Fast Object-Oriented Procedure Calls: Lessons from the Intel 432

Edward F. Gehringer
Department of Electrical and Computer Engineering
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-7911

Robert P. Colwell
Multiflow Computer, Inc.
175 No. Main St.
Branford, CT 06405

Abstract

As modular programming grows in importance, the efficiency of procedure calls assumes an ever more critical role in system performance. Meanwhile, software designers are becoming more aware of the benefits of object-oriented programming in structuring large software systems. But object-oriented programming requires a good deal of support, which can best be distributed between the compiler and architectural levels. A major part of this support relates to the execution of procedure calls. Must such support exact an unacceptable performance penalty? By considering the case of the Intel 432, a prominent object-oriented architecture, we argue that it need not. The 432 provided all the facilities needed to support object orientation. Though its procedure call was slow, the reasons were only tenuously related to object orientation. Most of the inefficiency could be removed in future designs by the adoption of a few new mechanisms: stack-based allocation of contexts, a memory-clearing coprocessor, and the use of multiple register sets to hold addressing information. These proposals offer the prospect of an object-oriented procedure call that can, on average, be performed nearly as fast as an ordinary unprotected procedure call.

1. Introduction

There are currently two general points of agreement about object-oriented programming systems: they represent a desirable programming environment [1]-[3]; and their existing implementations are too slow [4]-[7]. Procedure call/return linkage, a major overhead even in conventional systems, can easily dominate the performance of object-based systems. This paper will discuss some problems of object-based procedure-linkage mechanisms, offering strategies for combatting the overheads, and quantifying the effects of such strategies.

Object-oriented systems come in two major "styles". Smalltalk [3] is typical of one, and the Intel 432 [5] or IBM System/38 [8] could represent the other. The object-orientation of the 432 is manifested in several ways:

- All information is encapsulated into protected sets called *objects* (instruction segments, data structures, processes, messages).
- Every memory reference, whether an instruction fetch or operand access, is checked for read/write privilege and base/bounds validity.
- Pointers to objects are protected. These pointers are not directly manipulated by the user program, but only by trusted hardware and microcode on behalf of the user.

This style of object orientation can be viewed as a conventional system with runtime checking built into the addressing mechanism.

The Smalltalk style of object orientation is built on a different paradigm, but presents the same challenges to the implementor. In Smalltalk, computation proceeds via messages passed from one object to another, requesting that various operations, called *methods*, be performed by the receiver object on the receiver's data structures. The methods themselves can be altered, added, or deleted at runtime.

Procedure calls are among the most frequently executed instructions, accounting for 12% [9] to 25% [10] of executable statements. In conventional systems, call/return overhead is high enough to merit special attention at the architecture and compiler levels [9], [11]. The SOAR (Smalltalk On A RISC) project [12] devotes hardware resources to the call-linkage mechanism. The 432 incorporates an elaborate protected-call mechanism that provided separate contexts for each procedure, but runs so slowly that the overall performance is severely impacted.

The Intel 432 is an excellent vehicle for measuring the impact of object orientation on procedure-call efficiency. It performs all of the operations necessary to support object orientation. Through the courtesy of the Intel Corporation, several 432 programs have been traced at the register-transfer level [13]. All of the steps in a 432 procedure call have been accounted for. By analyzing which are needed in any object-oriented architecture, and which are artifacts of the 432's implementation, we can estimate how much overhead is inherent in an object-oriented procedure call.

This research has been supported in part by the U.S. Army Center for Tactical Computer Systems under contract number DAAB 07-82-C-J164. The cooperation of the Intel Corporation is also gratefully acknowledged.

0884-7495/86/0000/0092$01.00 © 1986 IEEE

2. Division of Labor: Compiler vs. Architecture

The implementation of an object-oriented system is a matter of teamwork between the compiler and the architecture. Determining what architectural complexity needs to be associated with object orientation is primarily a question of deciding what the architecture can do more effectively than the compiler. We propose a division of labor between the compiler and the architecture: the compiler is responsible for the correct functioning of each separately compiled module, while inter-module interfaces are entrusted to the architecture. Let us explore the reasons for this decision.

If a program is to run with high efficiency, it must avoid re-doing unnecessary work. Decisions that can be made once and for all at compile time should not be deferred until runtime, when they might have to be remade repeatedly, each time that a piece of code is executed. One example is bounds checking of a constant subscript, or a subscript whose range can be determined by flow analysis. On most architectures, compile-time bounds checking saves substantial execution time. Another example is pipeline optimization in the MIPS machine: the compiler is charged with scheduling the initiation of pipeline operations [14] to avoid collisions. Note that in both of these examples, the necessary checks could be made efficiently at run time, but at the cost of a more complicated architecture. A segmented memory system could be used for dynamic bounds checking, but most segments would probably be small [15], a source of further inefficiency [16]. Hardware could be provided to interlock pipe stages dynamically, as is done on most other machines, including the VAX and M68000. But, critics have charged, this hardware is usually complex, slows the basic clock cycle of the machine, provides little functionality that cannot be provided better by the compiler, and could have been used for other purposes to enhance overall performance.

A graphic indication of the pitfalls of re-doing unnecessary work is provided by the performance of two benchmarks on the Intel 432. The highly recursive Ackermann's function, which does little except pass parameters and perform procedure calls, was the worst performing of the 432 programs benchmarked at Berkeley [17], running 26 times slower than on a VAX. After calculating the effects of a variety of software and hardware optimizations, Colwell [13] still found it the slowest, running four times slower than on a "generic", non-object oriented architecture. The crucial factor is the *acker* function's need to use the 432's protected *call* instruction, which performs extensive address-space manipulations catering to protection requirements, even though *acker* calls only itself. The sole alternative provided by the 432 architecture is a simple *branch_and_link* instruction, which is insufficient for recursive calls, since it cannot change the size of activation records.[1]

Shifting responsibility to the compiler, then, achieves two important advantages: It allows programs to execute fewer instructions, and it helps reduce architectural complexity. Both of these factors contribute to better performance. In addition to bounds checking, a compiler can take on several tasks for which architectural solutions have been proposed, such as type checking (including variant-record discriminant checking) [18], automatically initializing pointers to nil, and replacement of some procedure calls by branch-and-link instructions [13]. Increasing the compiler's responsibility means we are entrusting the correct functioning of the module more completely to the compiler; but a module cannot run successfully in any case unless it is compiled correctly. This division of labor does not, however, make the correct functioning of the *system* (including separate programs and separately compiled modules) more dependent on the compiler.

