multiprocessor Workstation.
IEEE Transactions on Computers 37(8):909-920, August, 1988.