Performance Effects of Architectural Complexity in the Intel 432

ROBERT P. COLWELL
Multiflow Computer, Inc.
EDWARD F. GEHRINGER
North Carolina State University
and
E. DOUGLAS JENSEN
Kendall Square Research Corp.

The Intel 432 is noteworthy as an architecture incorporating a large amount of functionality that most other systems perform by software. It has, in effect, "migrated" this functionality from the software into the microcode and hardware. The benefits of functional migration have recently been a subject of intense controversy, with critics claiming that a complex architecture is inherently less efficient than a simple architecture with good software support. This paper examines the performance impact of the incorporation of several kinds of functionality into the Intel 432. Among these are the addressing structure, the caches, instruction alignment, the buses, and the way that garbage collection is handled. A set of several benchmarks is used to quantify the performance effect of each of these decisions. The results indicate that the 432 could have been speeded up very significantly if a small number of implementation decisions had been made differently, and if incrementally better technology had been used in its construction. Even with these modifications, however, the 432 would still have only one-fourth to one times the speed of its contemporaries. These figures may represent the real cost of the 432's style of object-based programming environment.

Categories and Subject Descriptors: B.5.m [**Hardware**]: Register-Transfer-Level Implementation—*miscellaneous*; C.1.1 [**Processor Architectures**]: Single Data Stream Architectures—*SISD processors*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures—*MIMD processors, parallel processors*; C.1.3 [**Processor Architectures**]: Other Architecture Styles—*capability architectures, high-level language architectures, stack-oriented processors*; C.4 [**Computer Systems Organization**]: Performance of Systems—*design studies*; D.3.2 [**Programming Languages**]: Language Classifications; D.3.4 [**Programming Languages**]: Processors—*compilers*

General Terms: Design, Measurement, Performance, Security

Additional Key Words and Phrases: Ada, address translation, caches, capabilities, CISC, Intel 432, object-based, object-oriented, procedure calls

This research was supported in part by the U.S. Army Center for Tactical Computer Systems under contract DAAB 07-82-C-J164. The cooperation of the Intel Corporation is also gratefully acknowledged.
Authors' addresses: R. P. Colwell, Multiflow Computer, Inc., 175 N. Main St., Branford, CT 06405; E. F. Gehringer, Department of Electrical & Computer Engineering, Department of Computer Science, North Carolina State University, Raleigh, NC 27695-7911; E. D. Jensen, Kendall Square Research Corp., 1 Kendall Square, Cambridge, MA 02139

1. INTRODUCTION TO THE INTEL 432

The Intel 432 was conceived in 1975 with several goals in mind. The primary objective was to produce a system that would improve the software programming environment, thereby lowering system life cycle cost. To achieve this goal, the 432 incorporates architectural run-time support for both data abstraction (programming with abstract data types) and domain-based operating systems. The principal insight of the 432 architecture is that both objectives can be supported by a common semantic model, known as the *object model* [22, 27].

Another goal, finding a design approach that would yield a range of performance from a single implementation, led to the concept of transparent multiprocessing. By using the *object model* the 432 system is able to run with a variable number of physical processors without recompiling the software. Supporting so many different concepts for the first time in a new machine inevitably led to a complex architecture. Moreover, initial measurements of the 432's performance were quite discouraging. Subsequent architectural enhancements improved execution speed, but not enough to satisfy either its champions or its critics. It is therefore appropriate to examine the design of the 432 and to investigate the factors that underlie its performance.

1.1 System Architecture

The Intel 432 is notable as an "object-oriented" or "object-based"[1] architecture. Just as the Smalltalk language [12] encapsulates all data into objects that can be manipulated in carefully constrained ways, the 432 imposes similar restrictions on all online storage:

—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.

The Intel 432 is also a shared-memory multiprocessor [1, 2, 17, 18, 19]. Figure 1 shows a simplified block-diagram configuration. (More elaborate 432 system architectures can readily be constructed for improved reliability or availability [21].)

The GDPs in Figure 1 perform essentially the same role as do the CPUs in more conventional systems. The main differences are that the GDPs perform no I/O (the IPs do that) and the GDPs are self-dispatching via routines provided in their on-chip microcode.

The 432 instruction set is notable for several reasons.

—Its instruction set is bit-aligned and encoded so that programs will be as compact as possible.

[1] The designers of the 432 referred to their system as "object-oriented," but more recently the term "object-oriented" has become understood to mean that a language or system supports inheritance [34].

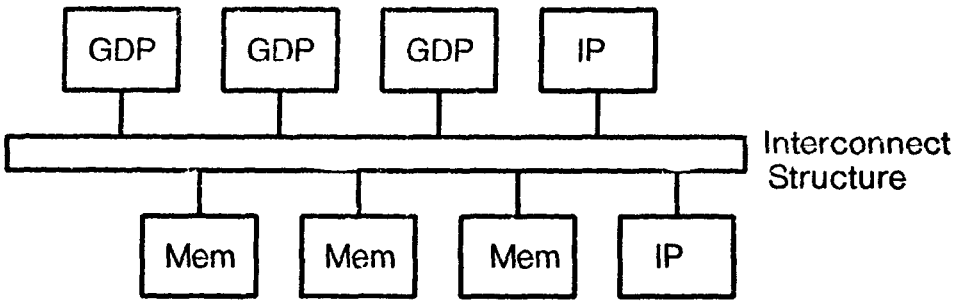

Fig. 1. Generic 432 system multiprocessor architecture.

—Its instruction set is very complex.[2] It has over 200 instructions ranging from *branch* to *Send* (an interprocess communication primitive).
—Each instruction can make 0, 1, 2, or 3 explicit data references, any of which can be to a scalar, record item, static array element, dynamic array element, or stack, and each can be direct or indirect.
—Neither instruction stream literals nor general data registers are included.
—The instruction set is complete, orthogonal, and symmetric, fully supporting each primitive data type with its own operations.

The ramifications of the bit-alignment scheme will be discussed further in Section 5.3, and the performance effects of the lack of registers and literals in Sections 5.2 and 5.4.

1.2 Functional Migration in the 432

There are several types of functional migration present in the 432 architecture. The first type could be classified as "object" related; for example, the base/length registers that serve as a cache in order to avoid repeated full traversals (lookups of addressing information, described in Section 3) of the 432 addressing path. One could also view the microcode that directly manipulates the object headers and performs the automatic rights checks (read/write access) as a migration of function, since what few checks are performed in operating systems such as UNIX™ are implemented in software.

The second type of function is intended to support high-level languages. The 432 instruction set can directly express in a single machine instruction high-level operations such as

A[i] := B[k] * C[j];

and

Structure.element := A[i] * D;

[2] The complexity of the 432's instruction set is evident mainly in the implementation resources required to realize it (the Instruction Decode Unit occupies an entire chip). Conceptually, it is not complex because of the instruction set's completeness and orthogonality.
™ UNIX™ is a trademark of AT&T Technologies.

This was expected to yield more compact and faster-executing code. For every supported operation in the instruction set, all machine-defined data types are supported (*char*, *integer*, *short integer*, *ordinal*, *short ordinal*, *real*, *short real*, *temporary real*) in the expectation that this simplifies the compiler writer's task. Machines such as the VAX and the S-1 have traditionally offered the same justification for having complete and symmetric instruction sets, but substantial doubt exists as to the overall importance of this design principle in light of the current predominance of high-level language usage over assembly language [7, 39] and the prospects for automatic generation of compilers [24]. Since the HLL users never see the machine's native instruction set, one of the important motivations for completeness is removed.

A third type of function migrated in the 432 deals with applications. The 432 GDP implements the IEEE standard 754 for both single- and double-precision floating-point formats.

A fourth type of migrated function comes from operating systems. The 432 subsumes operations such as interprocess communications, process scheduling, processor dispatching, virtual memory management, and I/O into its architecture. Except for I/O, other machines almost always perform these operations in software. The 432's Interface Processor transacts all I/O, allowing the GDPs to exist in an environment where there are only objects. To do this the IP performs whatever conversions are necessary between that environment and the real world of bytes, disks, and networks.

The 432 hardware, microcode, and OS software together implement a distributed fault handler that handles faults arising from any source, hardware faults, applications code run-time errors, explicit exceptions raised at the source code level, or memory management faults. Such a unified fault model contrasts sharply with the ad hoc, localized approaches found on machines such as the VAX, where run-time errors may be caught by the run-time system, the hardware, the memory management unit, or the operating system, each handling the fault in its own way. Microcode devoted to fault handling constitutes another kind of functional migration.

2. THE EXPERIMENTS

2.1 Performance as a System Metric

Since the primary goal of the 432 system was to improve software productivity and thereby lower life-cycle cost, it would seem logical to attempt to evaluate the 432's functional migrations according to how well the system met those goals. Unfortunately, the 432 was not a commercial success, and not enough large-scale programming development efforts exist for such an evaluation.

This paper concentrates on the performance effects of those functional migrations, leaving questions about the cost-effectiveness of object-based systems design for future research. It is important to note the scope of this investigation into performance. We do not assume here that any and all aspects of system architecture or implementation are fair candidates for alteration or disposal as long as overall system throughput on the benchmarks appears to be higher. Here

we pursue the question of how large an inherent overhead object orientation appears to be, and how effectively functional migrations can be used to combat that overhead. Consequently we seek the highest performance subject to certain run-time constraints that are intrinsic to the 432 class of object-based systems.

The reader should note that this paper is not about object-based systems per se; we do not pursue any of the myriad ways in which a real follow-up to the 432 might break radically with the 432 to achieve much higher performance (we suggested some in [10]). We view the 432 as an expensive experiment, the results of which have been badly misinterpreted. In order to draw supportable conclusions from this experiment, only incremental performance enhancements to the 432 could be investigated in this work. These enhancements were considered with the aid of the 432's silicon architects to insure feasibility. (These incremental enhancements, together with remedies for some basic errors built into the 432, suffice to improve performance to the realm of commercial viability. This conclusion is a key result of this work; we will return to it in Section 6.)

The performance of the 432 was evaluated using a set of benchmarks to drive Intel's Release 3.0 432 microsimulator, which was written in Simula and runs on a DEC KL10 at Intel in Aloha, Oregon. The simulator accepts 432 object code files and can output a cycle-by-cycle listing of the GDP's operations, including instruction fetches, memory accesses, and internal operations. This simulator created cycle-by-cycle log files, which were then analyzed via a suite of C programs created specifically for this purpose. Proposed architectural changes to the 432 were modeled with these programs or manually, using the number of cycles per instruction in the log files as a guide.

2.2 Benchmarking

Measurements of a real computer system must be made under some processing load. The art of benchmarking has evolved to provide programs that, taken as a whole, are thought to be representative of the processing load seen by the machine in actual use. Benchmarking is still an art, however. Little agreement exists on how to even characterize typical processing loads, much less to create benchmarks that accurately represent these loads. Even representative small-scale processes do not capture or duplicate such important systems-level conditions as process swap overhead and I/O interrupts. For systems with caches, small benchmarks may fit entirely within the cache, exaggerating the performance benefits of such caches [35].

Another issue in measuring system performance concerns the load represented by operating system code. Since there is wide variability in the use of OS functions, it can be very difficult to characterize that load, but it is often estimated that a substantial number (greater than 50 perecent) of the processor cycles are typically dedicated to the operating system. Processing loads of that magnitude cannot be ignored.

Despite the problems associated with using benchmarks, their use can be of great value in architectural design. If the benchmarks are constrained to be implemented in a single language, and for a single architecture that is

incrementally changed in various ways, then it is possible to draw unambiguous conclusions about the effects of those architectural changes.

2.3 Programming Environments: Large vs. Small

Benchmarks such as those used in the RISC work at Berkeley [7] and those used in the Military Computer Family study [9] can be described as "low-level": They attempt to exercise the primitive instructions in the machine, and are generally argued to be representative and meaningful because those operations constitute the bulk of a processor's instruction executions.

The 432 was designed to support large programming development environments, hence the ultimate verdict on whether it meets its design goals should be based on measurements indicative of such large-scale systems, and not on low-level benchmarks. Nevertheless, much can be learned from low-level performance studies. The reasons are as follows.

Even if the high-level functions of the 432 (interprocess communication, procedure calls) were free, executing in zero time, the performance of the 432 as reported in [13] would still be slow relative to other current systems. Establishing the reasons for this poor low-level performance can yield important insights about the design of object-based architectures, and complicated architectures in general.

Although only a few "programming in the large" systems exist for the 432, two of them are available for static module connectivity measurements (the 432 UNIX study at the University of California at Santa Barbara, and a large Ada development on the 432 done at Hughes Aircraft). In addition to these, the Carnegie-Mellon Mercury Mail System was written in Ada and is available for this study. These results will be compared to the Dhrystone benchmark (Version 1.1) developed by R. Weicker [35]. Dhrystone is a synthetic benchmark based on a set of language and OS studies. Taken together, these large-scale systems data will be used where architectural features have not been sufficiently exercised by the low-level benchmarks. Weicker argues that the Dhrystone is quite representative of loads in general, not just those imposed by operating systems. This paper will rely heavily on a combination of the low-level benchmarks and the Dhrystone in driving the 432 simulator.

3. ADDRESS TRANSLATION IN THE 432

The 432 has been called "one of the most sophisticated architectures in existence" [25]. Figure 2 supports this contention: It shows the full addressing path of this machine. Understanding this addressing mechanism is the key to understanding much of the implementation of the 432. This section will discuss the motivations for this kind of addressing mechanism. It is important to realize that the objects (for example, the Object Table Directory, the Object Tables, and the Context Object) depicted in Figure 2 all reside in physical memory, and on-chip caches are provided to ameliorate the potentially debilitating effect on performance of this chain of indirections. How well these caches work for low-level benchmarks is a major topic of this paper.