The compiler cannot be relied upon to insure inter-module protection, because it cannot guarantee the correctness of code beyond its purview. Code that it compiles can be linked with other languages, or programs from the same language compiled with older or newer versions of the same compiler. Other modules are subject to errors from source-language bugs, compiler bugs, or inconsistent assumptions about interfaces. Some errors can compromise protection, as, for example, when an arbitrary bit pattern is used as a pointer due to confusion about the number or ordering of inter-module parameters. Today's single-user systems run programs linked together from "reusable" modules supplied by programmers of uncertain competence or trustworthiness (witness the "Trojan horse" and the "worm"); this renders them nearly as vulnerable as conventional time-sharing systems. Hence some mechanism must be provided to allow new modules to be installed without jeopardizing other portions of the system.

In recent years, a variety of software mechanisms have been developed to promote inter-module integrity; e.g., Ada's compilation-order rules [19] and RCS/MAKE [20]. These systems keep track of inter-module dependencies, requiring recompilation whenever an interface changes. Although useful, these methods have their limitations. First, they work only with languages or projects that use them; and hence they cannot be relied upon as the sole means of inter-module protection. Architectural mechanisms, on the other hand, are inevitably used by all modules. Second, a bug in such a system could compromise all modules that use it; new versions of compilers sometimes contain bugs when delivered, so one would expect an inter-module control system to be subject to the same fallibility. Finally, in large software systems, a change to a heavily used module might require recompilation of a large volume of code, an expense that is sure to grow as software complexity increases. All of these factors argue for inter-module protection at the architectural level.

[1] Strictly speaking, this is not true. The 432 provides a special call instruction, *call through domain*, that can be used for intra-module calls. However, it is nearly as slow as the inter-module *call* instruction because it does all of the same address-space manipulation except for making a new *domain* object (which contains code and private data for the called module) addressable [5]

3. Factors Affecting the Performance of the Intel 432

The Intel 432 has been observed to execute slowly—early versions ran benchmarks from 10 to 26 times as slow as a VAX, or from 2 to 23 times more slowly than an 8 MHz 8086 [17]. Evidently, some of the penalty must be due to object orientation, and would be particularly obvious during procedure calls and return. To establish the performance impact of object orientation, we must first account for other aspects of the 432's architecture or implementation that influence its speed. Toward that end, we summarize a set of performance measurements that appear in greater detail in [13]. This summary includes four benchmarks:

- *acker*, a procedure to compute Ackermann's function,
- *sieve*, a prime-number finder,
- *CFA5*, a program for LU matrix decomposition, and
- *Dhrystone*, a synthetic benchmark based on a set of language and OS studies [21].

Several factors not directly related to object orientation have major effects on the benchmark measurements.

- The tendency of the 432 Ada compiler to generate spurious *enter_environment* instructions. To understand this, it is necessary to consider how a 432 program specifies a virtual address. Much as a virtual address in an ordinary segmented memory consists of a (*segment number, displacement*) pair, a virtual address in a capability-based system consists of a (*capability specifier, displacement*) pair. On the 432, capabilities are called *access descriptors* (AD's). The capability specifier selects an AD in one of four objects: the Current Context or Environments 1, 2, or 3.[2] These objects are called the four *entry access environments* (EAE's).

  EAE 0, the Current Context, is loaded automatically during a procedure call. EAE's 1, 2, and 3 are loaded by an *enter_environment*, which traverses several levels of addressing information, including access descriptors (capabilities) and object tables (directories). The first access to any object must be preceded by an *enter_environment*, but the Ada compiler generates *enter_environments* before the first access *in each basic block*. Most of these could be avoided by a compiler that performed a moderate amount of flow analysis.
- The 432 Ada compiler does not perform common-subexpression elimination, and does not re-use addresses and temporary data across several instructions.
- The compiler invariably passes **in** and **in out** parameters by value/result, even when the parameter is a large array. In the Dhrystone benchmark, passing a single array by value/result took 10 times as long as the rest of the benchmark!
- The compiler never uses the simple *branch_and_link* instruction in place of the much slower protected *call* instruction, even when the call graph would permit it (i.e., for intra-module calls that are neither recursive nor mutually recursive).
- Instructions are bit aligned in the 432, which requires the instruction decoder to reconstitute instructions after fetching and before decoding. During linear code sequences, reconstitution can be overlapped with the execution of another instruction. But for pipeline breaks such as jumps, calls, and returns, a number of cycles are lost while the execution unit is stalled waiting on the instruction decoder to flush the pipe and refill it from the new stream.
- The 432 instruction set does not provide for instruction-stream literals other than zero and one. A study performed within Intel early in the 432 project concluded that the constants zero and one would cover nearly all of the need for constants. This conclusion was almost certainly in error, but it facilitated the instruction-decoder/execution-unit split. This functional partitioning of the 432 system helped make it possible to fabricate such a complex system on silicon [22]. Since the 432's instruction stream is bit-aligned, literals would have had to be reconstituted in the instruction decoder's barrel shifter and then sent to the execution unit. No suitable transmission path existed for such a transfer.

The bar chart in Figure 1 shows the relative contribution of each of these items to the overall cycles that the 432 "wasted" on the benchmark set [13]. The pie chart indicates that approximately one-third of all cycles executed were wasted, one-third were required by intrinsic register-transfer paths, and one-third could have been saved if implementation technology had been advanced enough to allow wider buses, general registers, a separate floating-point unit, etc.

