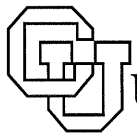
HEAPS O' STACKS: TIME AND SPACE EFFICIENT THREADS WITHOUT OPERATING SYSTEMS SUPPORT

Dirk Grunwald

CU-CS-750-94



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Heaps o' Stacks:
Time and Space Efficient Threads Without Operating System Support

Dirk Grunwald
Department of Computer Science,
Campus Box 430, University of Colorado,
Boulder, CO 80309-0430
(Email:grunwald@cs.colorado.edu)

CU-CS-750-94 November 1994

University of Colorado at Boulder

Technical Report CU-CS-750-94
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Dirk Grunwald
Department of Computer Science,
Campus Box 430, University of Colorado,
Boulder, CO 80309-0430
(Email:grunwald@cs.colorado.edu)

Heaps o' Stacks: Time and Space Efficient Threads Without Operating System Support

Dirk Grunwald
Department of Computer Science,
Campus Box 430, University of Colorado,
Boulder, CO 80309-0430
(Email:grunwald@cs.colorado.edu)

November 1994

Abstract

Modern languages and operating systems often encourage programmers to manage concurrent activity using *threads*, or independent control streams. Although threads can greatly simplify the control structure of complex programs, and are almost essentially for writing parallel programs, threads are not free – in conventional languages, each thread must store a series of *activation records*, usually stored on a stack. Efficiently managing storage for activation records is important for environments with limited storage or limit operating system support. It is also important for our intended purpose - allowing programmers to use threads to simplify the design of scientific parallel programs.

In this paper, we show how whole-program optimization can be used to efficiently create activation records for threads, resulting in safe, efficient threaded programs. Our method reduces TLB misses for programs with many small threads, is time and space efficient for scientific programs, does not require operating system support and is safe. Most importantly, we show how to create very small stacks for parallel scientific programs, allowing hundreds of threads to be used.

1 Introduction

Modern languages and operating systems often encourage programmers to manage concurrent activity using *threads*, or independent control streams. We are primarily interested in supporting ten's to hundreds of threads to mask latency in applications executing on parallel architectures. However, managing memory resources is also important for systems with little storage or limited operating system support, such as small, mobile computers or embedded systems.

Thread libraries typically allocate a private stack segment for each thread. For a small number of threads, this is suitable because a few very large stack segments can be created. Programmers writing programs with many threads or programs intended to run in an environment with limited memory face a more challenging problem – how large should they make each stack segment? Stack segments that are too

small can result in program faults, because most procedure calling conventions assume stacks are unbounded and contiguous, and there is no detection for 'over-running' the small stack segment. On the other hand, large stack segments cause performance problems. Since the stacks are large, they take space, leading to TLB misses and excessive paging.

More importantly, programmers often want to use threads to simplify program design, even in situations where memory space is at a premium. For example, programmers use threads in parallel architectures to mask communication latency [9, 10], and some multi-threaded architectures [2, 1] provide hardware threads for the same purpose. Although multi-threaded architectures provide register-files for tens or hundreds of threads, a large number of threads are often needed to adequately mask the latency in complex applications [8], and the space for activation records must still be managed by the runtime system. In situations that have tens to hundreds of threads, it is untenable to devote kilobytes of memory to each thread.

Considerable work has been done to alleviate some of these problems. The TAM project defines an abstract threaded machine, and there are a number of other such special-purpose runtime systems. However, compilers must explicitly target that runtime system to take advantage of their inexpensive thread models. An alternative model is to provide specialized threads [10], and exploit the semantics of each type of restricted thread.

While these are solutions for some programs, we were interested in a more general solution applicable to a number of "conventional" languages. In particular, we are building a runtime system for a parallel object-orient language [18] that allows programmers to call external C and FORTRAN functions. We can not modify the external compilers, nor do we know what compilers will be used. Similar problems are faced by the designers of runtime systems for parallel data base systems, and distributed computing environments such as OSF/DCE. In each case, a number of threads are needed, the compiler can not be radically changed, a number of compilers may be used, space and time efficiency is important and safety is paramount.

In this paper we show that whole-program or link-time optimization can be used to efficiently create activation records for threads, resulting in safe, efficient threaded programs. Our technique, *combined heap allocation*, works best for programs with well understood control-flow, but is applicable to modern languages using recursion and object-oriented programming techniques. Combined heap allocation can be employed during the initial compilation of the program, but we believe that further profile-based optimization can improve the technique considerably.

In §2.1, we discuss alternative mechanisms to allocation activation record, and then describe combined heap allocation. In §3, we discuss related work, including other systems that use heap-allocated activation records. We measured the cost of combined heap allocation for a limited number of parallel programs and several sequential programs. In §4, we describe the experimental infrastructure we used to determine the efficiency of combined heap allocation, which we present in §5. In §5.1 we describe the performance of the technique for a number of small parallel programs. We conclude in §6 with future work.

2 Heap Allocation For Threaded Applications

There are a number of mechanisms for allocating activation records for threaded applications, using either the operating system, the runtime system, or a compiler-assisted runtime system. Many of these methods have been explored in runtime systems for continuation-passing languages such as Scheme or ML, since the issues are very similar.

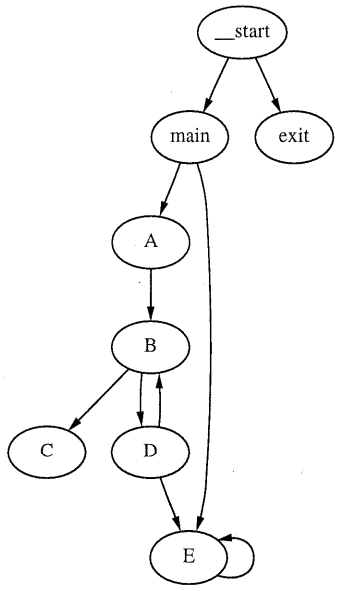
Most operating systems support a stack semantics for some region of memory. Page faults in the stack segment are filled with valid pages when a processor attempts to access those pages. This method can be extended to multiple threads, resulting in "sparse stacks". This method requires operating system support, and the minimum stack size for each thread must be at least one page. Furthermore, unless certain conventions are followed, it may be difficult to distinguish arbitrary memory access violations from stack accesses for procedures with large activation records.