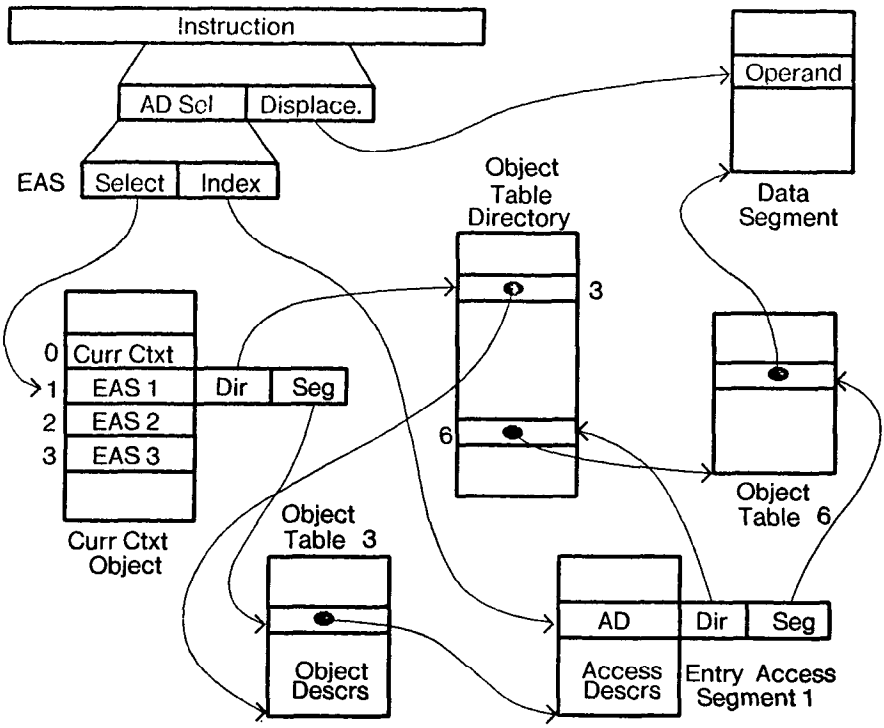

Fig. 2. The 432's full addressing path.

3.1 Protected Pointers

The arcs in Figure 2 represent *protected pointers*. The concept of a protected pointer is intrinsic to an object-based computer *architecture*.[3] A pointer is "protected" in that the user program cannot manipulate it directly (as one could in the C language; e.g., *int *ptr*; *ptr++*. The underlying architecture, often microcode, manipulates these pointers on behalf of the user program according to a set of rigid constraints. By structuring object accesses around this system-controlled pointer mechanism, it becomes possible to perform rights checks, operation type checks, and other system functions at the time when they are needed and only on those objects that are immediately affected.

Object-based machines must take special care that these pointers cannot be forged, for the object addressing mechanism will be relied upon completely and implicitly in ensuring system integrity. The 432, for example, does not even have a machine instruction for creating a protected pointer; this operation is performed by microcode. This is in sharp distinction to conventional architectures, where user programs perform both address and data calculations on the same hardware (registers and ALU). Barring microcode bugs in the 432 GDP, an errant user program can crash itself but cannot bring down the 432's operating system.

[3] As opposed to an object-based *language*, in which the integrity of pointers is assured by the compiler.

Various means have been explored for protecting these pointers. Systems such as the Burroughs 6500 [37] use tagged memory locations in order to distinguish the pointers from data. Other systems, including the IBM Sys/38 and the Intel 432, permit these pointers (called "capabilities" in these systems) to reside in the same address space as the data. They are distinguished from data via additional information contained in object headers. A capability is a protected pointer that has associated with it information on the range of operations that capability possesses for the object being referenced.

The use of capabilities as the basic addressing mechanism has some important ramifications on the design of the underlying structure of the architecture. A conventional architecture usually associates access rights with physical pages of memory (for example, placing the object code for system utilities into pages marked as "execute-only"). Capabilities provide a means to separate program-level concerns, such as modules and data structures, from irrelevant details such as physical memory sizes, paging characteristics, and disk structures. Advocates of the object-based programming style cite this separation of concerns as essential to improving the match between the programmer's desired abstractions and the machine on which his or her program must execute. However, the cost of memory accesses has a first-order effect on performance in conventional machines, and the additional manipulations implied by object orientation will only make it worse. Consequently, it is very important that we establish what the object overhead is, and to what extent that overhead can be removed by architectural support and other means.

3.2 The Intrinsics of 432 Object Orientation

The performance effects of the 432's object orientation are manifested in three major ways. First, procedure calls and returns are slowed substantially by the increased amount of information that must be dealt with in an object-based environment. Some of this information describes the types and access rights of various objects, and some represents addressability information on those objects needed immediately by the called procedure. Second, addressable *domains* (e.g., objects and procedures) must be made accessible via explicit *enter* operations. This operation, or sequences of them, are needed when executing intermodule calls or when traversing pointer chains. Third, every memory reference is checked for read/write privileges on the object being accessed, and every reference is checked to ensure that it lies within the physical boundaries of the object.

Conventional architectures assume that whatever bit pattern appears as the address of a datum or called procedure is correct unless the underlying memory management mechanism indicates otherwise. This approach minimizes run-time checks at the risk of yielding unauthorized or faulty access to routines or data by possibly malicious users. By contrast, the context in which a capability-based, object-oriented procedure executes is such that it is not even *possible* to access objects that have not been explicitly made available—a "need-to-know" arrangement. This strategy aims to contain system damage from faulty or malicious operations, and provides a closer semantic match to the programmer's abstract model of the processor. Since the object-based approach requires that all entities

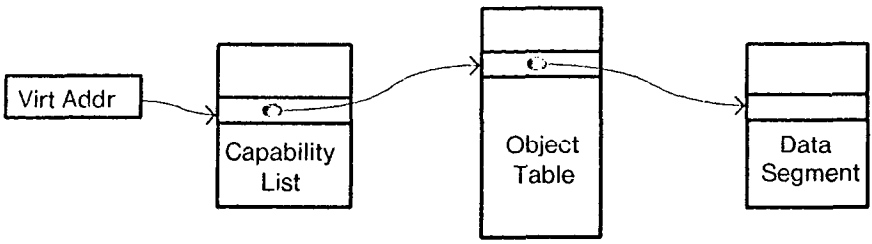

Fig. 3. A two-level addressing mechanism.

that will be addressed first be "qualified," or checked for validity (with a local-pointer/object-length pair established), this additional data manipulation appears as a degradation in the performance of object-based procedure calls and returns.

For normal execution of applications programs, the 432 system architecture was configured so that a "working set" of objects would become qualified through appropriate execution of *enters*. After object qualification (and data-segment cache updating) the addressing overhead should be no higher than on conventional architectures (providing that memory reference checking is done in parallel with the reference, as it is on the 432). However, for executing inter-module calls, or for traversing a chain of pointers, *enters* may have to be executed repeatedly, especially if the compiler does not perform code flow analysis.

Every memory reference made in the 432 is checked for access type violations (read/write) and displacement range violations (the object length is determined at object qualification time and stored on-chip). These checks are performed by hardware/microcode on the 432. (The equivalent cost of performing them in software is discussed in [5].)

3.3 The Addressing Structure

As in Hydra/C.mmp [40], STAROS[11], and CAP [38], the 432 employs a two-level addressing scheme. Figure 3 depicts this mechanism at its most abstract level.

The justification for two-level addressing is too subtle to be fully dealt with here; interested readers are referred to [8, 17, 25, 26]. Basically, a two-level addressing scheme allows information such as the location, size, and type of an object to be stored and manipulated independently of the rights various programs have to those objects. Contrast this with the inflexible protection and sharing available on standard architectures, which base their access checks on page tables or other artifacts.

The 432 implements two-level addressing as follows. Any memory reference begins with two pieces of information: an Access Descriptor (AD) selector, which refers to the object being accessed, and an offset into that object. ("AD" is the name that Intel uses for "capability.") The AD selector consists of an environment selection (one of four available environments) and an index into the list of capabilities available within that environment. The capability, or AD, selected from that environment refers to some object, and the physical address of the base of that object is found by using the Directory and Segment fields of the AD as indices into two object tables.

There are several reasons for having two object tables (the Object Table Directory and the Object Table) "between" the AD selected and the data object. Since the Directory and Segment fields of an AD are 12 bits each, collapsing all Object Tables into one would require that the object table be potentially 2^{24} entries long. The 432 allows objects to be at most 64K bytes long, and could not easily implement such an object. Most important, the Object Table would not be swappable, and most of it would be of no use at any given time, since only those entries accessible by the current process need to be present. It would therefore waste a great deal of physical memory, causing more swapping of the other objects in the system.

This explains why the Object Table Directory and Object Table 6 in Figure 2 exist. Object Table 3 exists because the 432 treats all information in the system in the same way: as objects. As a consequence, the list of capabilities in the Entry Access Segment reside in an object, so that object must be accessed exactly as any other object would, through the conversion of an AD for the object into the associated OD (object descriptor) and then into physical addresses.

3.4 Address Caches

The 432 incorporates two associative address caches in order to minimize the amount of memory traffic required to perform a virtual-to-physical address translation. Figure 4 shows the locations and sizes of these caches.

The 432 provides a set of 23 base/length register pairs, which together contain a great deal of information about the current state of the processing environment. Five of these register pairs form a data-segment cache, four register pairs form an object-table cache (the placement of these caches is shown in Figure 4), and the rest are dedicated to holding information on various system object segments. The caches are searched associatively and their contents managed according to a least-recently used replacement algorithm.

Cache size in standard computer systems is of critical importance in determining overall performance [4, 32]. The caches shown in Figure 4 contain base addresses of objects, however, not actual data values; hence it is the locality of reference to objects that determines how effective the 432's address caches will be, rather than frequency of reference to particular data values. Section 5.6.1 will discuss the performance effects of the cache sizes incorporated in the 432.

Two caches are conspicuously absent in Figure 4. The first is a data or instruction cache, which would normally contain hundreds or thousands of associative entries in order to have a usefully high hit ratio. The address caches we are concerned with on the 432 have fewer than ten, partly due to severe chip-space constraints. As a consequence, we will not consider on-chip data or instruction caches in this paper. The other "missing" cache is an AD cache, which would not be flushed at procedure call boundaries. This cache could save a great many memory references, which occur because the *entered environments* are invalidated at each procedure call, causing the data-segment cache to be flushed. If each procedure is accessing a global object (e.g., the Puzzle benchmark described in Section 4), the data-segment cache is considerably less effective than it might otherwise be.

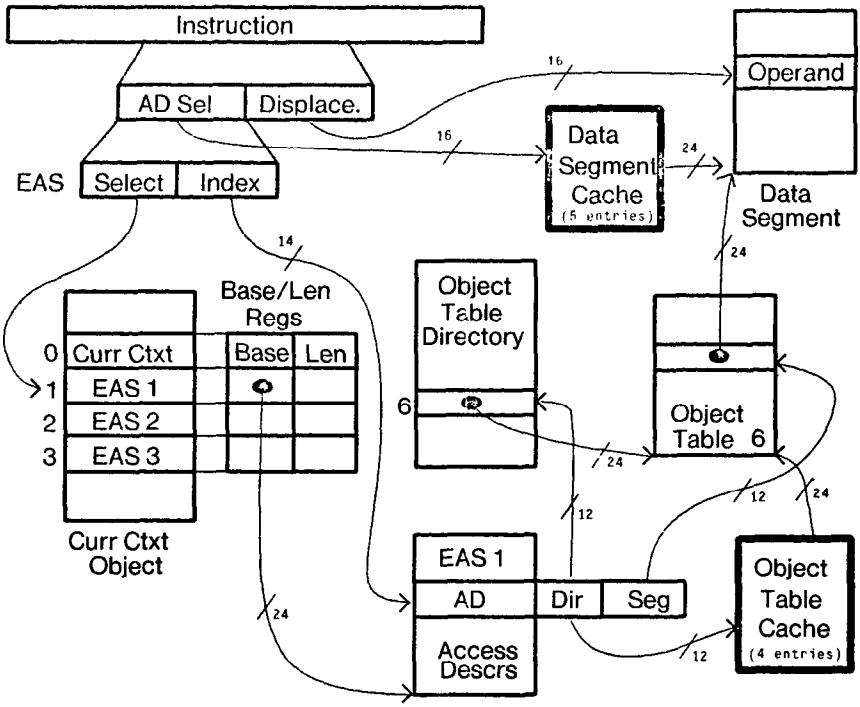

Fig. 4. 432 on-chip address caches.

3.5 Rights Checking

Access Descriptors contain the information describing the rights a program accessing an object has to that object. For example, read, write, and delete fields convey those respective rights to the bearer of the AD. A "type-rights" field encodes more general rights, such as the "return" right a called procedure must have to its caller's activation record (called a "context" in the 432).

Object descriptors contain information about the objects themselves, such as physical location, size in bytes, type of object, and other object-management information. Many object types are predefined in the architecture, such as context objects, processor objects, and port objects. These objects are referenced automatically by the 432 microcode on behalf of a user program during normal execution.

Tracing through an example will give an idea of the types of checks that are performed in the course of an instruction's execution. Table I shows a segment of Ada source code (part of the Computer Family Architecture benchmark 8: Hash-table search).

Function HashLook is called with three arguments, a record, and two integers. The integers are moved into the data part of the message (MSG) object, and an AD to the data object containing the record is placed in MSG's access part. Since the Called Context already has an AD to the MSG object in its access list, the integers are directly accessible.

Table I. Rights Checking Example: Ada Source Code Segment from CFA8

```
function HashLook (Table: in TableType;
                   size: in integer;
                   Kkey: in integer)
return BigRec is
check,I :integer;
Full:boolean;

begin
  check := Kkey MOD size;
  Full := FALSE;
  FOR I in 1. .size/2 loop
    IF Table (check).key=Kkey OR Table(check).key=0
      THEN
        return (Table(check),false);
      end IF;
    check := (check+I) MOD size;
  end loop;
  Full := TRUE;
  return ((0,0),true);
end HashLook;
```

Table II. The Enter Environment Algorithm

1. Clear the Data Segment Cache entries associated with the old environment
2. If AD is "access valid" then
   a. remove delete rights from the AD before placing it into the current context
   b. open the new access segment to force possible faults early
   c. set the "copied" bit in the OD associated with this AD
   d. get the "level" from the associated OD
   e. if OD entry type is not "storage" then get level from the Base object rather than this refinement
3. else, if AD is not valid, set "level" to its maximum value so that future attempts to store AD's into this access segment will fault
4. store level number into the Process Data Segment
5. store the AD into the memory image of the current context

The MSG is actually implemented as a refinement of the Calling Context. This saves time in setting up parameters to be passed. Unfortunately, this scheme forces the Called Context to traverse the refinement (reference the refinement's object-table entry) when setting up the data-segment cache pointer to the MSG object, at a cost of approximately 77 clock cycles. The ramifications of this scheme will be discussed in Section 5.6. Since there is no AD for the Table record in the Called Context, it cannot yet be accessed. To make the Table record accessible an *enter_env* operation must be performed.