4. Procedure Calls on the Intel 432

Even if all the above mistakes had been avoided, 432 procedure calls would still have been slow. Table 1 compares a 432 procedure call with two other architectures' in

[2]The full form of a virtual address is (*selector, index, displacement*), where *selector* is a two-bit field designating the Current Context or one of the other three environments, and *index* tells which AD in the selected environment is to be used.

Table 1: Comparison of Procedure-Call Memory Traffic

	Reads	Writes	Total Bits Transferred	Total Clk Cycles
VAX 11/780	3	10	392	85
MC68010	8	13	336	94
Intel 432	16	24	1848	1022*

These measurements assume four integers passed as parameters.

*Including 282 waitstates; 740 clock cycles w/o waitstates.

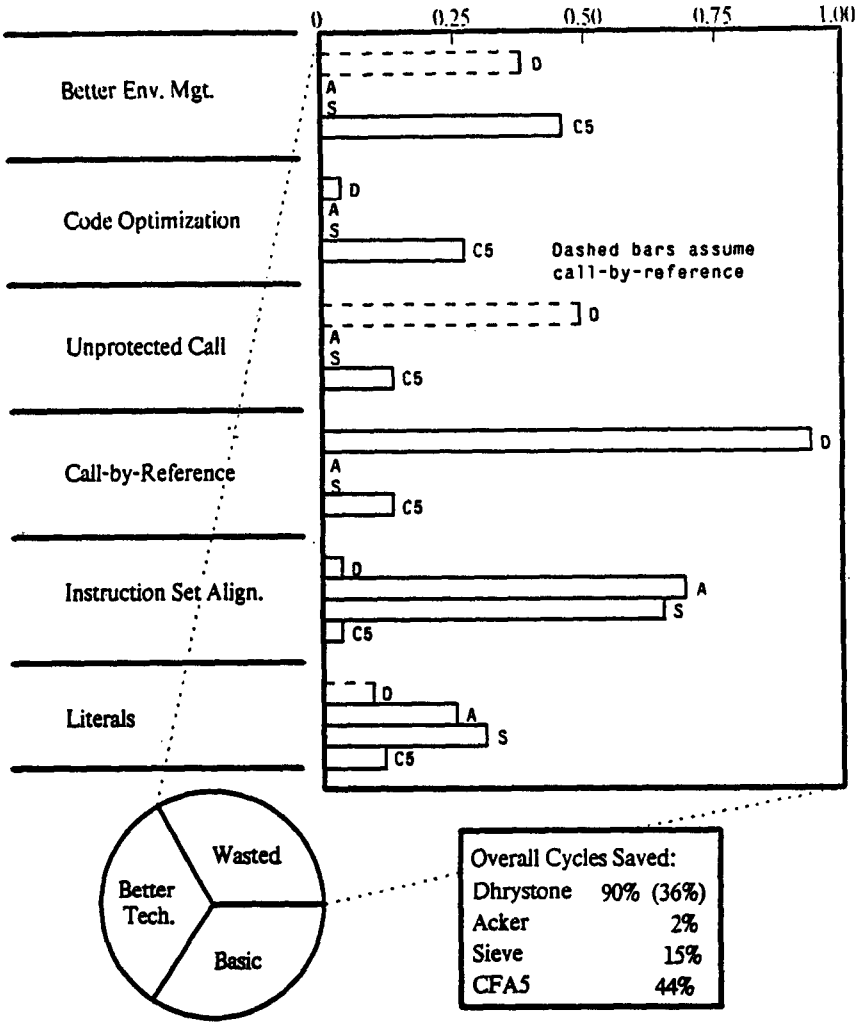

Figure 1: Relative Contributions of Cycle Sinks to Benchmark Execution Time

Table 2: How a Call Affects Fields in the Context Object

Access Descriptors		*Data Items*	
Current Context	P	Operand Stack	
Global Constants	P	Working Storage	
Context Message	P	Trace-Control Data	
Defining Domain	W	Instruction Pointer	W
Local Constants	W	Current Inst Obj DAI	W
Environment 1	C	Operand Stack Ptr	W
Environment 2	C	Context Status	W
Environment 3	C		
Calling Context	P		
Context Link	P		
Top of Descriptor Stack	W		
Top of Storage Stack	W		
IPC Message	C		
Static Link	W		

Key: P—Pre-Written; W—Written; C—Cleared

terms of the memory references they generate. Note that this measure is independent of the factors cited above. As we shall see, many of these references are performed for reasons tangential to object orientation.

At the outset, note that the 432 procedure-call timing contains 282 waitstates.[3] The Intel 432/670 development system incorporated a slow, asynchronous memory/bus interconnection that added a significant (but unspecified) delay to every memory reference, estimated at 6 waitstates [23]. Since it is possible to create faster memory bus designs for the 432, subsequent analysis will begin by assuming that waitstates have been eliminated, and that a procedure call takes 740 clock cycles.

4.1. Linked Allocation of Contexts

Each activation record created by a 432 program resides in a separate object, known as a *context object*. In Release 2.0 of the iMAX operating system, procedure calls used a linked (rather than stack-based) allocation mechanism. Hence, each call required the creation of a new context object, which is fairly expensive, since it means allocating memory to the object, entering the object in an "object table" directory, and creating an access descriptor (capability) for the object. The benefit of this strategy is that a single mechanism suffices for procedure calls, coroutines, and independent subprocesses [24]; but at the cost of several more memory references during a procedure call. Jones and Schiller [25] have compared the cost of the linked strategy with that of the stack-based strategy. They found that for an idealized implementation of each, the linked strategy required more than twice as many memory references (13 vs. 5) when no parameters were passed or returned, while both implementations required an additional two memory references for each parameter.