Alternatively, the responsibility for managing activation records can be vested with the runtime library. One common technique is to allocate contiguous memory regions and check that sufficient space remains for the current activation at each procedure call [14]. If the check occurs prior to the procedure, the caller must know the amount of memory needed by the callee; if this information is available, we show how a simpler implementation can be more efficient. The check can also occur in the callee; however, allocating space for the procedure after parameters have been pushed on the stack may require those parameters to be copied, complicating the process. In either case, these explicit checks are performed for every procedure call. For some systems, activation records can be allocated on the heap and recovered via garbage collection [3] or managed by explicit storage allocation.

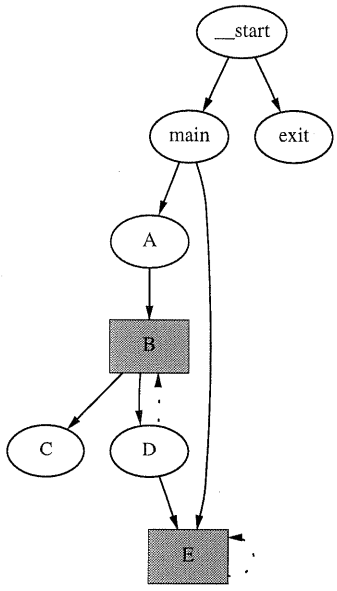
2.1 Combined Heap Allocation

The mechanics of our proposed stack allocation technique are shown in Figure 1(a) through 1(d). We assemble the procedures for a particular program and determine all the 'starting' procedures. These procedures determine the 'roots' of a multigraph representing the program call graph, shown in Figure 1(a). We may have multiple call graphs within a program because we are either unable to determine the exact control flow (due to indirect procedure calls), or because extraneous procedures were aggregated with the program by the linker. In Figure 1(a), there is a single starting procedure, labeled '`__start`'.

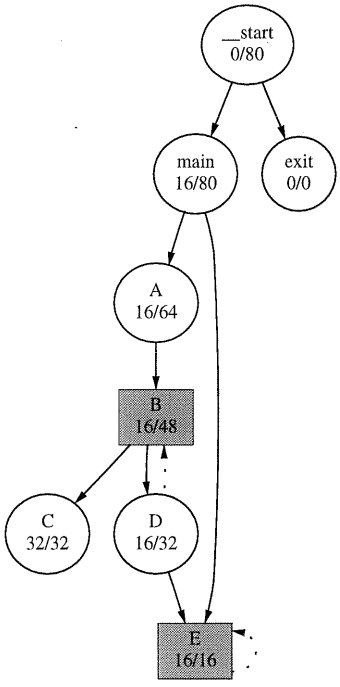
We perform a depth-first search of each independent graph and determine the 'back edges' in the call graph, as shown in Figure 1. Each back edge defines a 'procedural loop,' akin to a conventional loop in a control flow graph. Srivastava and Wall [23] have shown that it is profitable to 'hoist' invariant code from such procedural loops. In this paper, we recognize that procedural loops indicate indirect or direct recursive subroutine calls and 'hoist' stack allocation out of those loops. For example, in Figure 1(b), there are two procedural loops. The first contains $\{B, D\}$ and the second is the self-recursive loop $\{E\}$.



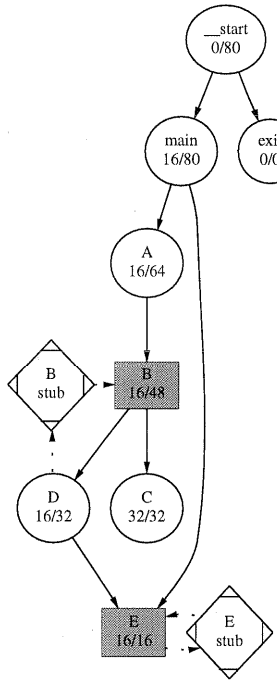
(a) Original Program Call Graph



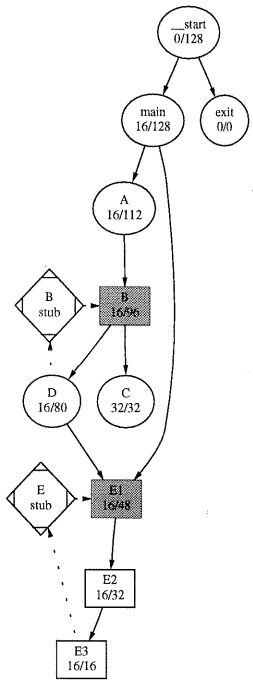
(b) Identifying Back Edges



(c) Computing Storage Space



(d) Inserting Allocation Stubs



(e) Procedure Unrolling

Figure 1: **The Process of Combined Heap Allocation:** Back edges are shown dotted in (b) through (e). Darkened nodes are procedural loop headers. Storage is allocated on entry and each time an "stub" is executed. The values in (c) through (e) are the amount of storage needed by the current procedure and all procedure that can be called without further storage allocation.

If there are no procedural loops in a call graph, the call graph is a directed acyclic graph (DAG) and no recursion can occur in the program. In this case, we can compute the space needed for an arbitrary sequence of procedure invocations in that DAG: each procedure must allocate enough storage for itself and the combined storage needed for each of of the procedures it calls. We use this property to compute the amount of space needed for the graph assuming the back edges are not executed. Later, we compensate for the possibility that the back edges are executed.

In Figure 1(c), each node has been labeled with two numbers. The first indicates the amount of space needed by the current procedure activation and the second the space needed by the current procedure and all procedures that can be called by this procedure, ignoring back edges. For example, procedure 'E' requires 16 bytes of storage. Procedure 'D' also requires 16 bytes of storage, but may also call procedure 'E', and thus must have 32 bytes of storage available. Likewise, procedure 'B' requires 16 bytes of storage, but must allocate 16 + 32 bytes in case 'C' or 'D' is called.

The last step is to compensate for procedure back edges. If we executed the program starting at '`__start`' and never execute the $D \rightarrow B$ or $E \rightarrow E$ edge, we would only need to allocate 80 bytes of storage. This storage could be allocated from the program heap. If the $D \rightarrow B$ edge is executed, than we may need up to 48 additional bytes of storage to accommodate the storage needs of procedure 'C', 'D' and 'E'. Again, as long as the $D \rightarrow B$ or $E \rightarrow E$ edge is not executed again, these 48 bytes of storage should be sufficient to accommodate all procedures callable from 'D'. In our example, procedure calls within the DAG use the conventional stack allocation mechanism of the native runtime system.