The *enter_env* instruction has the effect of copying the AD to the MSG object into the entered environment so that the ADs contained in the MSG become directly usable. Table II shows the sequence of operations involved in an *enter_env* (taken from [19] and [20]). (The "levels" referred to by the algorithm

Table III. Memory Operations in Executing Enter_Env

Access	Size	Purpose
1. Read ASLoad	16 bits	get constant for referring to new AD
2. Read Word	32 bits	get AD specified by AS (just read above)
3. Write Byte	8 bits	set copied bit of OD
4. Read EWord	80 bits	get OD for refinement object
5. Read Word	32 bits	get access part offset + length of refmt
6. Read Word	32 bits	get OD for base object
7. Read DByte	16 bits	read the level from base object
8. Write DByte	16 bits	update level of EAS 1 in Process Object
9. Write Word	32 bits	write AD image to mem copy of curr ctxt

are the mechanism the 432 uses to solve the dangling-reference problem. They are discussed in detail in [30].)

The *enter_env* instruction is part of the basic mechanism by which the 432 controls object accessibility. In order for a program to access an object, it must first establish its right to that object by arranging for an AD to the object to be placed into one of the four access environments of the current context (the current context itself, of Env's 1, 2, and 3). Thus the program must already have the appropriate AD. In the course of writing the AD into the current context, the 432 microcode writes the new access selector into the on-chip Data Segment Cache and invalidates any current entries for that environment. When a subsequent instruction attempts to use that environment as part of its access, the Data Segment Cache will miss and the microcode will pause in the current instruction's execution in order to refill the cache.

In carrying out the execution of the *enter_env* instruction, a number of checking operations were performed. Before using any object, the 432 microcode and hardware test validity by comparing the object's type, length, level, and other fields, as well as the AD and OD being used in the access. In this example, some of these checks were made at some earlier time (for instance, checking for process and context validity) and are therefore not repeated. Other tests are performed explicitly upon accessing an object for which the processor currently has no knowledge.

In a typical case, an *enter_env* executes the algorithm given in Table II and makes the sequence of memory references given in Table III. The first access that the *enter* makes is to the local constants object. The constants object is required because 432 lacks instruction-stream literals (Section 5.4). Unless this object is currently qualified, the rights possessed by the referencing AD will be checked against the object-type information contained in the OD, just as for any other object.

Once an object is qualified, subsequent accesses to that object are efficient, since the offset specified by the instruction stream is added to the object's base address (contained in the DS_Cache). While that reference proceeds, the object length (also contained in the DS_Cache) is compared to the base + offset address to ensure that the reference does not lie outside the object. Since arrays are allocated to separate objects, this makes dynamic array-index checking automatic in the 432.

3.6 Procedure Calls

Procedure calls and returns are potentially expensive operations in object-based systems like the 432, due to the large amount of state information associated with each context. On a conventional architecture, the graph structure of a program's call patterns and data structure accesses is represented in the form of virtual addresses embedded in the object code. In object-based systems—at least for intermodule calls—this graph information is explicitly preserved in the form of capabilities for objects such as instruction segments, data structures, and messages. These capabilities must be manipulated by machine instructions.

The 432's procedure calls are quite costly. A typical procedure call requires 16 read accesses to memory and 24 write accesses, and it consumes 982 machine cycles. In terms of machine cycles, this makes it about ten times as slow as a call on the MC68010 or VAX 11/780. The reasons for this complexity are discussed in [10]; object orientation is shown to be only a minor factor. Space does not permit a recapitulation of the analysis here.

4. RAW PERFORMANCE MEASUREMENTS ON THE 432

A 1982 paper compared the performance of the 432 with other contemporary machines on a set of four low-level benchmarks [13]. The benchmarks used were *search*, a string search routine; *sieve*, a prime-number program; *puzzle*, a binary bin-packing program; and *acker*, a short, highly recursive routine. Intel's Release 2.0 (4 MHz) 432 was reported to execute the benchmarks very slowly compared to the VAX 11/780, the 68000, and the 8086 (Table IV). At best (the *search* benchmark), the 4-MHz 432 was 10 times slower than the VAX; at worst (*acker*), it was 26 times slower. Compared to the 5-MHz 8086 the 4-MHz 432 ran between 2 and 23 times more slowly. Its performance with respect to the 8 MHz 68000 was similar. We ran these benchmarks on our Release 2.0 432 system at Carnegie-Mellon University, and our results corroborated the Berkeley numbers except for a constant speedup attributable to the 5-MHz clock in our system.

It is hard to draw architectural conclusions from these comparisons. The 432 was programmed in Ada, while the other machines used Pascal or C, so differing language semantics and compilers are reflected in the measurements. The 432 measurements used Release 2.0 of the operating system, which was the first version distributed by Intel. Release 3.0 incorporated several improvements, including preallocation of contexts to improve the speed of procedure calls. In addition the architectures being compared vary in implementation technology, memory speed, and bus systems. The most useful comparison is between the Release 2.0 432 and the 5-MHz 8086. Although the languages and compilers do differ, the implementation technology is the same, and the systems environment (bus and memory technology) for each is similar. Why does the 432 take over four times as long as the 8086 to run *sieve*? Why does *acker*(3, 6) take 21 times longer?[4] Section 5 will investigate these questions.

[4] Our Release 2.0 432/670 system could not run *acker*(3, 6) to completion. The system hung after the 72,461st procedure call. Calls were nested 752 deep at this point. We suspect that the system ran out of physical memory, which would hang the system, since virtual memory was not implemented in Release 2.0. The elapsed time reported for *acker*(3, 6) is estimated, based on the simulated *acker*(1, 2).

Table IV. Berkeley 4 MHz Intel 432 Measurements

Machine	Language	Word size	Time (in milliseconds)			
			Search	*Sieve*	*Puzzle*	*Acker*
VAX 11/780	C	32	1.4	250	9400	4600
	Pascal (UNIX)	32	1.6	220	11900	7800
	Pascal (VMS)	32	1.4	259	11530	9850
68000 (8MHz)	C	32	4.7	740	37100	7800
	Pascal	16	5.3	810	32470	11480
	Pascal	32	5.8	960	32520	12320
68000 (16 MHz)	Pascal	16	1.3	196	9180	2750
	Pascal	32	1.5	246	9200	3080
8086 (5 MHz)	Pascal	16	7.3	764	44000	11100
432 (4 MHz)	Ada (rel. 2)	16	35	3200	350000	260000
	Ada (rel. 3)	16	14.2	3200	165000	260000
	Ada (rel. 3)	32	16.1	3200	180000	260000

Presented below are the "baseline measurements" of the Release 3.0 432. These results were obtained by analysis of the simulator log files, with no adjustments made for any of the architectural or implementation problems that will be discussed later. While these baseline results are not in themselves very helpful in analyzing the 432 system, they indicate the real performance that 432 users experienced. These results also provide the starting point against which architectural modifications, compiler changes, and implementation decisions can be evaluated.

Table V shows the number of instructions executed per benchmark. Table VI shows the total cycles required to execute each benchmark, from the first macroinstruction of the benchmark through the final *return*. I/O was not included in any of the benchmarks simulated here. The Dhrystone benchmark result reflects a source-level programming change that forces a particular array to be passed by reference. This change was necessary in order to make the simulation feasible, since it reduced the total number of cycles by an order of magnitude. The tendency of the 432's Ada compiler to rely exclusively on "call-by-value/result" parameter-passing will be discussed in detail in Section 5.1.4.

The Intel 432/670 development system incorporated a slow, asynchronous memory/bus interconnection that added a significant (but unspecified) delay to every memory reference made by the GDPs, estimated at 6 waitstates (Konrad, Lai, private communication, June 1984). Since it is possible to create faster memory bus designs for the 432, all of the analyses in this paper were done for 0, 3, 6, and 10 waitstates. Six waitstates will be the default for performance comparisons.

5. MAJOR CYCLE SINKS IN THE 432

To establish the low-level performance effects of the 432's object orientation we must first account for other unusual aspects of its architecture and implementation that influence its performance. This section presents measurements of several such aspects. Later, several small incremental enhancements will be

Table V. Baseline Instruction Stream Statistics

Benchmark	Instructions executed	Instructions executed per fetch	Average instruction length in bits
Acker	150489	0.76	42.2
Sieve	1549095	0.69	46.2
CFA5	385005	0.68	47.3
CFA5R	556006	0.73	44.1
CFA10	602003	0.76	41.9
Dhrystone	500	0.82	39.1

Table VI. Total Baseline Cycles Executed with Standard 432 and Compiler

	Total cycles executed			
Benchmark	0WS	3WS	6WS	10WS
Acker	292355847	343503120	394650393	462846757
Sieve	5076556	6340021	7603486	9288106
CFA5	23857599	29387022	34916445	42289009
CFA5R	41972903	51555398	61137893	73914553
CFA10	33886612	41345050	48803488	58748072
Dhrystone	49980	59508	69036	81740

proposed and their performance effects estimated. These enhancements include changes to the 432 system (architecture, implementation, or compiler), which could plausibly have been made to the original 432 assuming (at most) slightly better technology or different implementation decisions. The resulting system will be used to investigate the overhead of object orientation.

After running benchmarks on the 432, we performed some architectural analyses to find out where the performance losses were in the 432. Our evaluation of the 432 yielded the following list of possible problems:

—The μInstruction Bus between the Instruction Decoder chip and the Execution Unit chip is only 16 bits wide and must carry both control and microinstruction data.

—The Execution Unit chip has only a single multiplexed 16-bit bus with which to transfer data, address, and control information to and from memory (the "ACD" bus).

—There are no user-programmable registers on chip that the compiler could use for temporary storage or intermediate results.

—The *entered_environment* "levels" are not on chip, so whenever levels must be checked, memory references are generated.

—There are only three *entered_environments*.

—The garbage collector cannot be turned off—each copy_AD instruction must mark the gray bit at a cost of 9 clock cycles.

—The instructions are bit-aligned, so decoding is complex.

—There are no literals or embedded data.

—Only the top 16 bits of the stack are on-chip, so stack references such as "Push an integer" cause memory operations.

—The Ada compiler has some problems:

(1) It does no common subexpression analysis, so many redundant array address calculations are performed.

(2) It does no code flow analysis, so it takes the brute force approach to handling the *entered_environments*: at each access to an object, it re-enters the environment.

(3) It uses "call by value/result" parameter-passing reference semantics, even where Ada would allow call-by-reference. This necessitates moving every **in out** parameter before and after every procedure call.

(4) Only protected procedure calls and returns are used, even though the instruction set contains a *branch-and-link* mechanism that could be used on nonrecursive intramodule calls.

—The caches have these limitations:

(1) The data-segment cache is only five entries deep, and is flushed upon procedure *calls*, *returns*, and *enters*.

(2) The object-table cache is only four entries deep.

(3) There is no cache of ADs that survives protected procedure calls, which would allow the Data Segment cache to be more quickly refilled.

The microsimulator was used to explore the effects of each of the above problems on the 432's low-level performance.

5.1 The 432 Ada Compiler

Compilers for the Ada programming language are notoriously difficult to write [31]. Language features such as up-level addressing, *rendezvous*, multitasking, packages, and separate compilation facilities are complex to implement. Perhaps these difficulties diverted the 432's compiler writers from object-code performance issues, since the 432 Ada compiler generates very inefficient code. Even on the low-level benchmarks used in this paper, which avoid the complex features of the Ada language, a large fraction of the generated instructions are unnecessary. On the Release 2.0 measurements reported in [13], for example, unnecessary instructions waste over 50 percent of the execution time for the Puzzle benchmark. The Ada compiler for Release 3.0 432 is considerably better, but still profligate in its management of the architecture. We now turn our attention to the performance effects of some specific problems with the compiler.

5.1.1 *Mismanaging the Entered_Environments.* Probably the worst characteristic of the 432's Ada compiler is its management of *entered_environments*. As discussed in Section 3.5, *enter* instructions are executed in order to make some new capability list directly accessible. For instance, data values passed as parameters of a procedure call are immediately available to the called procedure, since an AD to the Message object in which they are held is precreated in the called context. But when structures, arrays, and other objects are passed, an AD for them must be placed in the access portion of the Message object, and an

Table VII. Percentage of Total Benchmark Cycles Spent on *enter_envs* and the Resulting DS_Cache Misses

Benchmark	% Total cycles executing enters	% Total cycles refilling DS_Cache	% Total enters + DS_Cache
Acker	0.0	0.0	0.0
Sieve	0.0	0.0	0.0
CFA5	14.1	4.9	19.0
CFA5R	7.7	2.6	10.3
CFA10	17.0	5.8	22.8
Dhrystone	14.6	3.6	18.2

enter_env instruction must then be executed in order to use the Message object's ADs to access the passed objects.

Managing the entered_environments is, from a compiler's point of view, essentially equivalent to the classic general data-register allocation problem. In both cases, the compiler must schedule usage of a finite set of resources to optimize performance. The low-level benchmarks reveal that the 432 Ada compiler does a very poor job of managing these entered environments. It is clear that no flow analysis is being performed, and even some obvious heuristics that could improve performance are not employed.

Table VII shows the percentage of clock cycles attributable directly to the execution of *enter_envs*, and also shows the additional cycles lost due to related Data Segment (DS) cache misses. The "combined" column shows how large a percentage of all cycles executed in the benchmark went to performing an *enter_env* or re-filling the DS_cache entries for an environment. That column can therefore be regarded as an upper bound on the system speedup attainable by better environment management.