The cost of linked allocation is mitigated in Release 3.0, which in essence pre-allocates a chain of re-usable contexts so that procedure calls no longer need to perform dynamic memory allocation

[3] A waitstate is a clock cycle that the processor spends waiting for some memory reference to complete.

for every call. However, for a highly recursive program such as *acker* these pre-allocated contexts are depleted quickly, and the machine must soon resort to the original mechanism. Even pre-allocated contexts require five extra memory references on the 432.[1]

4.2. Anatomy of a 432 Procedure Call

At the time of a call, each context object is endowed with a number of AD's (capabilities) and several data items. Table 2 lists the items associated with each context on the Release 3.0 432, telling which are written at the time of context pre-allocation (P), or read (R) or written (W) when the call takes place. Objects in the called context which are neither pre-written nor written must be cleared so that arbitrary bit patterns left over from the last time the memory was used cannot be used as AD's pointing to random locations in memory. Because inter-module protection depends on it, clearing of memory is an intrinsic cost of object orientation. It consumes 263 cycles, or 35% of the time needed for a procedure call. In Section 5.3, we propose a technique for minimizing its impact.

The first "written" AD is the Defining Domain. It points to the *domain object* for the module to which the executing procedure belongs. It is used to access information that is "globally" accessible while executing within the module. Non-object-oriented architectures can get by with a single block of global information, but an object-oriented architecture cannot, since the globals change every time an inter-module call occurs. Hence this AD is intrinsic to object orientation.

Deferring consideration of the Local Constants AD for awhile, the next two "written" AD's are the Top of Descriptor Stack and Top of Storage Stack. The 432 uses these to reclaim storage automatically when a procedure returns. Any procedure activation may create objects. The 432 divides objects into two categories: those that are used for temporary storage which need not outlive the lifetime of the procedure activation that created them; and those that are used for more permanent storage, having a potentially unbounded lifetime [26]. Objects in the first category are allocated according to a stack discipline, and deallocated when a procedure returns.

The 432 uses a standard capability-based addressing mechanism (see Figure 2), in which location and length information for objects is found in *descriptors* located in object tables. It also uses storage-resource objects (SRO's), which contain information on allocated and unallocated regions of virtual memory. When an object is deleted, its descriptor and SRO entry[5] must be deallocated. The Top of Descriptor Stack and Top of Storage

[4]Two of these are essentially due to the fact that when a return is done, the "top-of-stack" is not necessarily at the end of the (fixed-length) calling-context object, so stack pointers must be maintained separately.

[5]If any. Not all objects have SRO entries. Some objects are created as *refinements* (sub-objects) of existing objects, and hence have no new storage associated with them.

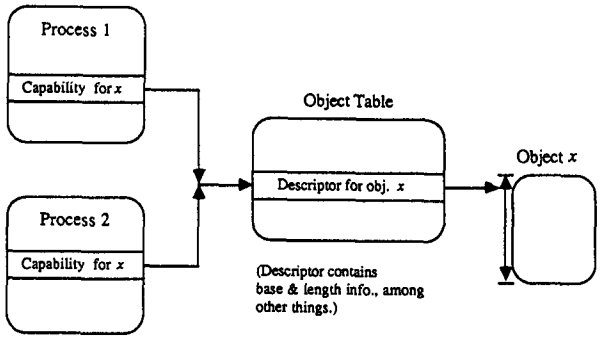

Figure 2: Capability-Addressing Paradigm

Stack AD's tell how many descriptors and SRO entries should be deallocated when the procedure returns. Although these are full-fledged AD's in the 432, in principle they could be simple integers, assuming they were protected from modification by the procedure itself (such modification could cause deallocation of objects belonging to its caller, for example). Two 16-bit integers could fit into a single 432 AD slot, and both could be initialized to 0 at the time of a call.

A fourth "written" AD, the Static Link, is needed only when lexical-level addressing is required. In fact, the 432 procedure call writes a static link only when one is supplied as a parameter to the call instruction. But static nesting of *modules* is less common than static nesting of procedures (although it is allowed in Ada).

The last "written" AD and the one of the "pre-written" AD, the Global and Local constants, are an artifact of the 432 implementation, which lacks instruction-stream literals. A future object-oriented architecture would undoubtedly include them, making these two AD's unnecessary, and saving one-and-a-half[6] memory references and approximately 19 microcycles per procedure call, as well as a similar pre-allocation cost at process-initiation time.

The first AD slot in the Current Context is occupied by a pre-written self-reference to the Current Context (EAE 0—see Section 3). Since an addressing path may start with the Current Context, an AD for it is just as necessary as the Segment-Table Base Register in a segmented memory system. However, because the Current Context is different for each procedure activation, a slot must be reserved for it in each context. The 432's four-environment addressing structure is not inherent to object orientation, but regardless of the structure used, the address-translation algorithm must have a reference from which to begin. This reference point may change with each inter-module call, hence the need to write a capability at each call. Section 5.2 will describe how this information can be held in a register rather than a main-memory location.

[6]A sixty-four bit memory reference could become a 32-bit reference.

The next pre-written AD points to the Context Message, an object that holds the parameters passed to the procedure. From an aesthetic standpoint, viewing the parameters as a message to the procedure fits nicely with the abstract notion of a procedure as an actor that receives inputs and produces outputs, and serves to emphasize the similarity between calling a procedure and initiating an independent process, which receives a message with its parameters. From a practical standpoint, the parameters could be placed at the end of the context object itself. Doing so would avoid the need for this AD. It would also save the cost of creating the Context Message object. Henceforth, we shall assume that parameters are kept in the Context Object.

The Calling Context AD is a link used during a procedure return. It would not be needed if a stack-based context-allocation scheme were used. However, because of inter-module protection requirements, even with stack-based contexts, a procedure cannot be allowed to move its stack pointer in such a way as to delete part of the context of its caller. The information necessary to prevent this (e.g., the value of the stack pointer at the time of the call), would be approximately as large as the Calling Context AD; hence the cost of writing the Calling Context AD can be considered intrinsic to object orientation. However, as we shall see, if contexts can be overlapped, it may not be necessary to write *both* the Current Context and Calling Context AD's.