As shown by Figure 1(d), we can enforce this additional allocation by inserting 'stubs' on back edges. For example, we create a stub for all procedures calls directed to 'B' along a back edge. This 'stub' allocates 48 bytes of storage and calls procedure 'B'. When that invocation of procedure 'B' returns, the storage is released in the 'B stub'. Storage allocation only occurs when an "allocation stub" is executed. Similarly, we create a 'stub' for the $E \rightarrow E$ back edge that allocates a 16 byte stack segment and then calls procedure 'E'. In fact, there is no need to place allocation stubs only on back-edges. Placing allocation stubs on forward edges reduces the amount of storage needed by callers, at the risk of possibly longer execution time. Likewise, in some situations, space can be sacrificed for faster execution, as shown in Figure 1(e). Here, the $E \rightarrow E$ procedure loop is "un-rolled"; a single storage allocation is used for three invocations of procedure 'E'.

There is one remaining control flow change that we must consider: indirect procedure calls. Our allocation technique is efficient because we model the control flow in the program and allocate sufficient space for the sections we can analyze. In general, we can not precisely model control flow in the presence of indirect procedure calls, although various interprocedural analysis techniques provide very accurate information for some programs [20, 13]. We can accommodate indirect procedure calls by noting where the program loads the address of procedures and substitute the address of a 'stub' function. This correctly

handles programs written in object-oriented languages, but is expensive in the presence of many indirect procedure calls. Fortunately, there are other alternatives and optimizations suitable for most programs.

3 Related Work

Dynamic memory allocation is usually considered to be significantly more expensive than stack allocation. We are aware of very few people that have proposed heap allocation of procedure activation records for threaded programs. As mentioned, the TAM system uses heap-allocated "frames" to store activation records. Among other, Hieb *et al* [14] discussed the problem of heap-allocation activation records for language with continuations. They also described a technique of eliminating stack checks via static analysis of some recursive routines. It is not clear from their paper how frequently these checks could be eliminated.

Appel and Shao [3] concluded that heap allocation of activation records was *more* efficient than stack allocation for languages with closures if a garbage collection system is already being used. They measured the performance of stack and heap allocation for a variety of programs using Standard ML of New Jersey. In their system, programs are compiled using continuation passing style, and all procedures have a fixed-sized 32 byte activation record; the system used by Hieb [14] normally used 30-byte activation records. However, as Hieb pointed out when suggesting that garbage collection might be faster than stack allocation, such a method assumes a large physical memory. Also, the conclusions of [3] may not hold for the programs written for systems where garbage collection is rarely used, procedure activations come in a variety of sizes, and continuation passing style is not used.

Nilsen and Schmidt [19] describe a hardware system for real-time garbage collection. They mention that they initially allocated procedure activation records using this mechanism, but found it to be very expensive because function calls contributed up to 99% of the allocation activity. They then used an efficient linked-list mechanism to pre-allocate a number of different sized activation records, and return unneeded activations to that linked list. Their technique allocates storage on each procedure call. However, we found that individual procedure activation records tend to come in a variety of sizes. Figure 2 shows a histogram of activation records for one execution of the Gnu C compiler. Although a few activation record sizes dominate, there are many distinct sizes. Accommodating this diverse range of activation record sizes on each procedure entry would cause internal or external heap fragmentation. Other researchers, including Hieb [14], have used a technique that allocates a large block of storage and uses explicit checks at each procedure call to insure there is sufficient space. Like the technique of Nilsen *et al*, this requires additional work for each procedure call, but causes less fragmentation. By using a compiler-assisted runtime system that combines information about the structure of the program control flow graph with information about the runtime environment, we can reduce the overhead of storage allocation for traditional imperative programs. For many programs, our technique incurs no additional execution overhead, yet it is simple to implement.

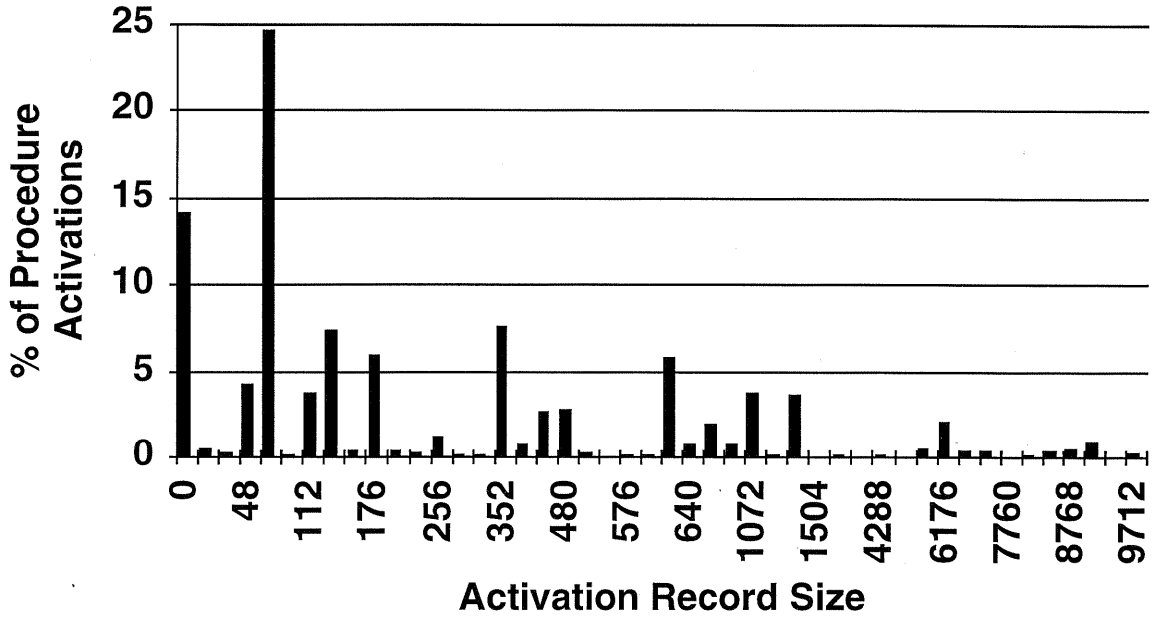


Figure 2: Fraction of activation record sizes for single run of Gnu C compiler. Activation records sizes larger than 10,000 Bytes have been eliminated, as have sizes contributing less than 0.01% of the total activation record.