The CFA5R benchmark illustrates how poor management of entered environments cripples 432 performance. This benchmark spends a significant amount of its total running time inside a tight inner loop (greater than 50 percent of the total, in the 432's case). The inner loop consists of a single source statement that executes 14,000 times during the benchmark. The procedure containing this loop is called 1,000 times. The compiler places an *enter_env* inside the loop. The same environment is *enter*ed on each iteration. The *enter_env* could be moved outside the loop without ill effects, since the loop does not need access to more than the three environments available simultaneously. Moving the *enter* outside the loop in this example will save approximately 5.4 percent of the total cycles executed due to executing 13,000 fewer *enters*, plus the savings due to 14,000 fewer DS_cache misses (1.8 percent).

While this is a worst-case example, there are several other cases where flow analysis could have removed unnecessary *enter_envs*. Compiler technology has reached the point where we should expect it to be routinely performed in production-quality compilers.

Table VIII shows the total cycles used by each benchmark after the code is adjusted for more efficient management of the entered_environments. We will use this set of results as the basis for the common subexpression analysis in the

Table VIII. Total Cycles Executed per Benchmark, Adjusted for Better Environment Management

Benchmark	Total cycles executed originally	Total cycles executed with improved envs	% Improvement
Acker	394650393	same	0.0
Sieve	7603486	same	0.0
CFA5	32168445	25490445	18.2
CFA5R	61137893	55059893	10.0
CFA10	48803488	41162488	15.7
Dhrystone	69036	60502	12.4

next section. The percentage improvement is shown in this table, but subsequent tables will not show percentages so that the cycles saved can be combined under various assumptions (e.g., better compiler and instruction stream literals, or eight general registers with wider buses).

5.1.2 *Common Subexpression Analysis. Common subexpression* analysis is a common compiler optimization technique that allows the results of some intermediate calculations to be reused rather than recomputed. For instance, when accessing an array, the address of the array element must be computed as a function of the base address of the array, the size of each array element, and the method of packing the array elements into physical memory. This calculation normally entails a multiplication and a type conversion. If the same array element is specified on both sides of an assignment operation, then reusing the address will save one multiplication and one conversion. Note that this optimization can be done even without local registers, since storing and retrieving such temporaries from memory may still be faster than repeatedly recomputing a sequence of addressing calculations.

The 432 has an addressing mode that permits a single macroinstruction to express an Ada source line such as

array[*x*] := *array*[*x*] + 30;

It can only be used for one-dimensional arrays, however. Arrays of two or more dimensions require explicit address calculations at the macroinstruction level. For these more complex addressing modes, the 432 Ada compiler provides no optimizations, even for the trivial case of identical array elements on both sides of an assignment. The compiler also fails to reuse addresses and temporary data across several macroinstructions.

The lack of common subexpression optimization is a problem in the CFA10, CFA5, CFA5R, and Puzzle benchmarks, since those programs involve extensive access to arrays or structures. In CFA5, for example, removal of the three redundant instructions would save 150 clock cycles out of the loop total of 1,110 cycles. Since this loop accounts for 56 percent of all cycles used in the benchmark, overall elapsed time would be decreased by (150/1100)(0.56)(100) = 7.6 percent (compared to the baseline measurements).

Hand-optimizing the 432 assembly code by eliminating common subexpressions produced the cycle savings shown in Table IX.

Table IX. Cycles Saved Due to Hand-Optimized 432 Assembler Code

Benchmark	Cycles saved
Acker	0
Sieve	0
CFA5	4044000
CFA5R	4560000
CFA10	3696000
Dhrystone	457

5.1.3 *Protected Procedure Calls.* The Release 3.0 Ada compiler and 432 system treat every call identically; calls on critical system routines look no different from the standpoint of execution efficiency than does a call to a private function within a user's procedure. While this strategy does preserve the call graph of the object model with admirable consistency, keeping this generality often seems pointless, especially when the compiler has full control over both the calling and called procedures simultaneously [16, 23]. Procedures that are private to a given package may be candidates for an optimization that relaxes the normal checks and constraints of a protected call.

Suppose that a compromise strategy were used to combat this procedure call overhead, with the philosophy that the security and correctness of *intra*-module calls are the province of the compiler. Protecting *inter*-module calls would remain the responsibility of the architecture. With this plan, nonrecursive intra-module calls and returns could be replaced with a simple "branch and adjust stack" (for new local variables) sequence, with the instruction segments made coresident in the same instruction object.[5] (Flow analysis would be required to determine the maximum depth of nested calls.) This would make this procedure call no more costly than a call on a conventional architecture.

The 432 instruction set does in fact provide instructions with the required semantics for such operations (*branch_intersegment*, *branch_intersegment_and_link*). However in none of the benchmarks reported here, nor in any of the dozens of other programs run during this research, were these instructions ever generated by the Ada compiler.

Dhrystone provides the best example of what kind of savings are possible with unprotected call mechanisms. Seven of its 15 procedure calls are intra-module, and eight of them inter-module. If these intra-module calls and their returns had been unprotected, then a total of 12,405 clock cycles would have been saved out of the baseline total of 60,502 cycles, for a performance improvement of 20.5 percent.

Table X shows how performance would improve if the "compiler-protected intra-module call" compromise suggested were available on the 432.

5.1.4 *Parameters Passed by Value/Result.* The Ada language allows the programmer to specify the formal parameters of a procedure call as **in**, **out**, or **in**

[5] In the case of a recursive call, the depth of recursion cannot in general be known at compile time, so the required context-segment size cannot be determined. Hence a new context segment is needed for each procedure call.

Table X. Summary of the Performance Improvements Possible if Intramodule Calls Were Protected by the Compiler

Benchmark	% Intramodule calls and returns	% Total cycles spent in intramod. calls	% Improvement possible over baseline
Acker	all recursive	75.0	0.0
Sieve	no calls	0.0	0.0
CFA5	100.0	7.4	7.4
CFA5R	100.0	3.6	3.6
CFA10	100.0	4.3	4.3
Dhrystone	47.0	44.7	20.5

out. **In** parameters are to be passed by value to the called routine, which cannot change the actual value. **Out** parameters can be assigned by the called routine so that the called routine can transfer data back to the caller. **In out** parameters allow both calling and called routines to read or assign values to the actual parameters.

Implementing the **in out** parameter passing convention is normally done with "call-by-reference" or "call-by-value/result" semantics. As long as the called routine terminates normally and has no side-effects, either method will achieve the same result [31]. However for large parameters the efficiency of these two methods is quite different, since "call-by-reference" requires only pointers to be passed, while "call-by-value/result" necessitates copying of the parameters before and after the call.

The 432 Ada compiler passes all **in out** parameters by value/result. (**In** parameters are always passed by value, and **out** parameters by result.) This unnecessary copying of data can be circumvented at the Ada source-code level by declaring pointers (Ada's "access types") to the data structures, and then passing the pointers instead of the structures. This was done in the Dhrystone benchmark.

Table XI summarizes the clock cycles spent in each benchmark moving **in out** parameters unnecessarily. The Dhrystone benchmark requires an order-of-magnitude more time to complete when the default "call-by-value/result" semantics are employed (total cycles for this benchmark were listed as 60,502 in Table VIII). One of the calls in Dhrystone passes two large arrays, one with 50 integers, and the other with 2,500. Copying these arrays both before and after the procedure call takes nearly ten times as long as the rest of the benchmark!

5.2 Lack of Local Data Registers

The 432 is a pure memory-to-memory architecture, with the single exception of 16 bits of the top-of-stack. There were three major reasons for this design approach. For its time of introduction, the 432 Execution Unit was a very large chip, and it was felt that local data registers could not be included without trading away essential features such as base/length registers or substantial amounts of microcode. Another reason was the speedup in process-swap time when on-chip state is minimized. Finally, there was felt to be a conceptual unity

Table XI. Clock Cycles Wasted by the 432 Ada Compiler's Use of "Call by Value/Result" Semantics

Benchmark	Cycles moving *in out* params	Cycles to pass ptrs	Cycles saved by "call-by-ref"
Acker	0	na	0
Sieve	0	na	0
CFA5	1034000	128000	906000
CFA5R	9563000	128000	9435000
CFA10	5844000	128000	5716000
Dhrystone	630584	256	630328

and simplicity afforded by a memory-to-memory machine, especially a shared-memory multiprocessor such as the 432. Since the 432 architecture was expected to outlive several generations of implementation technology, some loss of performance was felt to be acceptable.

However, the performance penalty paid for such a design can be large. For variables that are frequently referenced, such as loop counters, and especially for loop counters used within the loop (array indices, for instance), a very large percentage of otherwise redundant memory data transfers can be avoided if on-chip data registers are available.

In order to gauge the effects that this design decision had on the 432's performance on these benchmarks, the benchmarks were simulated using a 432 architecture enhanced with a set of general registers, assumed to be accessible in one clock cycle. The 432 requires approximately 15 clock cycles to read 32 bits from memory (9 internal clock cycles plus 6 bus/memory waitstates). If the 432 had incorporated eight 32-bit data/address registers, its performance on the benchmarks would have improved substantially, as shown in Table XII.

Had the 432 included some general-purpose registers, the object-code size would have decreased as well, since fewer bits are required to reference a register than to reference a memory location within an object. This in turn improves the overall execution time since fewer overall instruction fetches are required. The microsimulator log files show instruction fetch cycles, so the number of cycles saved due to shorter instructions can be estimated as follows. Scalar data references normally require 19 bits of the instruction stream: 2 bits for the data reference mode, 2 bits for the access selector mode, 1 bit determining the displacement length, 7 bits of access selection, and 7 bits of displacement (for a short displacement). We assume here that specification of a register would take 4 bits: 1 to decide register or memory, and 3 to select the register. As a result, we save $19 - 4 = 15$ bits in the instruction stream for each memory reference eliminated.

Using registers to pass variables can save many more cycles than the simpler scheme, which was modeled here, since only those register values that must be saved (via a memory operation) are saved, while those containing information useful to both calling and called routines can be left untouched. In those cases where the compiler has control over both the calling and the called routines, this is straightforward. Calls to procedures written in other languages, or compiled by different versions of the compiler, present different challenges that have not

Table XII. Cycle Savings Possible if Eight 32-Bit Data Registers Had Been Included in the 432

Benchmark	Memory access cycles saved	Instruction stream cycles saved
Acker	0	0
Sieve	2681926	$\sim 1 \times 10^6$
CFA5	3555000	$\sim 1.6 \times 10^6$
CFA5R	3261000	$\sim 1.6 \times 10^6$
CFA10	5715000	$\sim 2.7 \times 10^6$
Dhrystone	1305	$\sim 6 \times 10^2$

yet been resolved in the literature. Performance improvements possible for register-based parameter-passing are not further considered here.

An issue not addressed in this paper is the 432's multiprocessing support. Providing local registers that are under the control of the compiler (avoiding the problems associated with variables being shared by more than one process) would also improve system performance by decreasing the contention for memory among the GDPs in a 432 system.

5.3 Bit-Aligned Instructions

Because the size of object code can have a major influence on the performance of an architecture, designers attempt to minimize instruction width. From an information-theoretic point of view, instructions could be smaller if they did not have to occupy an even number of bytes or words. On the other hand, these instructions might begin at any bit within a byte or word. The 432 is the second important architecture to exploit such bit-aligned instructions; the Burroughs B1700 [28] was the first. Unfortunately the expected object-code size savings were not realized in the 432. Hansen et al. [13] reported that the 432 object code was not much smaller than that of the VAX or the Motorola 68000. The reason is the disproportionately large number of memory references that are made by the 432 due to its lack of data registers.

When a bit-aligned instruction is fetched, the first bit retrieved from memory is usually not the first bit of the instruction. The machine must perform two separate memory accesses, and then combine two bit strings via barrel shifting in order to reconstitute the instruction. The 432's Instruction Decoder is implemented in such a way that it can usually reconstitute and decode the instruction bitstream in parallel with the program execution occurring in the Execution Unit. 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.

For the benchmarks used in this paper, the number of cycles lost to Instruction Decoder stall can be quantified, since they are marked in the log files (see Table XIII).

5.4 Lack of Literals or Embedded Data

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

Table XIII. Cycles Lost to Instruction Decoder Stall

Benchmark	Cycles spent waiting on instr decoder
Acker	5936775
Sieve	682846
CFA5	743011
CFA5R	1171016
CFA10	1692005
Dhrystone	1288

constants. This conclusion was almost certainly in error, but it enabled the Instruction Decoder and Execution Unit to be built as separate chips, and thus facilitated building such a complex system on silicon (George Cox, private communication, January 1985). 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. As a consequence, when it became clear that zero and one would not suffice it was too late to rectify the mistake.

The lack of immediate data impacts performance in several ways. A reference to data in the local constants object usually occupies 19 bits, much more than most of the constants themselves. Hennessy et al. [15] report that for a set of Pascal programs, a 4-bit constant is sufficient for approximately 70 percent of all data constants, and 8-bit constant suffices for 95 percent. Besides the wasted instruction stream bits, the additional memory references required to fetch the constants are expensive.

Application-code data constants are not the only constants used by the 432. In calculating array offsets, and while manipulating system-defined objects, the 432 microcode frequently requires constants such as 4 or 8 with which to calculate addresses of various ADs or ODs. These constants are currently kept in the Global Constants Object. Fetching these constants degrades performance just like fetching local constants.

The 432 microcode uses constants fetched from the local constants object during procedure calls. If the local constants object is not qualified at the time of the call, then a data-segment (DS) cache miss will occur, adding approximately 77 clock cycles to the procedure-call total (see Section 5.6.1 for more details on the DS_Cache). If instruction-stream literals were available these expensive qualifications would not occur. For the Dhrystone benchmark, 1,078 clock cycles are lost to DS_Cache miss processing on the local constants object, or 1.8 percent of the baseline total.