Finally, the Context Link AD points to the pre-allocated context that will be used on the *next* procedure call. It is an artifact of linked allocation of contexts that would be unnecessary if stack-based allocation were used.

In summary, of the ten AD's pre-written or written at the time of a call—

- Two (Defining Domain and Current Context) are intrinsic to object orientation and would need to appear in some form in any object-oriented system.
- Two (Top of Descriptor Stack and Top of Storage Stack) contain information necessary to the 432's sophisticated memory-management system, which attempts to minimize the cost of garbage collection. They might be compressed into a single 32-bit AD slot by tighter encoding.
- One (Calling Context) is necessary, but if contexts could be overlapped, the Current Context AD from the previous context could be used instead.
- One (Static Link) is used in lexically scoped languages on non-object-oriented architectures. It is used less frequently in inter-module calls on object-oriented architectures.
- Four (Global Constants, Local Constants, Context Message, and Context Link) are artifacts of the 432 implementation, and need not appear in other object-oriented architectures.

[7]An earlier version of the 432 architecture required separate segments for AD's and data.

In addition to the AD's that are written at each procedure call, several data items are also written into the context object of the *calling* procedure. (The 432 employs the "fenced segment" approach[7] to segregating AD's from data: AD's are placed at one end of an object, and data at the other; see Figure 3). A bounds field known as a "fence" separates the two. Operations on AD's are automatically interpreted as applying to the AD portion of the object, so that data cannot inadvertently or maliciously be treated as an AD.) Two of these, the Current Instruction Object DAI and the Instruction Pointer, specify the index and displacement, respectively, of the return address. Analogous to the (*segment number, displacement*) return address in an ordinary segmented-memory system, these are intrinsic to any procedure call, object oriented or not. A third field, the Operand Stack Pointer, is used to restore the top-of-stack on a return. It would not be needed if contexts were allocated from a stack, since the top-of-stack for the caller would be the same as the base of the called context. The last data item written is the Context Status, used to tell whether the context is faulted or tracing, among other things. It provides functionality unrelated to object orientation and could be included or excluded in a future architecture depending on whether the added functionality is deemed worthwhile. In summary, of the data items written—

- Two (Current Instruction Object DAI and Instruction Pointer), are intrinsic to any procedure call, object oriented or not.
- One (Operand Stack Pointer) is an artifact of the 432's linked allocation of contexts.
- One (Context Status) provides functionality unrelated to object orientation.

A procedure call performs 40 memory references altogether (Table 3). Space does not permit a complete discussion of all of them. Four of them deserve mention because they are directly attributable to object orientation. These are used to find the instruction object for the called procedure. An AD and descriptor for the domain object must be read; the domain object contains an AD for the instruction (code) object, and the descriptor for this object is read too.

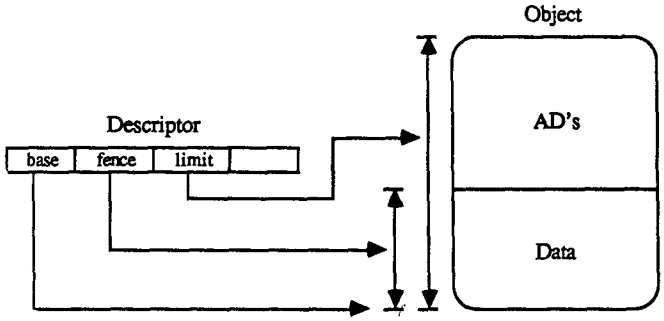

Figure 3: Segregating Capabilities from Data in the 432

Table 3: How a Procedure Call Spends its Time		
Category	Memory References	Clock Cycles
Read instruction	2	35
Otherwise needed in non-o-o calls	4	96
Unique to obj.-oriented call	8	144
Due to 432 memory management	3	55
Due to linked context allocation	5	71
Due to clearing memory	14	275
Due to lack of literals	1	9
Miscellaneous	3	39
Waitstates not included.		

The eight memory references directly due to object orientation are only one-fifth of the total, or about one-third if memory-clearing references are excluded. As a comparison, adding eight memory references to the procedure call of the VAX would represent an increase of 62%; eight references in the 68010 call would represent an increase of 38%. The microcycles attributable to object orientation are less encouraging, because they represent more than the total procedure-call time on either of the two other architectures. However, 70 of the 144 are directly due to memory references, which take 6-12 cycles depending on the width; further, the 432 microengine is not very well optimized for extracting bit fields, which is important when checking rights.

5. Improving Procedure-Call Performance

Let us consider several strategies to improve the performance of object-oriented procedure calls. We will begin by attacking the three memory references and 55 clock cycles attributable to 432 memory management. The 432 maintains a *level number* for each context allocation, in order to facilitate deallocation of objects on a LIFO basis, as noted in Section 4.2. A discussion of the exact method is beyond the scope of this paper, but may be found in [26]. The level numbers could in principle be stored in on-chip registers, but due to space constraints in the 432 processor chip, were instead maintained inside the process object. If a register were provided for this purpose, or if this storage-allocation philosophy were discarded in favor of more sophisticated storage management by the compiler, the memory references and most of the clock cycles could be avoided. Assuming the existence of registers, about six to ten cycles would be required to manipulate them, a savings of three memory references and nearly 50 clock cycles.