Our technique uses whole-program or link-time optimizations. Whole program compilers are common for functional languages, but less common for imperative languages. Wall and Powell describe the Mahler [25] system, which allowed link-time optimizations. Benitez and Davidson [4] describe VPO, the *very portable optimizer*, which was capable of some link-time optimizations. Santhanam and Odnert [21] described an inter-procedural register allocation algorithm constructed using the compilers on the HP PA-RISC. More recently, Srivastiva and Wall [23] describe OM, a system for intermodule code optimization. The program analysis system we use, ATOM, is based on this technique. Link-time optimizations has drawbacks, because higher-level information such as variable type information or possible indirect call targets for languages such as C++ are not represented in the final program. The compiler for the Tera [2, 15] computer system also performs whole-program optimization using a "program database". We feel that whole-program or link-time optimization will become more prevalent, because it will become essential when compiling programs for modern wide-issue architectures.

4 Experimental Design

We wanted to determine the execution penalty for combined heap allocation. We used two experiments to evaluate the method. In the first, we modified three parallel programs to allocate activation records using combined allocation. This experiment demonstrates the effect on cache and TLB miss rates. However, these

programs, although drawn from actual applications, use a limited programming style. How well would combined allocation perform for different programs? To determine this, we instrumented and traced the execution of a number of programs and estimated the additional cost from our activation record allocation mechanism. We did not actually transform these programs.

The parallel programs were the `mp3d` fluid-flow program from the SPLASH-I benchmark suite, the `ocean` multi-grid program from the SPLASH-II suite and a simple Red-Black SOR kernel from a turbulence calculation in an associated NSF "Grand Challenge" program. We also instrumented the programs from the SPEC92 benchmark suite, the Perfect club benchmark suite and other programs, including object-oriented programs written in C++. We used ATOM [22] to instrument the programs. The programs were compiled on a DEC 3000-400 and either the DEC C compiler, DEC FORTRAN compiler or DEC C++ compiler. The systems were running the standard OSF/1 V2.0 operating systems, and all programs were compiled with standard optimization (`-O`). For the SPEC92 programs, we used the largest input distributed with the SPEC92 suite. For the Perfect Club programs, we used the standard input. The alternate programs include: `cfront`, version 3.0.1 of the AT&T C++ language preprocessor written in C++, `groff`, a version of the `ditroff` text formatter written in C++, `idl`, a C++ parser for the CORBA interface description language and TEX, a text formatting system. We selected these programs because we found that the SPEC92 suite did not typify the behavior seen in large programs or C++ programs [7]. Since languages such as C++ introduce new problems in our stack allocation technique, we felt it was important to consider these alternate programs. For these alternate programs, we used sizable inputs we hoped would exercise a large part of the program – for example, the TEX program formatted a 45-page document.

Table 1 shows the basic statistics for the programs we instrumented. The first columns lists the number of instructions traced; this is not shown for the parallel programs, because we ran those programs multiple times, and present more information later. The remaining columns indicate *static* characteristics of the program, or attributes concerning the program body rather than program execution. For example, the program `doduc` contained 707 procedures, and 2,897 call instructions. We classified 28 of those calls as 'back edges', or potential recursive edges, and 2,781 edges as standard 'forward edges'. There were also 180 indirect function calls, where we could not determine the program flow during static analysis. It may seem odd that FORTRAN programs would contain so many, indeed, would contain *any*, indirect procedure calls. However, the DEC FORTRAN runtime system calls subroutines written in C, and this introduces most of the indirect function calls. Likewise, back edges in the FORTRAN programs, indicating potential recursion, primarily arise from the FORTRAN runtime system.

5 Performance Comparison

We first describe the performance for the parallel programs, and then the sequential programs. In each case, we are concerned with different metrics. We use the parallel programs to demonstrate that combined

		Occurances in Program Text				
Program	Instructions Traced	Procedures	Call Sites	Back Edges	Foreward Edges	Indirect Calls
RB	–	190	556	6	503	47
mp3d	–	287	1,101	6	996	99
ocean	–	249	821	6	766	49
doduc	1,149,864,756	707	2,897	28	2,781	88
fpppp	4,333,190,877	684	2,729	29	2,612	88
hydro2d	5,682,546,752	715	2,946	29	2,829	88
nasa7	6,128,388,651	705	2,874	29	2,757	88
ora	6,036,097,925	667	2,630	19	2,523	88
spice	16,148,172,565	814	5,046	28	4,930	88
swm256	11,037,397,884	676	2,666	29	2,549	88
tomcatv	899,655,317	616	2,480	18	2,374	88
APS	1,490,454,770	796	3,855	28	3,739	88
CSS	379,319,722	817	5,075	27	4,960	88
LWS	14,183,394,882	713	3,135	28	3,019	88
NAS	3,603,798,937	739	3,448	28	3,332	88
OCS	5,187,329,629	716	3,236	27	3,121	88
SDS	1,108,675,255	767	3,257	28	3,141	88
TFS	1,694,450,064	714	3,173	27	3,058	88
TIS	1,722,430,820	680	2,755	17	2,650	88
WSS	5,422,412,141	756	3,263	17	3,158	88
alvinn	5,240,969,586	211	569	3	558	8
compress	92,629,658	148	468	3	457	8
ear	17,005,801,014	289	967	4	955	8
eqntott	1,810,540,418	211	758	41	698	19
espresso	513,008,174	550	3,213	48	3,142	23
gcc	143,737,915	1,650	8,747	621	8,051	75
li	1,355,059,387	550	1,757	48	1,697	12
sc	1,450,134,411	511	2,704	118	2,573	13
cfront	19,001,390	943	10,191	849	9,271	71
idl	21,138,201	1,458	4,761	6	3,383	1,372
tex	147,827,875	831	5,252	92	5,076	84
groff	41,522,284	1,710	6,179	113	5,407	659

Table 1: General Information Concerning Traced Programs

				64KB Stack Size		Small Stack Size	
Program	Threads	Context Switches	Instructions Per Switch	Load Miss Rate	TLB Misses and Rate	Load Miss Rate	TLB Misses and Rate
RB	4	8004	8,980,764	2.92%	41 (0.00%)	2.92%	28 (0.00%)
RB	64	128,064	582,163	3.89%	162,709 (0.06%)	3.97%	25,323 (0.01%)
RB	128	2,049,024	51,854	9.75%	2,178,450 (0.47%)	10.69%	380,695 (0.08%)
MP3D	4	18,320	347,958	10.84%	14,840 (0.06%)	9.43%	4,874 (0.02%)
MP3D	64	230,228	55,187	9.75%	209,423 (0.35%)	10.13%	16,783 (0.03%)
MP3D	128	454,302	42,710	10.94%	437,453 (0.48%)	11.18%	67,335 (0.07%)
OCEAN	4	4,329	18,536,623	12.87%	2,980,601 (1.55%)	12.87%	2,979,534 (1.55%)
OCEAN	64	69,045	1,205,854	13.02%	449,258 (0.22%)	13.02%	415,494 (0.21%)
OCEAN	128	138,042	620,358	13.18%	409,637 (0.19%)	13.16%	340,244 (0.16%)

Table 2: Information Collected From Parallel Program Execution

allocation is practical, and that it can reduce TLB misses for some parallel numeric programs. We argue that the reduced TLB misses offset the additional instructions (if any) needed to allocate storage using combined allocation. Thus, it is possible to have efficient threads while preserving safety by combined allocation.