Another subtle performance degradation due to accessing constants objects is the increase in size of the context segments. Each context must have ADs to both local and global constants objects, and the overhead of maintaining and using these ADs is paid in extra cycles for procedure calls and returns.

Table XIV shows the performance speedups possible if the 432 instruction set architecture had included immediate data. We assume here that a 16-bit data path from the instruction decoder to the execution unit is available, and that a transfer takes 2 cycles.

Table XIV. Cycles Saved with Instruction Stream Literals

Benchmark	Cycles spent referencing data constants	Cycles spent referencing addressing constants	Totals
Acker	—	2927961	2927961
Sieve	413532	34461	447993
CFA5	720000	250944	970944
CFA5R	720000	260885	980885
CFA10	876000	404613	1280613
Dhrystone	636	1550	2186

5.5 Three Entered Environments

The 432 provides each context with four addressing spaces: the current context, and three "entered environments." (The operation of the *enter_environment* instruction was discussed in Sections 3.5 and 5.1.1.) These environments are used to provide fast access to a "working set" of objects. If there are too few *entered_environments* then a working set may not be achieved, and a substantial performance loss may be incurred in repeated changes of environments.

Because each *entered_environment* requires a substantial amount of chip resources, only a few *environments* can be provided. The 432 provides three, the minimum reasonable: triadic instructions such as $a := b + c$ can generate references to three separate objects. The Ada modules containing a, b, and c would have to have been "entered" prior to execution of this addition, of course.

Even with good management of environments, there are addressing requirements that make use of more than three environments from a given procedure's context. Traversing a linked list of structures, for instance, may require that a new *enter* be executed per node of the list.

One would expect this to be a problem for large programming systems, where many modules exist and call each other in patterns that are not completely determinable at compile time. Of the benchmarks used here, only Dhrystone would be speeded up by the availability of additional entered environments. The other five benchmarks would actually run slower, because the procedure call and return time is partly a function of the number of environments.

To get a more meaningful measurement of how large Ada programming systems would use the entered environments we have measured the module connectivity of three such systems. The first is CMU's "HG" mail system, developed by Michael Horowitz and David Nichols, consisting of approximately 33K lines of Ada source code. The second is the Adix kernel, developed at the University of California at Santa Barbara under the direction of John Bruno and Laurian Chirica, consisting of approximately 20K lines of Ada source code. The third is a relational database program developed at Hughes Aircraft under the Distributed Software Architecture Project by Paul Rabow and his colleagues, comprising approximately 5K lines of source code. The Adix kernel and the Hughes programs were written with the 432 as their intended execution engine.

For each of these three programs, Figure 5 shows the fraction of Ada packages that reference a particular number of other packages. This graph shows that all three Ada programs exhibit roughly similar organizations in terms of their

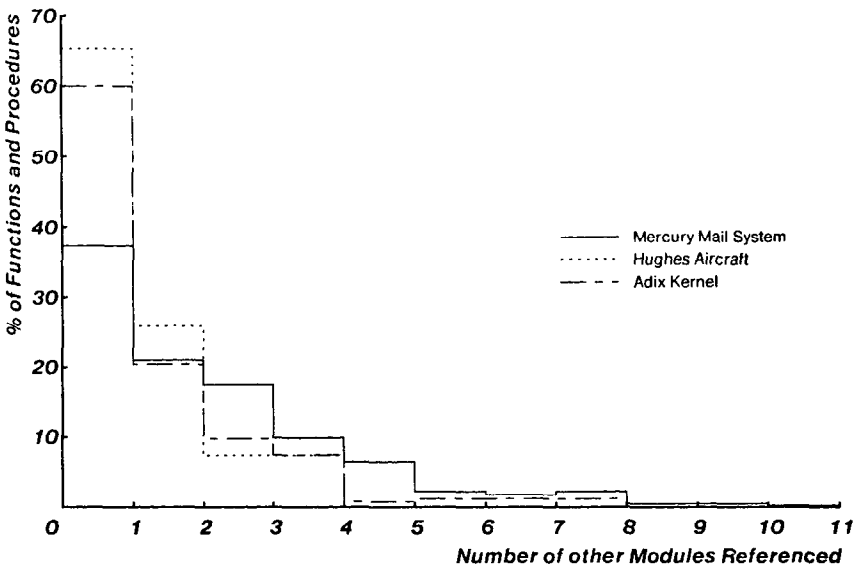

Fig. 5. Large Ada system module interconnectivity.

intermodule connectivity. Of the three programs, the Mercury Mail System is believed to be the most reliable, since it is the largest program, and it is the only one of the three that is in daily use. Figure 5 shows that 38 percent of all Mercury routines (functions + procedures) make no routine invocations to other packages, 22 percent call only one other package, 18 percent call two other packages, and 10 percent call three other packages. The remaining 12 percent of all routines call more than three other packages.

These figures cannot be used to extrapolate the number of modules that could profit from more than three entered environments. For example, in the best case, if a sequence of calls to routines in four separate modules were to be invoked and only one environment were available on-chip, duplicate enters can still be avoided if the call pattern is **A, A, A, B, B, B, C, C, C** .... The worst case occurs when more modules are referenced than there are environment slots on-chip. If the call pattern is **A, B, C, D, A, B, C, D** ... then a new enter must be performed upon each new call, independent of the number of environments. Thus a module that calls more than three modules may not benefit from having more than three environments.[6]

This analysis seems to indicate that, at least for the programs studied, the benefits of adding more environments would be slight. Also, the meager improvement might be offset by increased procedure Call/Return time due to the additional state associated with the extra environments. During a Call, the environments are cleared for the new context, but Returns must restore the environment values as they existed just prior to the Call. Thus increasing

[6] Environments may also have to be used to permit access to data residing in other modules. However, in well modularized code, data shared directly without the benefits of an intervening type manager are extremely rare.

the number of environments makes the Return operation slower. Analysis of the 432's Return microcode shows that 144 clock cycles are required for each environment that must be restored. If all three environments were in use prior to the procedure Call, then the Return will execute approximately 430 clock cycles out of a total of 850 cycles in restoring the environments. If the Return instruction had to restore 10 environments, the total time to execute the Return would more than double to 230 μS.

5.6 Caches

Two "addressing" caches were included in the 432: a set of four base/length registers pointing to the most recently used object tables (the object-table cache), and a set of five base/length registers pointing to the most recently referenced data objects (the data-segment cache). A third cache for ADs was considered for the 432, but not provided due to implementation constraints. This section explores the effects on performance of the cache sizes (4, 5, and 0, respectively), cache management, and system usage patterns of the caches.

The address caches (OT and DS) are crucial to throughput. In a conventional architecture, the ratio of memory access time to cache-hit access time may be from 2:1 to 5:1 [3]. In the 432, there are no data or instruction caches, and just to generate an address efficiently assumes a high hit rate in the DS_Cache (and if that misses, the OT_Cache). Assuming a word access to memory is underway, a DS_Cache hit will cause the memory transaction to be 12 cycles long, including waitstates. If the DS_Cache misses, but the OT_Cache hits, then the transaction takes 89 clock cycles. When both caches miss, the transaction requires 179 clock cycles. Thus the cache-miss access ratios for the 432 are between 7:1 and 16:1.

5.6.1 *The Data Segment Cache.* In issuing an operand reference, the 432 microcode tests whether the translated virtual address maps into an object for which a base/length register pair is available on-chip. If it does, the memory reference proceeds normally, with the bus delays and memory waitstates described elsewhere in this paper. If no match is found, the microprogram raises an exception and calls the Data_Segment_Cache_Fault handler. This microcode finds the least recently used entry in the Data Segment (DS) Cache and flushes that entry. The microcode then "qualifies" the referenced object by testing its type, length, and rights against the type of access being attempted by the program and the rights the program has to that object. The flushed base/length registers are then refilled with the base address and length of the new data object, and the read/write rights associated with that object are stored on-chip. From that point the memory reference causing the DS_cache miss is retried. A simplified version of the 432 address caches are shown in Figure 6.

DS_Cache entries include the base address of the data segment, the length of the segment, read and write rights, and the "altered" bit. The length is used for bounds checking every access to the segment. The read/write access rights are also checked upon every memory access.

DS_Cache miss processing costs 77 clock cycles at 6 waitstates per memory access (148 clock cycles when the reference is to a refinement). A substantial fraction of the baseline cycles executed (Table VI) are due to DS_cache miss

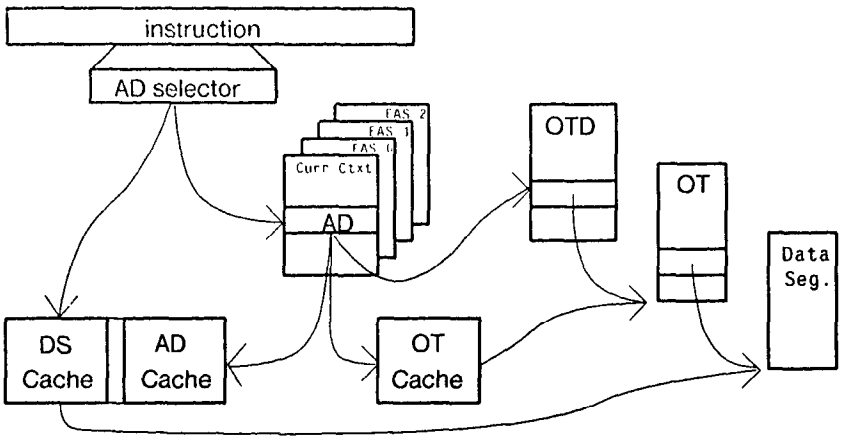

Fig. 6. The 432 addressing caches.

processing, but are not really necessary. This is because execution of *enter_environments* causes whatever DS_cache entries were associated with that environment to be flushed, and as Section 5.1.1 demonstrated, a large proportion of all *enters* executed are redundant.

The DS_Cache entries are flushed when the *entered_environment* to which they correspond is altered. This is done because the cache is associatively searched using a tag composed of the *entered_environment* number concatenated with the AD_Selector. Since procedure calls and returns cause changes to all *entered_environments*, the DS_Cache is always empty immediately following any context switch.

Due to the empty DS_Cache, the first reference to a passed parameter within the called context causes a DS_Cache miss. In serving this miss, the microcode will discover that the called routine's AD to the Message object is actually a refinement and will then proceed to traverse this refinement to get the base and length of that portion of the Message object to which the called routine is entitled.

By making the Message object a refinement of the calling context, the calling context saves the DS_Cache miss, which would otherwise be associated with accessing a separate data object. For the situation where a calling routine invokes only one procedure (with parameters) this scheme saves 30 cycles.[7] The economics of this parameter-passing scheme will be discussed further in Section 5.6.3, since the addition of a cache which remains intact across procedure calls makes other options more attractive.

For the benchmarks used in this paper, the DS_Cache is large enough—the "least-recently-used" management policy is never tested here. However, managing the DS_Cache is responsible for a large percentage of the clock cycles used in Dhrystone, due to the large number of *calls*, *returns*, and *enters* executed.

[7] The difference between the price for traversing a refinement, 148 cycles, and the cost for both the caller and called routines to each take a DS__Cache miss on their first access to the Message object: 89 + 89 = 178 cycles.

To find out how well the DS_Cache performs in the 432, the Dhrystone benchmark log files were analyzed to determine the reason for every DS_Cache miss that occurred. Six reasons were found:

(1) *First Access*: A DS_Cache miss occurred because this is the first time that the object is being referenced.
(2) *Local Constants Object*: Data needed by the microcode during execution of a procedure call resided in the Local Constants Object, which was not qualified at the time.
(3) *Message Object Overflow*: The parameters to be passed did not fit into the Message Object, and were placed into a separate Overflow object, which required separate qualification.
(4) *Environment Mismanagement*: Poorly placed enter_environments invalidated the DS_Cache slot.
(5) *Environment Reuse*: An environment was reused, invalidating the corresponding DS_Cache entries as a side-effect.
(6) *Call Wipe*: Upon returning from a procedure call, the environments are restored but the DS_Cache is empty.

Table XV shows the frequency of occurrence of these six categories. Notice that overflow of the DS_Cache itself is not one of the reasons for DS_Cache misses (i.e., the cache is not too small). The cache is large enough because local variables do not require a DS_Cache entry in order to be accessible; they reside in the context data part, which is qualified as part of the current context. The distribution of operand localities in Dhrystone is 48.5 percent locals, 7.9 percent globals, 18.7 percent parameters, 2.1 percent function results, and 22.8 percent constants [35]. Of these operands, only the globals, parameters, and constants (approximately half of all operands) require assistance from the DS_Cache to be made accessible.

By examining the simulator log files from the Dhrystone execution, it is possible to estimate the DS_Cache hit ratio. The total number of DS_Cache hits was 403, with 40 misses. Thus the DS_Cache hit ratio is

$$F_{dh} = 403/(403 + 40) = 0.9097 \qquad \text{(91 percent)}$$

Ackermann's function shows the DS_Cache scheme at its worst. *Acker* consists almost solely of recursive procedure calls and returns, along with some trivial additions and subtractions and some conditional branching. The cost to pass two integers in each recursive call is 148 clock cycles, a very high price to pay for access to the passed parameters. Analysis of this benchmark shows that if local data registers were available to the 432, and the compiler were adept at using them for parameter-passing between recursive contexts, *Acker* would speed up by over 20 percent due to the lack of DS_Cache misses and faster arithmetic in the simple operators category.

5.6.2 *The Object Table Cache.* This paper has concentrated on low-level compute-bound benchmarks so that the primitive operations of a machine with a significant object-based overhead can be investigated. For such benchmarks, however, the size of the Object Table Cache (OT_Cache) is irrelevant as long as

Table XV. Reasons for Misses in the DS_Cache

Reason for DS_cache miss	Number and percentage
First access to object	15 (37.5%)
Local constants object	13 (32.5%)
Call wipe	4 (10.0%)
Env. mismanagement	3 (7.5%)
Env. re-cycling	3 (7.5%)
MSG. object overflow	2 (5.0%)

there is at least one slot in the cache. For all of the benchmarks used here, including Dhrystone, the compiler and linker allocated every application-level object out of the same object table. As a result, these benchmarks took only one OT_Cache miss early and hit on all subsequent attempts.