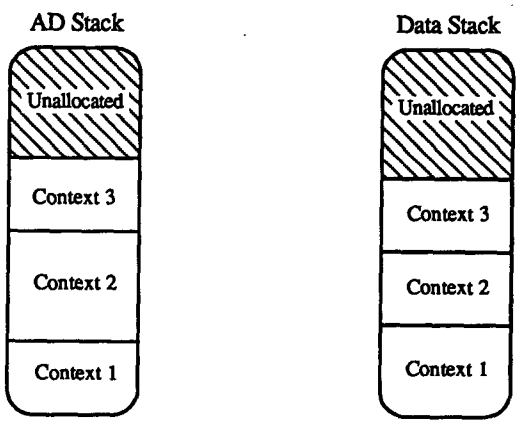

Figure 4: Contexts as Refinements of Two Stack Objects

5.1. Stack-Based Context Allocation

Since linked context allocation is a significant source of inefficiency, let us briefly sketch how stack-based allocation might be accomplished in an architecture like the 432's. The discussion will necessarily omit details, but should serve as a basis for the optimizations to be presented later.

The basic idea is to make contexts refinements (sub-objects) of a stack object. As noted earlier (Figure 3), 432 objects contain both AD's and data. Refinements must be contiguous, but all contexts must overlap both the AD and data portions of the stack object. This would require the information adjacent to the fence to be within all context objects, which clearly violates protection. Our solution, then, is to use two stack objects per process, one to hold the data portions of context objects, and the other the AD portions (Figure 4). Note that this mechanism does not suffice for coroutines or the creation of independent processes; in this case, stacks must be linked together much like "spaghetti stacks" of deep-binding Lisp systems, or the "cactus stack" of the B6700/6800. However, coroutines and subprocesses are much less frequent than procedure calls [27].

5.2. Register Sets for Addressing Information

Multiple register sets with overlapping windows are by now a well known mechanism [9] for maintaining a small amount of information in fast storage and exchanging it at each procedure call. In a 432-like object-oriented architecture, information used in address translation fits both criteria: it must be rapidly accessible and must change on each inter-module call.[8] Candidates for inclusion are the Current Context (both the data and AD portions, which are no longer contiguous), the Defining Domain, and the Top of Descriptor and Storage Stacks. It is important for Context Objects to be created rapidly; to facilitate this, a descriptor for a context object can be held in an extension of the register containing its AD (Figure 5). The descriptor need not be written to the Object

[8]None of the addressing information need change on an intra-module call. The Current Context can be expanded instead of changed. The 432 uses a different Instruction Object for each procedure, but a single Instruction Object could in principle hold all the procedures of a module.

Table in memory at all, except if the context is shared by having its AD copied or if a subprocess is initiated.

Although not directly involved in a procedure call, the Environments 1, 2, and 3 must be rapidly accessible, so three registers are reserved for them. The remaining register is devoted to an AD parameter. There is good reason to believe that a single parameter register will be sufficient for most calls. The "domain capability" approach asserts that modules should be callable only by invoking an operation on a (single) object that they implement; this is consistent with the philosophy of Smalltalk methods [3] and the MONADS system [28].

The registers may be overlapped so that the parameter remains accessible after a call. An AD for the Current Context (AD portion) remains accessible, too, and serves as the Calling Context AD. Thus, ten registers are accessible at a time.

As shown in Figure 5, a register set actually "shadows" a part of the corresponding Context Object (AD portion). It is occasionally necessary to save the register set to memory, either because the system is about to run out of register sets, or because a subprocess has been initiated. Then the registers are copied to the associated Context Object, except for the register extensions, whose information is copied into the Object Table.

Provision of the register sets saves four of the eight "object-oriented" memory accesses during a procedure call: writing the descriptor for the new context object, writing the AD for the defining domain, and reading and writing the Top of Descriptor and Storage Stacks. The savings amount to forty cycles for memory references, plus sixteen for address translation. New costs include two cycles to read and write each register and extension (except for Environments 1-3, which are written only by the callee) for a total of approximately 7 x 2 = 14 cycles. Net savings are 42 cycles, or about 30% of the estimated cost of object orientation. The register sets also save two references that would otherwise be incurred in the switch to stack-oriented context allocation: writing the Current Context and Calling Context AD's. The five memory references and most of the 71 cycles associated with linked allocation are thereby avoided.

How large would the register sets need to be? No data is available on the nesting depths of inter-module calls. As a first-order approximation, we may extrapolate from Weicker's [21] survey, which found that just over one-half (8/15) of procedure calls were inter-module. If we assume that four sets—half the number on the RISC I—are sufficient, then we need a total of 32 registers plus eight 64-bit extensions, for a total of 1536 bits, which is 37.5% of the size of the RISC I register file.

The SOAR [12] architecture employs register sets to hold data across calls in a Smalltalk program, and the Caltech Object Machine [29] maintains addressing information in an associative context cache. But until now, stack-oriented register sets have never been suggested for the addressing mechanisms of an object-oriented architecture. But it is in object-oriented architectures that they may be most appropriate. Interest among RISC designers has recently turned to algorithms for optimizing register usage across procedure boundaries as a possible substitute for multiple register sets [30]. Clearly the same approach is inapplicable to separately compiled modules. Register sets for AD's could even be combined with register-allocation optimization for data in the same architecture.

5.3. Using the Memory Controller to Clear Memory

In order to prevent unauthorized transfers of information between tasks conventional timeshared systems often clear memory used by a task before that memory is re-allocated to another task. In capability-based systems it is even more important to "sterilize" memory. Otherwise, random collections of bits could be erroneously or maliciously used to gain access to data or instructions that could ultimately bring down the entire system. The 432's procedure call spends a very high proportion of its time