5.1 Parallel Programs

We modified three parallel programs to use a simple light-weight thread library based on the Quick-Threads [16] library. We modified the programs assuming we were using a compiler assisted runtime system for parallel programs and only applied combined allocation to the code executed by threads. We used ATOM to construct the call graph for each program and determine the amount of storage needed for each procedure. We then manually changed the stack allocations for the individual threads and ATOM to build a simulator modeling the first cache level and TLB of the Alpha AXP 21064. The stack modifications to the programs were very simple, and in practice, we would use a tool such as OM [23] to transform the binaries. However, we can not currently modify programs using OM and then trace those programs using ATOM. The need to trace the final applications introduces another complication into our study: the ATOM analysis routines also require stack space to store activation records specific to the tracing task. When modifying the programs, we had to leave additional room for the ATOM analysis routines.[1]

As with many scientific programs, each of the programs we examined contained no recursion. Only the initial stack segment allocated for a thread needed to be modified. For these application, we are concerned with three ways that combined allocation can affect the program and its performance: total storage needed for threads, any increase or decrease in TLB misses and any increase or decrease in cache miss rates.

[1]We hope to avoid this in the future by modifying ATOM. Furthermore, we hope to examine more parallel programs in the future. However, many programs in the SPLASH benchmark suite are 32-bit specific, and require pre-emptive threads and strongly-consistent memory because of implicit synchronization. Although I modified those applications and ported them to the Alpha, they still suffer from floating point exceptions and unaligned access violations.

We ran the programs with a "Large" stack size of 64KBytes, and a "Small" stack size particular to each application. When executing the instrumented program, we also added sufficient storage for the ATOM analysis routines; this storage was computed by disassembling and analyzing the instrumented programs. Each thread in the "Red-Black" and "mp3d" programs required 384 bytes, or 1200 bytes when instrumented. "Ocean" required 2256 bytes, or 3072 bytes when instrumented. In each program, an error routine requiring $\approx 12,000$ bytes of storage was called; we modified the forward call to that routine to allocate a new stack segment, but that routine was never called. The RB and MP3D applications used small datasets ($\approx 256KBytes$ each) and the OCEAN model used a larger dataset ($\approx 4MB$).

Table 2 shows the information collected from the parallel programs, using four, 64 and 128 threads. The table shows the number of context switches and the number of instructions executed per context switch using the large threads. We recorded the load miss rate and TLB misses and miss rate for both the large and small stack sizes. We modeled a memory system similar to that of the Alpha AXP 21064 processor, including a 8KB direct-mapped, no-write-allocate cache with 32 bytes cache lines and a 32-entry fully associative TLB for 8KByte pages dedicated to data references. We only report the load miss rate because the cache did not allocate cache lines on stores to addresses not already in the cache; missing stores assemble in a write buffer and would not greatly delay the processor.

Uniformly, threads using smaller stack segments have a slightly higher load miss rate and a slightly lower TLB miss rate. The applications were very sensitive to memory organization for reasons we have not determined, with three cache lines accumulating $\approx 25\%$ of all misses. The lower TLB miss rate is a direct outcome of the smaller stack segments. For threads larger than a virtual memory page, each thread stack requires a full TLB entry. When using 1200 bytes for a stack segment, six threads can share the same TLB entry, indicating the reason for the six-fold decrease in the TLB miss rate for "mp3d" and "RB". The TLB miss rate for the "Ocean" program decreases because the threaded implementations change the access order to the main data structure, effectively blocking the computation [17].

Although TLB misses are relatively expensive on most architectures and reducing those misses is important, the most important point to conclude from the examination of the parallel programs is how little space is needed for threads in these applications. We believe that runtime systems for parallel, scientific computation can benefit from a large number, possibly thousands, of threads per processor. This example demonstrates that for some applications, the memory needed by for these threads is fairly small, and can managed by a simple compiler-assisted runtime system.[2]

[2]Ideally, I would have shown detailed timing to contrast the change in cache and TLB miss rates; however, all of our Alpha workstations were being used to collect information for another paper, and it was hard to collect consistent results.

Program	Foreward Calls	Backward Calls	Indirect Calls	Non-Zero Allocs	Zero Allocs
doduc	6,729,093 (99%)	0 (0%)	7 (0%)	1	7
fpppp	3,254,384 (99%)	0 (0%)	7 (0%)	1	7
hydro2d	4,944,298 (99%)	1,603 (0%)	9 (0%)	1	1,612
nasa7	11,240,400 (99%)	0 (0%)	7 (0%)	1	7
ora	44,232,393 (99%)	0 (0%)	7 (0%)	1	7
spice	46,172,975 (99%)	0 (0%)	7 (0%)	1	7
swm256	138,265 (99%)	1,200 (0%)	7 (0%)	1	1,207
tomcatv	10,206 (99%)	0 (0%)	4 (0%)	1	4
APS	3,242,555 (99%)	2,880 (0%)	29 (0%)	1	2,909
CSS	1,873,265 (99%)	0 (0%)	7 (0%)	1	7
LWS	98,720,298 (99%)	10 (0%)	10 (0%)	1	20
NAS	21,349,424 (99%)	0 (0%)	29 (0%)	1	29
OCS	702,861 (99%)	0 (0%)	4 (0%)	1	4
SDS	284,360 (99%)	2,018 (0%)	13 (0%)	17	2,015
TFS	1,414,912 (99%)	0 (0%)	9 (0%)	1	9
TIS	1,396 (99%)	0 (0%)	4 (0%)	1	4
WSS	6,228,623 (99%)	0 (0%)	22 (0%)	1	22
alvinn	3,065,741 (99%)	0 (0%)	5 (0%)	1	5
compress	251,419 (99%)	0 (0%)	3 (0%)	1	3
ear	240,592,630 (99%)	127 (0%)	4 (0%)	128	4
eqntott	1,457,027 (31%)	8,436 (0%)	3,215,050 (68%)	8,620	3,214,867
espresso	1,982,372 (94%)	27,509 (1%)	84,753 (4%)	27,510	84,753
gcc	1,133,555 (76%)	275,925 (18%)	80,811 (5%)	306,342	50,395
li	27,512,772 (86%)	3,427,531 (10%)	919,968 (2%)	4,347,497	3
sc	13,996,239 (87%)	1,920,865 (12%)	39,969 (0%)	1,960,780	55
cfront	266,849 (94%)	15,984 (5%)	85 (0%)	16,067	3
idl	376,200 (43%)	0 (0%)	497,087 (56%)	205,689	291,399
tex	817,104 (95%)	36,063 (4%)	24 (0%)	36,064	24
groff	589,627 (70%)	40,110 (4%)	207,082 (24%)	130,552	116,641