The 432 was designed to support "programming-in-the-large," so the fact that these benchmarks only required one object table is not compelling evidence that the OT_Cache could safely have been made only 1 slot deep. However, both the Adix and Hughes programs allocated all code segments from one object table as well, so they do not provide raw data for an investigation of OT cache size. To test the effect of OT_Cache size, a mathematical model was developed, as described in [5]. It assumed locality of reference to the object table; i.e., that the next object referenced had a certain, fairly high probability of being in the same object table as the previous one. Given the observed number of DS_Cache misses for the Dhrystone benchmark, the model suggests that there would be little benefit from increasing the size of the OT_Cache beyond its current 4 entries.

5.6.3 *The Hypothetical AD Cache.* As the 432 manages the DS_Cache, it is often loaded with an entry for some data object, then cleared as a side-effect of some operation, and finally reloaded with the same information. For instance, when both the calling and called routines must access the same data object (e.g., data that are within the scope of both routines, or data for which a pointer was passed to the called routine) the DS_Cache is first qualified by the caller. A protected procedure call must ensure that the called routine cannot access any objects for which it has no AD, so the DS_Cache is cleared during the procedure call. Consequently, in order to access the data, the called routine must requalify the data object.

Without violating any of the fundamental principles of object orientation, it is possible to place a new address cache into this addressing mechanism. This proposed new cache would fit between the OT_Cache and the DS_Cache, matching on access descriptors in the event of a DS_Cache miss. Matching at the AD stage provides addressing information that is early enough in the addressing chain so that a hit would still provide a relatively quick reference. Most important, though, the AD_Cache would not be affected by context changes or by the vagaries of *enter_environments*. Hence one would expect the usual intercontext types of data locality to provide a high hit rate in this new cache, significantly improving overall performance.

The best way to view the operation of this new cache is as a part of the DS_Cache (this is why these two caches were shown connected in Figure 6.)

Figure 7 shows a block diagram of this combined cache. The columns in the cache are used as follows. A single cache entry takes up a horizontal slot in the diagram. The rights (Read or Write) information goes into the right-most column. A bit indicating the current validity of the Access Selector in column three is stored in column two. Column four contains the 24 bits of the AD, and column five holds a tag that is used to reference the base/length pairs in the DS_Cache. Column six holds the matching tag value, and columns seven and eight contain the base/length information for the referenced data object. The "tag" feature is not strictly necessary. However, without this additional mechanism, separate base/length information would have to be kept for the cases where multiple ADs (with different rights, for example) are being used to refer to a single object. Since the base and length information is the same regardless of the accessing rights any particular AD has to that object, this feature is expected to (in effect) make the cache larger.

The following algorithm shows how this combined cache would work:

```
—first, attempt to match the DS_Cache as usual.
begin
  if (Access_Selector matches any AS slot)
  and (that slot is valid)
    then begin
      use slot's tag to get base and length;
      generate operand reference;
      end;
  else begin   —DS_Cache missed. Try AD_Cache.
    fetch AD to object being referenced;
    if (AD matches any AD slot)
    and (AS is not valid)
      then begin
        fill in slot rights from fetched AD;
        fill in AS slot from instruction AS;
        set AS slot to valid;
        use slot tag to get base and length;
        generate operand reference;
        end:
      else begin   —AD_Cache missed too.
        do normal OT_Cache processing;
        do LRU replacement on DS/AD cache;
        generate operand reference;
        end;
    end;
end
```

If the AD_Cache is provided, the economics of parameter-passing change substantially. The two mechanisms under consideration for the 432 were

(1) Parameters are passed as a refinement of the calling context, saving a DS_Cache miss by the caller, but causing the called routine to traverse this refinement (148 clock cycles).

(2) Parameters are always placed into a separate object, with both caller and called objects taking a DS_Cache miss on first access ($2 \times 89 = 178$ clock cycles).

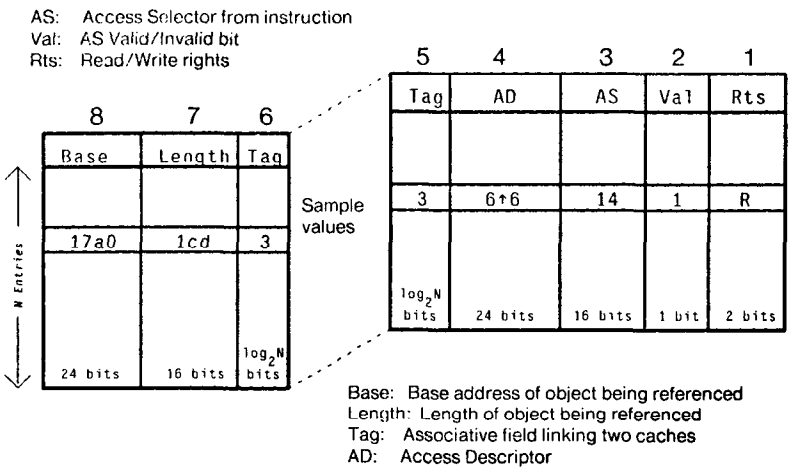

Fig. 7. Proposed DS/AD cache organization (sample values).

With an AD_Cache, it may be more advantageous to place parameters into a separate object. If a routine calls two or more other routines, and the AD_Cache is large enough, the initial DS/AD_Cache miss processing by the caller will allow all called routines to reference these data with a DS_Cache miss but an AD_Cache hit. The caller will also only experience a DS_Cache miss and an AD_Cache hit after the return from the first called routine. The next and subsequent calls will hit the AD_Cache. The total difference in cycles for this scheme is 89 (caller 0) + 30 (called 1) + 30 (caller 0) + 30 (called 2) = 179 cycles. Using refinements, this sequence would require 0 (caller 0) + 148 (called 1) + 0 (caller 0) + 148 (called 2) = 296 cycles.

Providing this AD_Cache on a next-generation 432 would require substantial chip resources. How much would performance be improved by such a cache? Would using these resources in other ways, such as to enlarge the DS_Cache, be more advantageous? We have argued that neither the OT_Cache nor the DS_Cache are too small. The main problem with the DS_Cache is that it rarely stays loaded for long, since *calls*, *returns*, and *enters* all invalidate it.

What kind of a hit rate could we expect from the AD_Cache? To estimate it, we will use Dhrystone, but we will have to make some assumptions. Table XV showed that 32.5 percent of the DS_Cache misses were due to the 432's lack of instruction stream literals. If literals were available, these references would never get to the AD_Cache, because they would not require independent memory accesses. If literals are not available, then only the first reference to the local constants object would miss the AD_Cache, and subsequent references to local constants would hit, driving up the apparent AD_Cache hit rate appreciably.

All DS_Cache misses due to "call wipes" and *environment* recycling would also hit the AD_Cache (if literals are unavailable), accounting for 22 out of the original 40 DS_Cache misses, an AD_Cache hit rate of 0.55. With literals, the total DS_Cache misses would have been 27, with the AD_Cache hitting on 10, a hit rate of 0.37.

The reason that the AD_Cache hit rate is so low is due to the parameter-passing convention of the 432. As discussed earlier, parameters are placed in a Message object, which is actually a refinement of the calling context. This saves a DS_Cache miss on the part of the calling context, but forces the called routines to traverse the refinement to fetch parameters. If the calling context were to arrange for an AD to the Message object to be placed into the AD_Cache as part of the caller's context-qualification, then the called routine would be able to access the passed parameters much more readily. Under this assumption (and assuming instruction-stream literals) the AD_Cache hit rate is $\frac{25}{27}$, or nearly 93 percent.

Estimating the effect of AD_Cache size on the hit rate is difficult for lack of suitable benchmarks—the same reasons that make analysis of the DS_Cache and the OT_Cache difficult. Assuming that the 432 had incorporated instruction stream literals, it appears than an AD_Cache of four entries would allow Dhrystone to execute without having to reuse any AD_Cache slots. But this does not imply that any large Ada program that is represented well by Dhrystone can be expected to not reuse any AD_Cache slots when only four are available. Intuitively, the same locality of reference to data objects that make the DS_Cache hit rate high (in the absence of call perturbations and *enter* manipulations) would apply to the AD_Cache hit rate, except that the AD_Cache would not be cleared by those perturbations, hence the AD_Cache hit rate should be higher.

Lacking extensive statistics on the DS_Cache, we modeled the effects of the AD_Cache for DS_Cache hit rates of between 0.7 and 0.95 [5]. For Dhrystone, an AD_Cache would have given an average operand access time of between 13 and 18 cycles. This compares favorably with the 22-to-25 cycle averages with an OT_Cache only.

Several additional architectural changes would have improved 432 performance: making buses 32 bits wide instead of 16 bits; using a slightly more elaborate signaling convention on the microinstruction bus between the two halves of the CPU; updating reclamation information only while garbage collection is in progress; and caching more than just 16 bits of the top-of-stack on chip. The individual contributions of these items are detailed in [5]; space constraints preclude recounting them here.

6. THE SYNTHETIC 432

This paper has analyzed the low-level and large-system uniprocessor performance of the Intel 432 to prepare for an analysis of its functional migrations. This section presents that analysis, using a "synthetic" 432, and then extends the lessons learned to other machines and systems. The synthetic 432 is a hypothetical microprocessor based on the 432 but altered for improved performance as described in Section 5. These improvements are presented incrementally to accommodate various assumptions about what sets of changes are reasonable to the architecture, compiler, and implementation technology.

We begin with those changes that could have been made in a straightforward manner: items such as compiler shortcomings, lack of immediate data, and the bit-alignment of the instruction stream. We then show how performance would

Table XVI. New Baseline Cycles and Percent Improvement over Original Baseline

Benchmark	Cycles saved	% Original base cycles saved	Synthetic baseline cycles
Acker	8864736	2.2	385785657
Sieve	1130839	14.9	6472647
CFA5	15228248	43.6	19688197
CFA5R	24207058	39.6	36930835
CFA10	21795608	44.7	27007880
Dhrystone	655452	93.7	44168

have been improved had the implementation technology been incrementally better (and if those additional resources had been used as assumed here). With these new performance numbers, comparisons will be made to the baseline 432 and to other processors, so that conclusions about the inherent cost of 432 object orientation can be drawn.

6.1 The Synthetic Baseline 432

Several of the cycle sinks discussed in Section 5 have a significant impact on overall performance, yet are unrelated to architectural complexity, functional migration, or object orientation. As a baseline for further discussion, we will assume that the 432 had been released with the improvements listed below:

—better *enter_environment* management (Section 5.1.1)

—better code optimization by the compiler (Section 5.1.2)

—compiler determination of the appropriate protection mechanism for procedure calls (protected call vs. *branch-and-link*, Section 5.1.3)

—use of the fastest parameter-passing mechanism by the compiler (Section 5.1.4)

—a non-bit-aligned instruction stream (Section 5.3)

—provision for literals in the instruction set (Section 5.4)

Table XVI shows the combined cycles saved when the above assumptions are made. The results are unimpressive for Ackermann's function since that benchmark executes mostly procedure calls and therefore exhibits no speedup with a better compiler. Because *Sieve* does no procedure calls and executes mainly simple instructions and loops, the cycles lost to bit-aligned instruction stream decoding have a large effect. The CFA benchmarks exhibit a 30–40 percent reduction in the total number of cycles needed for execution.

The Dhrystone benchmark shows an enormous reduction of nearly 94 percent, due almost entirely to forcing a single array during a single procedure call to be passed by reference instead of by value/result. The other cycle-sinks become significant only when this array is passed more efficiently; they then constitute approximately 36 percent of the remaining cycles needed to execute Dhrystone.

Since we have asserted here that these changes to the architecture and compiler should have been incorporated in the 432, and would have required little or no additional chip resources, we will assume that the data in Table XVI represent

Table XVII. Relative Contributions of Improvements to Synthetic Baseline Cycles

Benchmark	Enters	OptCode	Pr.Calls	Params	Align	Consts
Acker	0	0	0	0	5936775	2927961
Sieve	0	0	0	0	682846	447993
CFA5	6678000	4044000	1886293	906000	743011	970944
CFA5R	6078000	4560000	1982156	9435000	1171016	980885
CFA10	7641000	3696000	1769987	5716000	1692005	1280613
Dhrystone	8534	457	12403	630584	1288	2186

Table XVIII. Relative Contributions of Improvements over Original Baseline, in Percentages

Benchmark	Enters	OptCode	Pr.Calls	Params	Align	Consts
Acker	0	0	0	0	67	33
Sieve	0	0	0	0	60	40
CFA5	44	27	12	6	5	6
CFA5R	25	19	8	39	5	4
CFA10	35	17	8	26	8	6
Dhrystone	1.2	0.1	1.8	90.0	0.2	0.3

the new baseline benchmark cycles. Additional architectural enhancements and performance comparisons will be conducted using this table as the reference.

The implications of Table XVI must be clearly understood. This table shows that from 35–45 percent of the 432's total benchmark execution cycles are wasted. These cycles are not spent in pursuit of object orientation; they are not the inevitable fallout of a complex instruction set; and they do not reflect the alleged inefficiency of a microcoded processor. We assert that these cycles are consumed because of suboptimal design decisions or outright errors, and that such errors could have been committed on any new system design, whether object-based or not.

The relative contributions of each of the improvements itemized above will be of interest later, so they are shown individually in Table XVII and (as percentages) in Table XVIII.

Figure 8 graphically depicts the relative contributions of each of the six categories. This figure is arranged such that the fraction of total wasted cycles due to each source is shown by the length of the corresponding bar in the bar chart. For example, *Sieve* wastes 15 percent of its total baseline cycles (see the box in the lower right), and of those 15 percent, instruction-set-alignment "contributes" 70 percent and lack-of-literals is responsible for approximately 30 percent. To make the figure less cluttered, the CFA benchmarks are represented here by CFA5. Figure 9 shows the relative importance of each cycle sink by benchmark.

Since every category in Table XVIII and Figure 8 contributes substantially to the speedup of at least one of the benchmarks, the data suggests that each of these improvements is significant and should have been incorporated into the 432 originally.