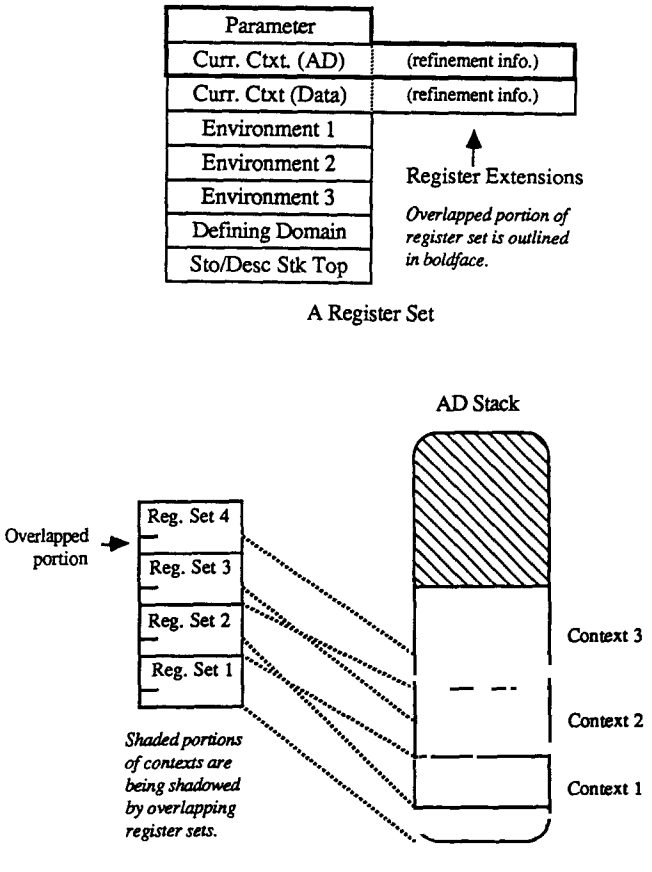

Figure 5: Register Sets and Contexts

clearing the called context's AD list (21%) and data segment (13%). This has a first-order effect on the overall length of the procedure call.

At a register-transfer level, it is clearly sub-optimal for the central processor to perform the memory-clearing operation. The processor must transfer each address to be cleared, and then the data (0), a highly redundant set of information transfers. The memory-clearing operation is so simple, however, that after the first address/data transfer has taken place, enough information has been given to the memory that the rest of the writes can be controlled locally without further assistance by the processor.

To accomplish this, the memory controller needs to have a counter, loadable from the processor-memory data bus, so that it can keep track of the number of writes being done. The memory address register can be a parallel-loadable controller, with normal writes being propagated through to the memory array. To clear a section of memory, the processor sends a control word to the controller, signifying the Clear operation. The processor then writes a zero to the first location to be cleared, and the memory controller performs the rest of the clears automatically. This mechanism has similarities to the copy-back techique proposed for the context cache in the Caltech Object Machine [29].

This strategy saves cycles in four ways. First, no waitstates are wasted on processor-memory bus transfers which are redundant anyway. Second, the memory clearing data path can be wider than the processor needs. For instance, the memory data path can be made 256 bits wide during the clear operation, allowing the clear to terminate in fewer cycles. Third, minimizing the bus traffic reduces contention with other processors. Fourth, the processor can be doing useful work while the clear is transpiring. This is true for two reasons: a clear can be initiated when memory is freed rather than when it is allocated; also, during a 432 procedure call, a very large amount of on-chip activity occurs that does not require memory accesses.

Some practical details will have to be taken into account for this idea to work. For instance, in an interleaved memory system each controller will have to make sure that only the appropriate writes are performed, skipping addresses which may not belong. If the processor (or any other processor) needs to get access to memory while a clear is in progress, provision can be made for the clear to suspend until the access is satisfied. It is worth noting that workstations with bitmapped displays often include a *BitBlt* operator, which is already capable of writing areas of memory to a given pattern (say, 0!); these could implement the memory-clearing operation at no additional cost in hardware or software. At any rate, the magnitude of the savings in the case of the 432 is such that providing hardware support for memory-clearing may be a wise allocation of resources.

If we assume that the cost of communication between the central processor and memory controller is 30 cycles (two I/O writes) and that only minimal interference occurs, the procedure call would be speeded up by approximately 31%.

5.4. Summarizing the Savings

The mechanisms we have proposed have resulted in savings in several of the categories in Table 3. The biggest reduction is in the time to clear memory, but other significant reductions are obtained by avoiding linked allocation, using multiple register sets and a register for level numbers, and from including instruction-stream literals. Table 4 summarizes the savings.

Table 4: Savings Due to Mechanisms from Section 5

Category	Memory References	Clock Cycles
Register sets for obj.-oriented call	4	42
Provision of level reg. (mem. mgt.)	3	45
Stack-based context allocation	5	60
Using controller to clear memory	14	245
Provision of literals	1	9
Totals	27	401

In total, 401 of 740 clock cycles, or 54% of the time for a procedure call has been saved. More impressively, the 13 remaining memory references and 472 bits transferred compares quite favorably with the figures for the VAX and M68010 from Table 1.

This paper has considered only the cost of a *Call* instruction and not the cost of a *Return*. However, the two are highly correlated due to the *Return*'s need to restore information from the same places the *Call* saves it. In the 432 the cost of clearing memory at a *Call* has its counterpart in the work performed to restore the addressing state for the EAE's upon return. This could be greatly speeded by expanding the 432's Data-Segment Cache to include AD's. Colwell [13] estimated that an expansion of the cache from five to nine entries could yield a hit rate of 93%, if the Dhrystone benchmark is representative of a typical large Ada program in its AD reference patterns.

6. Conclusions

The overhead of object orientation derives largely from the need to maintain and traverse more complicated addressing information. However, a reasonably sized cache usually speeds up the traversal enough to make it negligible [13], [31] in comparison to the cost of maintenance—loading the cache and updating the environment on a procedure call. This paper has focused on the second of these problems. If the cost of clearing memory is neglected, the cost of an object-oriented call has been

shown to be between 1.5 times (in terms of memory references) and 2.5 times (in terms of clock cycles) as expensive as a non-object-oriented call. There are good reasons for considering the lower bound a better estimate.

By proposing two additional mechanisms, multiple register sets for addressing information and a memory-clearing memory controller, we have shown that the overhead of a procedure call can be limited to four memory references and the associated clock cycles (plus whatever cost is associated with occasionally saving and restoring the registers). Since all of this overhead has to do with locating the procedure to be called, it is subject to speedup by more sophisticated caching schemes.