Table 3: Information from Trace Driven Simulation

5.2 Sequential Programs

The parallel programs demonstrate that combined allocation can reduce TLB misses and memory requirements for some programs. However, we could only examine a limited number of parallel programs because such programs are less common and frequently not portable. To measure the performance of combined allocation across a broader spectrum of programs, we instrumented and measured several sequential programs, reasoning that most parallel programs will be no more complex than common sequential programs.

Table 3 shows the information we collected using trace-driven simulation of our sequential program suite. We recorded the number of procedure calls executed along 'forward' edges - these calls use the stack semantics to allocate activation records. We also recorded the number of 'backward' (i.e., potentially recursive procedure calls) and 'indirect' procedure calls. Not all of these procedure calls requires heap-based

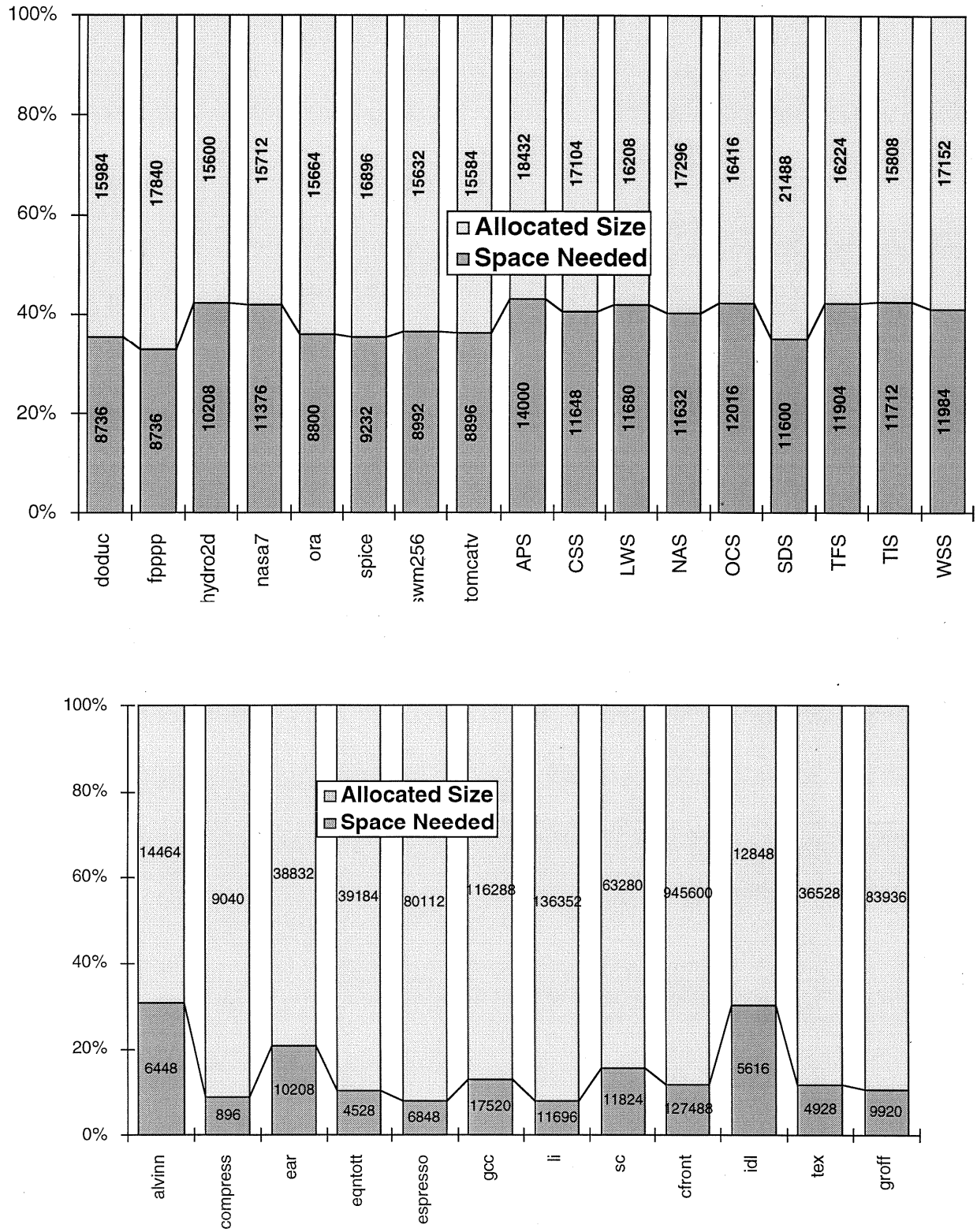


Figure 3: Normalized Execution Time for FORTRAN, C and C++ Programs.