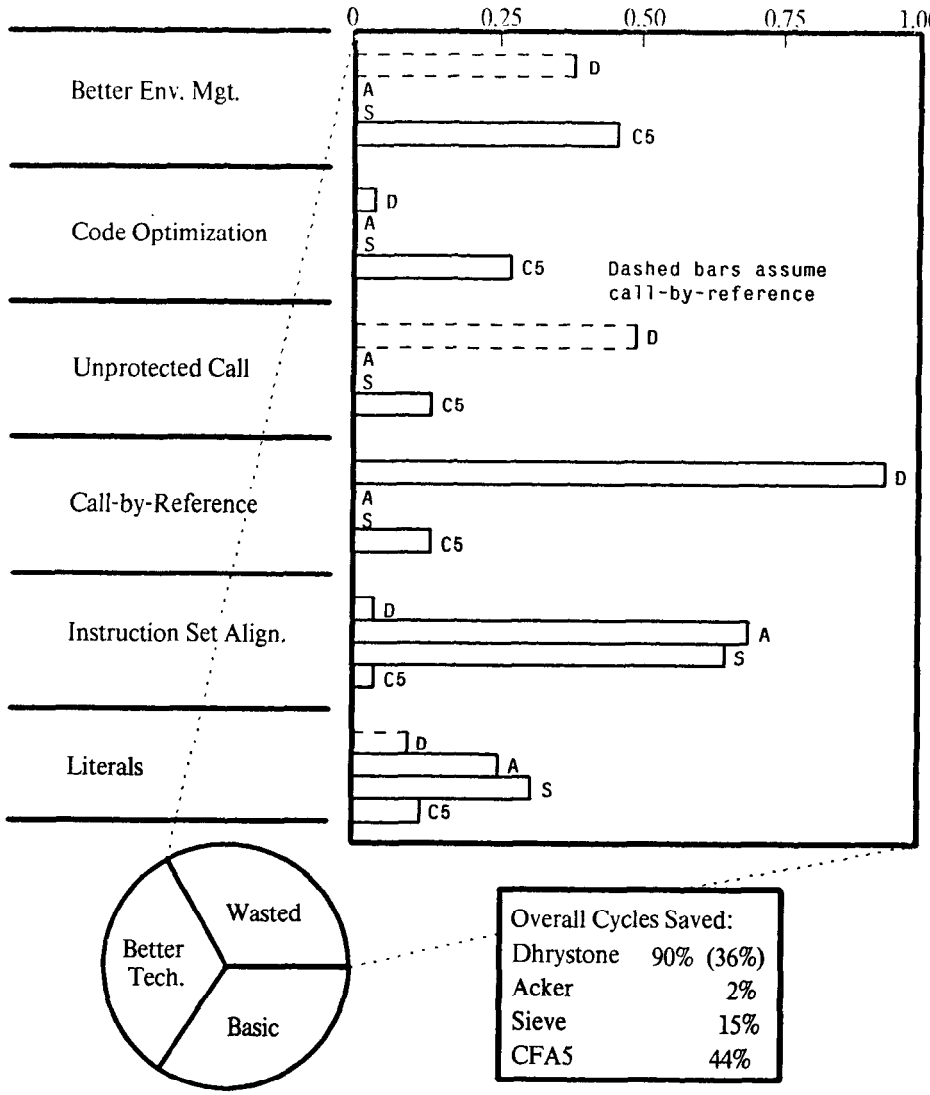

Fig. 8. Relative contributions of cycle sinks to overall wasted cycles.

6.2 Incrementally Better Technology

The improvements to the 432 system and architecture discussed in the previous section were mainly rectifications of errors in the 432 design or implementation, assuming the original implementation technology. In this section we consider the performance improvements possible if an incrementally better technology (smaller feature size, for instance) were available for a new instantiation of the 432 architecture. This situation often occurs in the microprocessor design industry (e.g., a processor is currently being marketed and sold, with the next

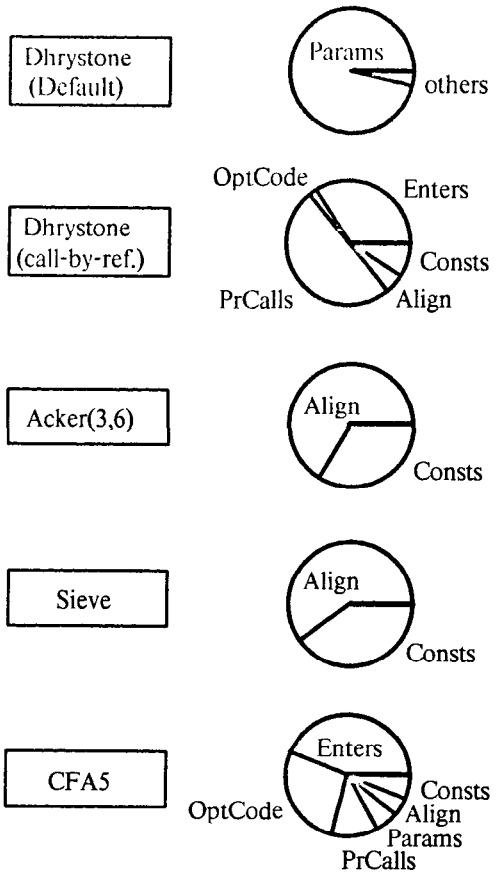

Fig. 9. Relative contributions of cycle sinks to overall wasted cycles by benchmark.

generation design underway). The Motorola 68000/68020/68030 and the Intel 8086/80286/80386 microprocessors are examples. We consider the following improvements to the 432.[8]

—provision for local data registers (Section 5.2)
—expansion of internal and external buses to 32 bits
—expansion of the Top-of-Stack register to 32 bits
—an extra bit on the μInstruction Bus
—an AD_Cache (Section 5.6.3)
—a memory-clearing primitive operation.

Each of these items was discussed in detail in Section 5 except the memory-clearing primitive. A protected, object-based procedure call must clear memory

[8] We are not dealing with clock-rate improvements here. Smaller feature size makes the gates faster and the capacitance lower, so a faster clock rate becomes possible. However, without making major changes to the architecture, the clock rate is not one of the parameters that is under the architect's direct control. Here we assume that the clock rate is fixed by the basic technology, and that the design goal is to optimize use of the available resources.

Table XIX. Cycles Saved with Incrementally Better Implementation Technology

Benchmark	Data regs	32-bit buses	32-bit TOS	17-bit μInstr	AD_Cache	Mem Clr
Acker	0	58556312	0	1668603	15845470	54381776
Sieve	3681926	793614	0	14065	0	0
CFA5	5155000	4842412	2516000	55763	2278752	364870
CFA5R	4861000	12218554	3944000	228111	4350745	396490
CFA10	8415000	7418454	595000	93244	4179242	326120
Dhrystone	1905	10242	221	123	6188	5187

before allocating it to a new context. Otherwise, the memory could contain leftover random data that could masquerade as ADs, permitting the callee to access random regions of online storage. It would be possible [10] to relegate the memory-clearing operation to some dedicated hardware, perhaps to a suitably modified 432 Interface Processor or to the Memory Control Unit. The performance speedup from shifting this responsibility is the difference in cycles between the time that the GDP takes to complete this operation vs. the cost of communicating the size and location of the segments to be cleared to the I/O controller, plus cycles lost to memory contention thereafter (while the controller performs the clearing). We assume here that the cost of communication between the GDP and the I/O controller is 30 cycles (two I/O writes), and that only minimal interference occurs.[9] Given these assumptions, the procedure call would be speeded up by approximately 31 percent.

Table XIX shows the cycles that could be saved over the original baseline numbers if these improvements were incorporated. This table can be used to compare the relative cycle contributions of the "errors" discussed in the previous section to the contributions of the architectural changes discussed here. Figure 10 depicts these contributions graphically. Figure 11 shows how the contributions change with each benchmark. Table XX shows the overall improvement broken down by percentages, and Table XXI shows the percent speedup over the original baseline numbers.

Some care must be taken here in using the cycle savings reported in Section 5 for these improvements. In analyzing the benchmark log files, categories for cycle use were strictly segregated, but some interaction is unavoidable. For example the cycles saved by adding local registers are not eligible to be speeded up due to wider buses. If instruction stream literals were available, then better *enter_environment* management does not save quite as many cycles as it would otherwise (since the first memory reference of an *enter* is to a Constants object). To avoid errors due to double-counting, each category's total in Table XIX has been adjusted as appropriate (this is why they do not match the totals shown in Section 5).

Table XXII shows the additional performance improvement over the synthetic baseline due to the architecture and implementation changes listed above. It indicates that the combined effect of all changes made to the architecture, compiler, and implementation technology was uniform across the

[9] This seems reasonable because the 432 procedure call is not heavily memory-intensive except for the memory-clearing sequence.

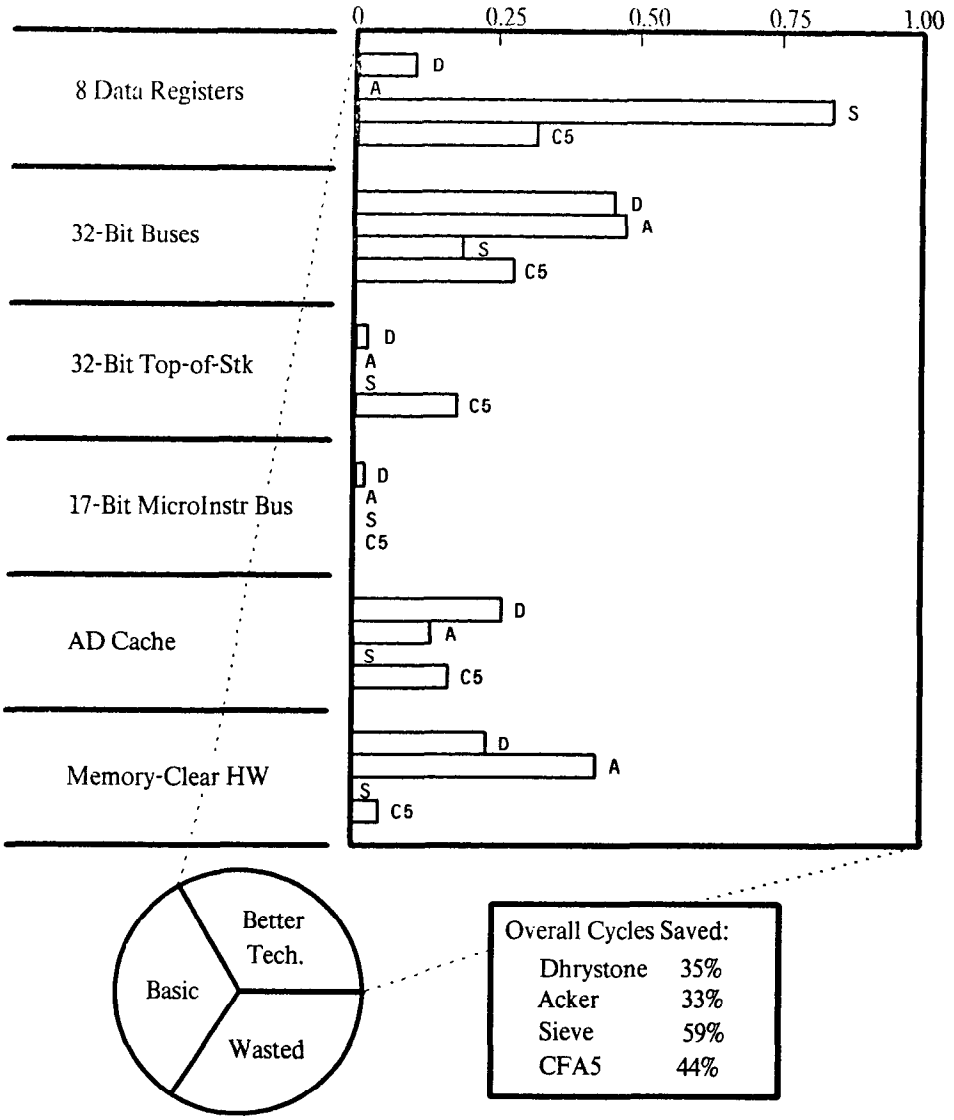

Fig. 10. Relative contributions of incremental technology improvements.

"compute-bound" benchmarks such as CFA5, CFA5R, and CFA10 in spite of the different ways these benchmarks stress the machine. This table also shows that the performance of the *Sieve* benchmark has been increased to the point where it is now competitive with other machines such as the Motorola 68000 and the Intel 8086 (compare the real time in Table XXII to the original Berkeley measurement for the other machines, listed in Table IV). The 432 architects have long asserted that, for such benchmarks, the 432 should exhibit no major performance liabilities once the object-based operations such as DS_Cache

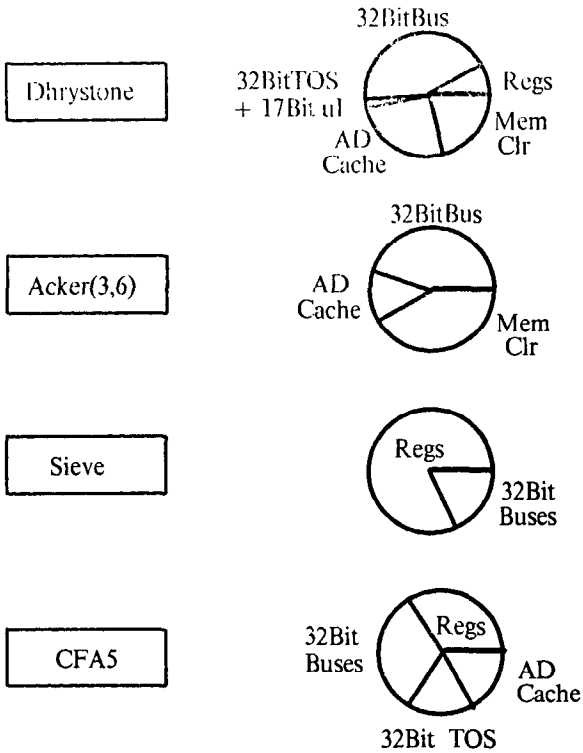

Fig. 11. Relative contributions of incremental technology improvements by benchmark.