This paper has attempted to identify the cost of a particular *aspect* of object orientation. The two mechanisms it proposes could be used to speed procedure calls in any object-oriented system. It also suggests that object orientation need not unduly complicate a machine; the Intel 432 supports a wide variety of high-level programming concepts. Most of them, such as support for memory management and transparent multiprocessing are orthogonal or nearly orthogonal to the intrinsic cost of an object-oriented call.

The Intel 432 represents a first attempt at developing a production-quality object-oriented architecture. It has followed the classic paradigm for construction of any large system: straightforward implementation of the basic concepts, followed by careful measurement to identify performance bottlenecks. The results it has provided are sure to find widespread application in its successors and other future object-based systems.

References

[1] Brad J. Cox, "Message/Object Programming: An Evolutionary Change in Programming Technology," *IEEE Software*, pp. 50-61, January 1984.

[2] G.D. Buzzard and T.N. Mudge, "Object-Based Computing and the Ada Language," *IEEE Computer*, vol. 18, no. 3, pp. 11-19, March 1985.

[3] Adele Goldberg and David Robson, *Smalltalk-80: the Language and its Implementation*. Addison-Wesley, 1983.

[4] Y. Ishikawa and M. Tokoro, "The Design of an Object Oriented Architecture," *Proceedings of the 11th Symposium on Computer Architecture*, pp. 178-187, 1984.

[5] Elliott Organick, *A Programmer's View of the Intel 432*. McGraw-Hill, 1983.

[6] S.H. Dahlby, G.G. Henry, D.N. Reynolds, and P.T. Taylor, "The IBM System/38: A High-Level Machine," in *Computer Structures: Principles and Examples*, A. Newell, Ed McGraw Hill, pp. 533-536, 1982.

[7] J.R. Falcone and J.R. Stinger, "The Smalltalk-80 Implementation at Hewlett-Packard," in *Smalltalk-80: Bits of History, Words of Advice*, Glenn Krasner, Ed Addison-Wesley, pp. 79-112, 1983.

[8] M. E. Houdek, F. G. Soltis, and R. L. Hoffman, "IBM System/38 support for capability-based addressing," *Proceedings of the Eighth Annual Symposium on Computer Architecture*, pp. 341-48, May 1981.

[9] D. A. Patterson and C. H. Sequin, "RISC I: a reduced instruction set VLSI computer," *Proceedings of the Eighth Annual Symposium on Computer Architecture*, pp. 443-457, May 1981.

[10] Andrew S. Tanenbaum, "Implications of Structured Programming for Machine Architecture," *Communications of the ACM*, vol. 21, no. 3, pp. 237-246, March 1978.

[11] Butler W. Lampson, "Hints for Computer System Design," *ACM Operating Systems Review*, vol. 17, no. 5, pp. 33-48, 1983.

[12] David Ungar, Ricki Blau, Peter Foley, Dain Samples, and David Patterson, "Architecture of SOAR: Smalltalk on a RISC," *Proceedings of the 11th Annual Symposium on Computer Architectures*, pp. 188-197, June 1984.

[13] R. P. Colwell, "The performance effects of functional migration and architectural complexity in object-oriented systems," Ph.D. Thesis, Department of Electrical and Computer Engineering, Carnegie-Mellon University, August 1985.

[14] J. Hennessy, N. Jouppi, F. Baskett, T. Gross, and J. Gill, "Hardware/Software Tradeoffs for Increased Performance," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 2-11, March 1982.

[15] A. P. Batson and R. E. Brundage, "Segment sizes and lifetimes in Algol 60 programs," *Communications of the ACM*, vol. 20, no. 1, pp. 36-44, January 1977.

[16] R. M. Needham, "The CAP project: an interim evaluation," *Proceedings of the Sixth Symposium on Operating Systems Principles*, pp. 17-22, November 1977.

[17] P.M. Hansen, M.A. Linton, R.N. Mayo, M. Murphy, and D.A. Patterson, "A Performance Evaluation of the Intel iAPX 432," *Computer Architecture News*, vol. 10, no. 4, p. 17, June 1982.

[18] E. F. Gehringer and J. L. Keedy, "Tagged architecture: how compelling are its advantages?," *Proceedings of the Twelfth International Symposium on Computer Architecture*, pp. 162-170, June 1985.

[19] "Military standard: Ada programming language," MIL-STD-1815, December 10, 1980.

[20] W. F. Tichy, "RCS—a system for version control," *Software Practice and Experience*, vol. 15, no. 7, pp. 637-654, July 1985.

[21] Reinhold P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark," *Communications of the ACM*, vol. 27, no. 10, pp. 1013-1030, October 1984.

[22] George Cox, January 1985, Private communication.

[23] Konrad Lai, June 1984, Private communication.

[24] B. W. Lampson, "Fast procedure calls," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 66-76, March 1982.

[25] A. K. Jones and Lee Schiller, "Dynamic support for small domains," Department of Computer Science, Carnegie-Mellon University, September 1978.

[26] F.J. Pollack, G.W. Cox, D.W. Hammerstrom, K.C. Kahn, K.K. Lai, and J.R. Rattner, "Supporting Ada Memory Management in the iAPX-432," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, March 1982.

[27] L. Peter Deutsch and Alan M. Schiffman, "Efficient implementation of the Smalltalk-80 system," *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 297-302, January 1984.

[28] J. L. Keedy, "Support for software engineering in the MONADS computer architecture," Ph.D. Thesis, Department of Computer Science, Monash University, August 1982.

[29] William J. Dally and James T. Kajiya, "An object-oriented architecture," *Proceedings of the Twelfth International Symposium on Computer Architecture*, pp. 154-161, June 1985.

[30] D. A. Patterson, "Reduced instruction set computers," *Communications of the ACM*, vol. 28, no. 1, pp. 8-21, January 1985.

[31] E. F. Gehringer, Z. Z. Segall, and D. P. Siewiorek, *Cm*: an Experiment in Multiprocessing*. Digital Press, 1986.