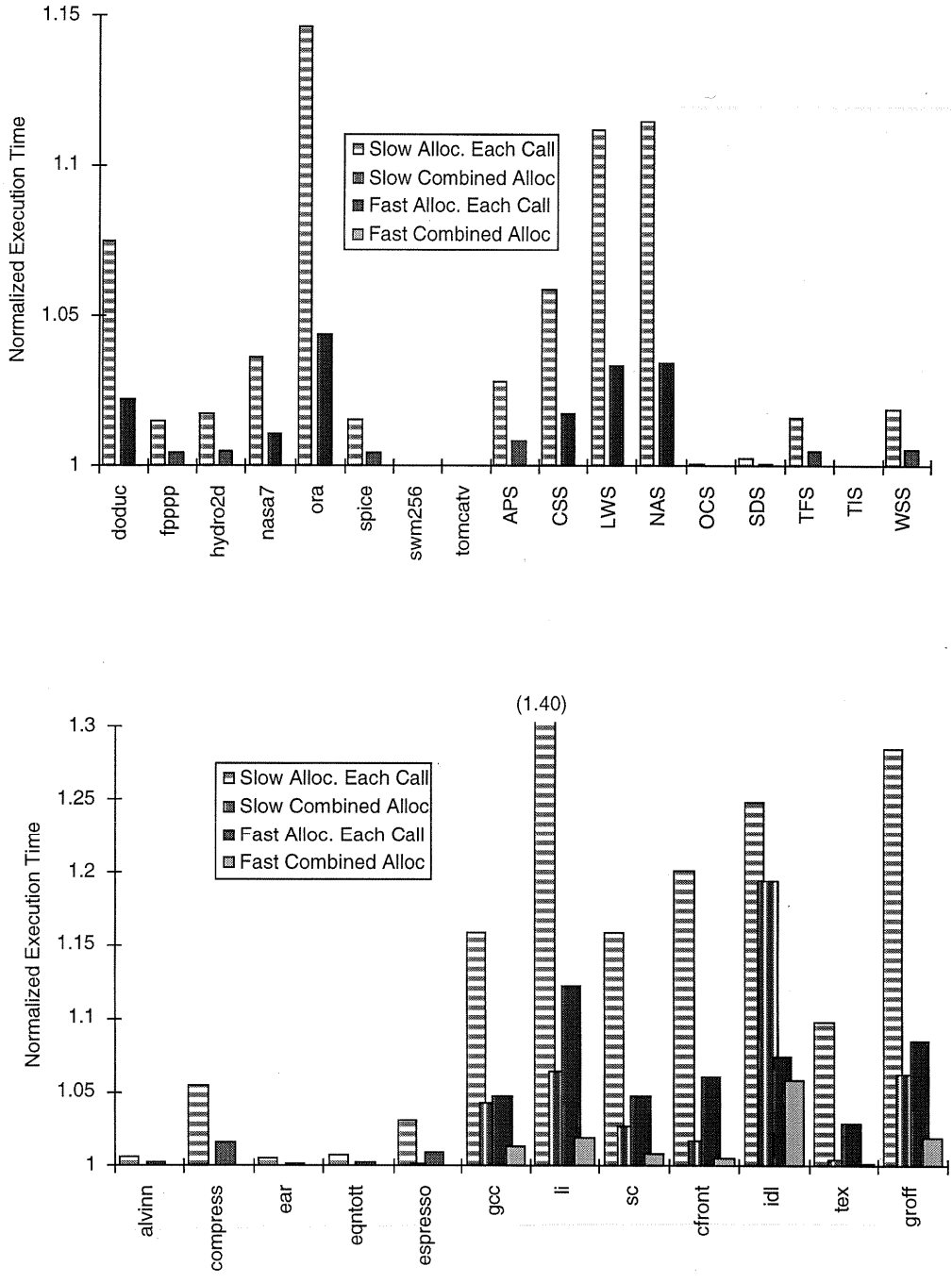


Figure 4: Normalized Execution Time for FORTRAN, C and C++ Programs. Most values for Combined Allocation are very close to zero, and are not visible on this graph.

allocation. Many procedure calls are to procedures that do not allocate an activation record. In these cases, shown in the column labeled 'Zero Allocs', no storage is needed. The column labeled 'Non-Zero Allocs' shows the number of actual heap-based allocations needed during the program execution. Note that most of the scientific programs use a single heap-based allocation at the beginning of the program execution. By comparison, programs with highly recursive control flow, such at `li`, use considerable heap-based allocation. Note that althrough `eqntott` makes many indirect calls, the callee allocates no storage.

Figure 3 shows the total amount of space allocated by the programs. For each program, the bottom portion of the bar represents the fraction of memory allocated that was actually needed. The allocated and needed values are overlayed on the bars. In general, we allocate about twice as much storage as for the scientific programs, and four to five times as much storage for the C and C++ programs. Why? The conventional calling mechanism only allocates space for what is actually needed – we allocate space for what *may* be needed. For example, under OSF/1, the `abort` routine in the C library requires 64 bytes of space. Although this routine is not executed in the normal course of execution, our technique allocates space in case it is called. It may be possible to reduce the amount of space needed during typical executions using profiles or execution estimates [24, 6].

It's interesting to note how little space is required by most of the programs. Thread libraries that allocate large stack segments (e.g., 64Kbytes) would be wasting considerable space if these programs are representative of parallel programs; in fact, we found that parallel programs allocated considerably less space. However, `cfron` shows that using such fixed sized stack allocations are too small for some programs.

Figure 4 shows the normalized instruction count for each program using two variants of heap based activation record allocation. The normalized instruction count is '1.0' for each program. Values higher than '1.0' indicate that more instructions were executed – for example, the value '1.05' indicates that 5% more instructions were executed. Without a more detailed architectural simulation, we can only state the execution cost using instruction counts – the actual execution time may be greater or smaller.

We considered two activation allocation mechanisms and two different costs for the memory allocation routines. Combined allocation is not used in the columns labeled 'Alloc Each Call,' resulting in many heap allocations. This could be implemented in existing compilers that do not support whole-program or link-time optimization, and provides an indication of the value of applying whole-program analysis to this problem. The columns labeled 'Combined Allocation' list the normalized instruction count using our proposed mechanism. The values for combined allocation for all the FORTRAN programs, `alvinn`, `compress`, `ear`, `eqntott`, `espresso` and `tex` are present, but very close to zero.

Since the naive allocation strategy makes considerable use of memory allocation, increased memory allocation costs dramatically affect that method. From previous experience with memory allocation routines [11, 12], we felt that we could allocate and release memory in 22-cycles or fewer on almost any architecture and operating system, given the information available to the compiler. Thus, the 'Slow

Alloc' column represents our estimate of the 'worst case' performance we would expect for a uniprocessor, or 'good case' performance for a parallel processor, where additional locking is needed. With a good deal more work, we believe that we could allocate and release most activation records in eight cycles, using the technique described below. We feel that it's unlikely that we could reduce this allocation time. Thus, the 'Fast Alloc' columns reflect our estimate of the 'best case' performance for each technique. We assumed that the conventional allocation mechanism typically uses two machine instructions - a register-relative add instruction to increment or decrement the stack. For larger stack segments, more complicated instruction sequences are needed, but this rarely occurs in practice. In CUSTOMALLOC, we optimized heap allocations using distinct linked-lists for each allocation size. This process is suitable for combined heap allocation, since the compiler knows the size of each activation record. This technique was also used by Nilsen and Schmidt [19], but they performed the allocation on every procedure call, and has to address a many allocation record sizes. By combining the allocations, we require fewer allocation sizes.