Table XX. Cycles Saved with Incrementally Better Implementation Technology by Percentage

Benchmark	Data regs	32-bit buses	32-bit TOS	17-bit μInstr	AD_Cache	Mem Clr
Acker	0	45	0	1	12	42
Sieve	82	18	0	0	0	0
CFA5	34	32	17	0	15	2
CFA5R	19	47	15	1	17	1
CFA10	40	35	3	0	20	2
Dhrystone	8	43	1	1	26	22

management, context creation, and *enter_environments* have been done. This result is the first direct evidence for that claim.

We can relate these results to other machines by comparing the 432's Dhrystone real-time to the VAX and current microprocessors. Weicker's report of some preliminary measurements on contemporary processors [36] can be summarized as follows. The VAX 11/780 runs the Dhrystone in 540–1,800 microseconds, depending on operating system (VMS or UNIX), language (C or Pascal), and compiler switches selected (optimizing or not, checking enabled or disabled). The ELXSI superminicomputer requires 110–135 microseconds, and other superminicomputers are in the 200–500 microsecond range. Recent 16-bit

Table XXI. New Benchmark Cycles and Percent Improvement over Original Baseline

Benchmark	Cycles saved	% Original base cycles saved	Improved techn. cycles
Acker	130452160	33	264021113
Sieve	4489605	59	4489605
CFA5	15212797	44	19692875
CFA5R	25998902	43	25983605
CFA10	21027060	43	27769185
Dhrystone	23866	35	45150

Table XXII. Total Synthetic Baseline Cycles, Percent Improvement over Original Baseline, and Real-Time in Milliseconds

Benchmark	Final total cycles	% Original base cycles saved	Real time in mS
Acker	257319033	35	32165
Sieve	2653785	65	332
CFA5	11025390	68	1378
CFA5R	21050576	66	2631
CFA10	15394492	68	1924
Dhrystone	28709	58	3.59

microprocessors (Intel 80286, Motorola 68000) require from 1,000–1,500 microseconds, with 8-bit microprocessors (Intel 8088) taking 2,400–9,600 microseconds, again depending on operating system, language, compiler switches, memory speeds, and clock frequency.

As a rough approximation, these results imply that the synthetic and technology-improved 432 is approximately 3–4 times slower than the newer 16-bit microprocessors. The synthetic/improved 432 is faster than (or at least competitive with) some other reported microprocessor results, such as the 5.2 millisecond time of the Osborne machine under TurboPascal, or the 4.8 milliseconds reported for the IBM PC running Pascal. Keep in mind that the 432 was programmed in Ada while all other machines in this comparison were programmed in C or Pascal. The 432 result includes the code optimization, addition of local registers, wider buses, call-by-reference parameter-passing where appropriate, and all of the other changes discussed above. Consequently the 432 speeds are indicative of the best performance to which the 432 could have aspired originally. Allowing for differences in implementation technology between the 432 and the 16-bit microprocessors, we estimate that the synthetic 432 would still have taken between two and three times as long as other microprocessors to run the Dhrystone benchmark. This will be our estimate for the inherent cost of the 432's style of object orientation.

Note that this performance ratio must be used with care. This is a rough estimate, since it is essentially comparing apples and oranges (but that is what is called for in estimating the overhead of object orientation vs. conventional systems). This data point does not prove that all object-based systems can only

hope to run within a factor of two or three of conventional systems. The 432 represents a specific point in the design/implementation space, incorporating a certain set of design decisions. What we have shown here implies that a designer who builds in an equivalent set of decisions about object orientation into a new machine will incur an overhead in performance that is similar. By approaching object orientation in a different way, but still without discarding the flexibility of capability-based protection, even greater improvements should be possible. For example we have argued elsewhere [10] that more radical changes to the 432's procedure-call mechanism could bring it up to a speed that compares favorably, by most measures, to that of the VAX or MC68010.

7. CONCLUSIONS

The 432 was unique among microprocessors in the degree to which it incorporated architectural innovations. Perhaps due to the initial barrage of publicity and the consequent high expectations, the disappointing reality of the 432's performance made it the favorite target for whatever point a researcher wanted to make [14, 16, 29, 33]. Through a detailed case study, this paper has shown that many of the RISC criticisms of the CISC design style find apt targets in the Intel 432. However, we have also argued that, in several cases, published RISC work does not indicate what the 432 *should* have done.

This paper has shown that the 432 loses some 25–35 percent of its potential throughput due to the poor quality of code emitted by its Ada compiler. Another 5–10 percent is lost to implementation inefficiencies such as the 432's lack of instruction stream literals and its instruction stream bit-alignment. These losses are substantial, and essentially unrelated to instruction set complexity or object orientation. As such they constitute a stark warning to all computer architects about the magnitude of losses that can appear in any implementation unless close control over every aspect of the design is maintained.

Having established what the 432 should have done differently, we proceeded to investigate what it *could* have done had its implementation technology been incrementally better. We found that a combination of plausible modifications to the 432, such as wider buses and provision for local data registers, increased performance by another 35–45 percent.

This left the 432 executing the benchmark programs used throughout this paper from one to four times slower than conventional processors (where "one times slower" means approximately equal in performance). We called this ratio the inherent cost of the 432's style of object orientation, but cautioned that other approaches to object-based architecture are possible.

There is no doubt that as a commercial venture whose purpose was to earn profits for its manufacturer, the 432 failed completely. But it is more enlightening to view it as a research effort that happened to be funded by an IC manufacturer. The 432 probably tried to do too many new things all at once (while getting a few old things wrong along the way) to succeed commercially. The market it was targeted for, high reliability/high availability/large software-systems development, may still not be large enough or defined well enough to economically support the introduction of special systems.

As a research effort the 432 was a remarkable success. It proved that many independent concepts such as flow-of-control, program modularization, storage hierarchies, virtual memory, message-passing, and process/processor scheduling could all be subsumed under a unified set of ideas. These concepts have attracted wide interest, but interest has lately been dulled somewhat by a fear that the 432's experience strikes at the viability of the concepts. It is to be hoped that interest can be rejuvenated by our demonstration that the 432's performance has been dominated, in large part, by artifacts and not by concepts.

REFERENCES

1. BAYLISS, J. A., COLLEY, S. R., KRAVITZ, R. H., MCCORMICK, G. A., RICHARDSON, W. S., WILDE, D. K., AND WITTMER, L. L. The instruction decoding unit for the VLSI 432 general data processor. *IEEE J. Solid-State Circuits SC-16*, 5 (Oct. 1981), 531–537.
2. BAYLISS, J. A., DEETZ, J. A., NG, C. K., OGILVIE, S. A., PETERSON, C. B., AND WILDE, D. K. The interface processor for the Intel VLSI 432 32-bit computer. *IEEE J. Solid-State Circuits SC-16*, 5 (Oct. 1981), 522–530.
3. CLARK, D. W. Cache performance in the VAX-11/780. *ACM Trans. Comput. Syst. 1*, 1 (Feb. 1983), 24–37.
4. CLARK, D. W., AND EMER, J. S. Performance of the VAX-11/780 translation buffer: Simulation and measurement. *ACM Trans. Comput. Syst. 3*, 1 (Feb. 1985), 31–62.
5. COLWELL, R. P. The performance effects of functional migration and architectural complexity in object-oriented systems. Ph.D. dissertation, Carnegie-Mellon Univ. 1985. Also available as technical report CMU-CS-85-159, Dept. of Computer Science, Carnegie-Mellon Univ.
6. COX, G. W., CORWIN, W. M., LAI, K. K., AND POLLACK, F. J. Intreprocess communication and processor dispatching on the Intel 432. *ACM Trans. Comput. Syst. 1*, 1 (Feb. 1983), 45–66.
7. DITZEL, D. R., AND PATTERSON, D. A. Retrospective on high-level language computer architecture. In *Proceedings of the 7th Annual Symposium on Computer Architecture* (Le Baule, France, May 6–8, 1980). IEEE Computer Society and ACM, New York, 1980, pp. 97–104.
8. FABRY, R. S. Capability-based addressing. *Commun. ACM 17*, 7 (July 1974), 403–412.
9. FULLER, S. H., AND BURR, W. E. Measurement and evaluation of alternative computer architectures. *Computer* (Oct. 1977), 24–35.
10. GEHRINGER, E. F., AND COLWELL, R. P. Fast object-oriented procedure calls: Lessons from the Intel 432. In *Proceedings of the 13th Annual Symposium on Computer Architecture* (June 1986), pp. 92–101.
11. GEHRINGER, E. F., SIEWIOREK, D. P., AND SEGALL, Z. Z. *Parallel Processing: the Cm* Experience.* DEC Press, 1987.
12. GOLDBERG, A., AND ROBSON, D. *SmallTalk-80: the Language and its Implementation.* Addison-Wesley, Reading, Mass., 1983.
13. HANSEN, P. M., LINTON, M. A., MAYO, R. N., MURPHY, M., AND PATTERSON, D. A. A performance evaluation of the Intel iAPX 432. *Comput. Architecture News 10*, 4 (June 1982).
14. HENNESSY, J. L. VLSI Processor Architecture. *IEEE Trans. Comput. C-33*, 12 (Dec. 1984), 1221–1246.
15. HENNESSY, J., JOUPPI, N., BASKETT, F., GROSS, T., AND GILL, J. Hardware/software tradeoffs for increased performance. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems* (Palo Alto, Calif., March 1–3, 1982). ACM, New York, 1982, pp. 2–11.
16. HILL, D. D. An analysis of C machine support for other block-structured languages. *Comput. Architecture News 11*, 4 (Sept. 1983), 7–16.
17. INTEL CORPORATION. Introduction to the iAPX 432 Architecture Manual 171821-001, Intel Corporation, Santa Clara, Calif., 1981.
18. INTEL CORPORATION. iAPX 432 Object Primer Manual 171858-001, Rev. B. Intel Corporation, Santa Clara, Calif., 1981.
19. INTEL CORPORATION. iAPX 432 General Data Processor Architecture Reference Manual, Rev. 3 (Advance Partial Issue) Manual 171860-003, Intel Corporation, Santa Clara, Calif., 1982.

20. INTEL CORPORATION. Ada Description of iAPX 432 Microcode Algorithms (Rel. 3). Intel Corporation, Santa Clara, Calif., 1982.
21. JOHNSON, D. The Intel 432: a VLSI Architecture for fault-tolerant computer systems. *Computer 17*, 8 (August 1984), 40–48.
22. JONES, A. K. The object model: A conceptual tool for structuring software. In *Operating Systems*, R. Bayer, R. M. Graham, G. Seegmuller, Eds. Springer-Verlag, New York, 1979, pp. 7–16.
23. JONES, D. W. Systematic protection mechanism design. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems* (Palo Alto, Calif., March 1–3, 1982). ACM, New York, 1982, pp. 77–80.
24. LEVERETT, B. W., CATTEL, R. G. G., HOBBS, S. O., NEWCOMER, J. M., REINER, A. H., SCHATZ, B. R., AND WULF, W. A. An overview of the production quality compiler-compiler project. *Computer 13*, 8 (Aug. 1980).
25. LEVY, H. M. *Capability-Based Computer Systems.* Digital Press, Billerica, Mass., 1984.
26. MUDGE, T. N., BUZZARD, G. D., VERHAEGHE, D. J., HILL, J., AND WINSOR, D. C. Object-based computer architectures. Tech. Rep. CRL-TR-18-83, Dept. of EECS, Univ. of Michigan, April, 1983.
27. ORGANICK, E. *A Programmer's View of the Intel 432.* McGraw-Hill, New York, 1983.
28. ORGANICK, E., AND HINDS, J. A. *Interpreting Machines: Architecture and Programming of the B1700/B1800 Series.* North Holland, Amsterdam, 1978.
29. PATTERSON, D. A. Reduced instruction set computers. *Commun. ACM 28*, 1 (Jan. 1985), 8–21.
30. POLLACK, F. J., COX, G. W., HAMMERSTROM, D. W., KAHN, K. C., LAI, K. K., AND RATTNER, J. R. Supporting Ada Memory Management in the iAPX-432. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems* (Palo Alto, Calif., March 1–3, 1982). ACM, New York, 1982, pp. 117–131.
31. PRATT, T. W. *Programming Languages, Design and Implementation.* Prentice-Hall, Englewood Cliffs, N.J., 1984.
32. SMITH, A. J. Cache memories. *Comput. Surv. 14*, 3 (Sept. 1982), 473–530.
33. TREDENNICK, N. Future microprocessors. Lecture given at Carnegie-Mellon Univ., Dept. of Computer Science, Pittsburgh, Pa., April 12, 1985.
34. WEGNER, P. Dimensions of object-based language design. In *OOPSLA '87 Conference Proceedings* (Orlando, Fla., Oct. 4–8, 1987). ACM/SIGPLAN, ACM, New York, 1987, pp. 168–182. Also in *ACM SIGPLAN Not. 22*, 12 (Dec. 1987).
35. WEICKER, R. P. Dhrystone: A synthetic systems programming benchmark. *Commun. ACM 27*, 10 (Oct. 1984), 1013–1030.
36. WEICKER, R. P. Execution times for the 'Dhrystone' benchmark program. March 1985. To be published.
37. WELCH, T. A. An investigation of descriptor-oriented architecture. In *Proceedings of the 3rd Annual Symposium on Computer Architecture* (Jan. 19–21, 1976). IEEE Press, New York, 1976, pp. 141–146.
38. WILKES, M. V., AND NEEDHAM, R. M. *The Cambridge CAP Computer and its Operating System.* North Holland, Amsterdam, 1979.
39. WULF, W. A. Compilers and computer architecture. *Computer 14*, 7 (July 1981), 41–48.
40. WULF, W. A., HARBISON, S., AND LEVIN, R. *Hydra/C.mmp: An Experimental Computer System.* McGraw-Hill, New York, 1981.

Received January 1987; revised November 1987; accepted December 1987