What do our performance estimates tell us? First, if we assume that heap-based memory allocation isn't that expensive (e.g., between 22 and 8 cycles), combined allocation offers significant advantages over the naive allocation. We believe heap-based allocation must be efficient for *all* programs for it to be widely adopted. Happily, our combined allocation mechanism reduces the space and time needed, making heap-based allocation a negligible part of the execution cost.

There are opportunities to improve upon our technique. Object oriented programs tend to use a large number of indirect function calls. In our technique, we assume that we can not improve any of these indirect function calls. In a related paper, we measured the performance and behavior of a number of C++ programs [5, 7]. Many programs have a few, limited number of possible call targets for a given indirect function call. For example, in the `idl` program, simple whole-program analysis can eliminate almost 100% of the indirect function calls - this would then evince the program flow and reduce the cost of heap based allocation. Furthermore, even in cases where we can not eliminate indirect procedure calls, we may still be able to use heap-based allocation. For example, assume that program analysis indicates that an indirect function call may call four different procedures. We can simply allocate storage to accommodate the largest of those procedures – this may waste some space, but would improve program execution time.

6 Conclusions and Future Work

We have shown that heap-based allocation of procedure activation records is a tenable option for threaded programs. Using our technique, heap based allocation is usually as fast as conventional stack allocation, increases the safety of programs using explicit threads, may use considerably less memory than heuristics in existing thread libraries and can reduce TLB misses and paging in some applications.

For parallel programs that do require a great deal of memory, such as "RB" and "mp3d", the smaller stack segments reduce TLB misses, improving performance. For programs that require more memory,

such as "Ocean", smaller stacks permit more threads that can be used to mask communication latency. For all programs, regardless of memory needs, combined allocation provides efficient stack allocation that safely accommodates recursion. Whole program analysis can also be used to reduce the execution overhead of threaded applications using inter-procedural live-register analysis to reduce the cost of context switch routines. We are implementing both combined allocation and context switch reduction using a link-time optimization package.

Future Work: Table 3 showed that combined heap-based allocation occasionally consumed considerably more memory than the conventional allocation method. It is also useful for activation records to use a few, distinct sizes to reduce the cost of memory allocation. We are using measured and estimated [6] program profile information to identify infrequently called sections of the program, to insert additional 'allocation stubs' at those points. This reduces the amount of memory needed during normal execution. We are also exploring the time savings of "procedure loop unrolling."

Acknowledgements

I would like to thank Alan Eustace and Amitabh Srivastava for developing ATOM, and David Wall and Amitabh Sirvastava for developing OM. Rich Neves and David Keppel provided comments on an earlier version of this paper. This work was funded in part by NSF grant No. ASC-9217394, ARPA contract ARMY DABT63-94-C-0029 and assistance from Digital Equipment Corp.

References

[1] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–115. IEEE, 1990.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *International Conference on Supercomputing*, pages 1–6, June 1990.

[3] Andrew W. Appel and Zhong Shao. An empirical and analytic study of stack vs. heap cost for languages with closures. Technical Report CS-TR-450-94, Princeton University, March 1994.

[4] M.E. Benitez and Jack W. Davidson. A portable global optimizer and linker. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 329–338, Atlanta, GA, June 1988.

[5] Brad Calder and Dirk Grunwald. Call prediction in object-oriented languages. In *1994 ACM Symposium on Principles of Programming Languages*, January 1994.

[6] Brad Calder, Dirk Grunwald, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. Corpus-based static branch prediction. Technical Report CU-CS-XXX-94, University of Colorado, November 1994. (In Preperation).

[7] Brad Calder, Dirk Grunwald, and Benjamin Zorn. Quantifying behavioral differences between C and C++ programs. Technical Report CU-CS-698, Univ. of Colorado-Boulder, January 1994.

[8] D. Callahan and B. Smith. A future-based parallel language for a general-purpose highly-parallel computer. In A. Nicolau D. Gelernter and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 95–113. The MIT Press, Cambridge, Massachusetts, 1990.

[9] D.E. Culler, A. Sah, K.E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: a compiler-controlled threaded abstract machine. In *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, Santa Clara, CA, April 1991.

[10] Dawson R. Engler, Gregory R. Andrews, and David K. Lowenthal. Efficient support for fine-grain parallelism. TR 93-13, Univ. Arizona, April 1993.

[11] Dirk Grunwald and Benjamin Zorn. CUSTOMALLOC: Efficient synthesized memory allocators. *Software—Practice and Experience*, 23(8):851–869, August 1993.

[12] Dirk Grunwald, Benjamin Zorn, and Rob Henderson. Improving the cache locality of memory allocation. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*. ACM, June 1993.

[13] Mary W. Hall and Ken Kennedy. Efficient call graph analysis. *Letters on Programming Languages and Systems*, 1(3):227–242, September 1992.

[14] Robert Heib, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presenceof first-class continuations. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 66–77, June 1990.

[15] Robert Henry, Allan Porterfield, and Burton Smith. Tera compiler overview. (discussion), January 1994.

[16] David Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE 93-05-06, University of Washington Department of Computer Science and Engineering, May 1993.

[17] Monica Lam, Edward Rothberg, and Michael Wolf. The cache performance and optimization of blocked algorithms. In *Proc. of the Fourth Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, 1991.

[18] Jenq Kuen Lee and Dennis Gannon. Object-oriented parallel programming experiments and results. In *Proceedings SuperComputing '91*, pages 273–282, November 1991.

[19] Kelvin D. Nilsen and William J. Schmidt. A high-performance hardware assisted real-time garbage collection system. In *Proceedings of the 5th Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct 1994.

[20] Barbera G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5(3):216–226, May 1979.

[21] Vatsa Santhanam and Daryl Odnert. Register allocation across procedure and module boundaries. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 28–39, June 1990.

[22] Amitabh Srivastava and Alan Eustace. Atom: A system for building customized program analysis tools. TN 41, DEC-WRL, January 1994.

[23] Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimizations at link-time. *Journal of Programming Languages*, March 1992. (Also available as DEC-WRL TR-92-6).

[24] Tim A. Wagner, Vance Maverick, Susan Graham, and Michael Harrison. Accurate static estimators for program optimization. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 85–96, Orlando, Florida, June 1994. ACM.

[25] David W. Wall and Michael L. Powell. The mahler experience: Using an intermediate language as the machine description. In *Proceedings of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 100–104. ACM, October 1987.