Evaluation of On-Chip Cache Memories

Mark D. Hill

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

ABSTRACT

Advances in integrated circuit density are permitting the implementation on a single chip of features, functions and performance enhancements beyond those of basic eight and sixteen bit processors. One performance enhancement of proven value is a cache memory; placing a cache on the processor chip can reduce both mean memory access time and bus traffic. Optimum design points for on-chip caches will differ from those for mini and mainframe computers because of the limited chip area and the small number of pins, and thus on-chip caches will be smaller (e.g. 32-2048 bytes) and will have smaller block (line) sizes. The potential also exists for on-chip caches to be "smarter" through the use of special purpose cache control logic to exploit the processor state information that is on the chip.

In this paper, we use trace driven simulation to study design tradeoffs for small (on-chip) caches. Miss ratio and traffic ratio (bus traffic) are the metrics for cache performance, and cache cost is measured by the gross cache size, which includes address tag area as well as data array area. Particular attention is paid to sub-block caches (also known as sector caches) and the RISC II instruction cache. In sub-block placement, address tags are associated with blocks, each of which contains multiple sub-blocks; sub-blocks are the transfer unit. Using traces from two 16-bit architectures (Z8000, PDP-11), we find that caches of less than 256 bytes (gross size) (e.g. 190 bytes) yield usefully low miss ratios (PDP-11-0.367, Z8000-0.257) and reduced traffic ratios (0.734, 0.514). Limited results from the 32-bit IBM System/370 show that a cache of 380 bytes (gross size) performs with a miss and traffic ratios of 0.382 and 0.766. The use of sub-blocks allows tradeoffs between miss ratio and traffic ratio for a given cache size. Load forward is quite useful. Extensive simulation results are presented.

The RISC II instruction cache explores four ideas in VLSI caches: multi-chip expansibility for flexibility, remote program counter for decreased access time, support for dynamic code expansion for increased effective size, and fault tolerance for decreased cost.

December 8, 1983
MARK D. HILL
Author

Evaluation of On-Chip Cache Memories

Title

RESEARCH PROJECT

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, to partial satisfaction of the requirements for the degree of Master of Sciences, Plan II.

Approval for the Report and Comprehensive Examination:

Committee: Research Adviser

D.A. Patterson, Second Reader

December 7, 1983, Date
# Table of Contents

1. Introduction .................................................................................................................. 1
   1.1. Review of Literature ............................................................................................... 3

2. Design of On-Chip Caches .......................................................................................... 5
   2.1. Effects of VLSI ....................................................................................................... 5
   2.2. Two Examples of On-Chip Cache Architectures .................................................. 5
   2.3. Cache Design Parameters ....................................................................................... 7
   2.4. Relevant Metrics ..................................................................................................... 8

3. Methods ........................................................................................................................ 10
   3.1. Analytic Modeling, Hardware Monitoring, and Simulation .................................. 10
   3.2. Trace-Driven Simulation ....................................................................................... 11
   3.3. Simulation with Priority Stacks ............................................................................. 12
   3.4. Simulator Implementation: Dinero ....................................................................... 13
   3.4.1. Structure of Trace-Driven Simulator Dinero .................................................... 13
   3.4.2. Features of Trace-Driven Simulator Dinero ...................................................... 13
   3.4.2.1. Static Structure Parameters ........................................................................... 14
   3.4.2.2. Dynamic Characteristics .............................................................................. 16
   3.4.2.3. I/O and Multiprogramming .......................................................................... 17
   3.4.2.4. Input Specifications ....................................................................................... 17
   3.5. Traces Studied ........................................................................................................ 17

4. Results ......................................................................................................................... 20
   4.1. Early Sub-Block Placement .................................................................................... 20
   4.2. PDP-11 Results ..................................................................................................... 21
   4.3. Paged-Mode PDP-11 Results ................................................................................ 26
   4.4. Z8000 Results ....................................................................................................... 28
   4.5. Load-Forward Results ........................................................................................... 28
   4.6. Other Results ......................................................................................................... 35

5. RISC II Instruction Cache .......................................................................................... 39
   5.1. RISC Background ................................................................................................. 39
   5.2. Four Architectural Ideas ....................................................................................... 40
   5.3. RISC II Simulation Results ................................................................................... 42

6. Conclusions .................................................................................................................. 46

References .......................................................................................................................... 49
Evaluation of On-Chip Cache Memories

Mark D. Hill

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

1. Introduction

Advancing integrated circuit fabrication techniques are increasing the scope of what can be done on a single chip. This scale is better measured by the function of a chip rather than a transistor count. Capabilities have moved from NANDs to ALUs to extremely simple 4-bit processors to today's microprocessors that can execute full mainframe instruction sets. Future fabrication improvements promise us the next functional leap: computer systems on a single chip. Currently, although one can put more than a simple processor on a chip, it is not possible to put an entire system on a chip. We see two options for effectively using chip area beyond what is necessary for a simple processor: 1) special-purpose logic to expand functionality or increase performance and 2) on-chip cache memories. Eventually additional logic will not be cost-effective because the speed-ups are too small (too rare), or the microprocessor will become "pin-limited" (i.e. the computational engine is limited by its ability to get information from and to the outside world). In either case, it makes sense to put a cache memory on the chip to reduce access time and decrease the frequency of external accesses.

Caches are a time-tested mechanism for for improving memory system performance by reducing access time and memory traffic through the exploitation of temporal and spatial locality. Temporal locality says a location recently referenced is likely to be referenced again in the near future. Spatial locality says the same thing for locations near the recent reference. A good survey of cache memories was written by Smith [1].

On-chip caches will differ from traditional caches. Initially these caches will be small (32 to 2048 bytes) because they will only occupy the area that they use more effectively than special-purpose logic. As fabrication techniques provide room for more transistors, more transistors will be used more effectively in a cache than elsewhere. Blocks in on-chip caches will tend to be smaller than traditional blocks because packaging limitations prevent large parallel loads. Worry about bus traffic could induce schemes that exploit temporal locality more than spatial locality. Lastly, special-purpose cache control logic is inexpensive and can easily get processor state information. This should lead to more intelligent caches.
This paper examines small caches with trace-driven simulation to find to measure how well these caches perform so that microprocessor designers can compare these benefits with those of adding some special-purpose logic. In particular, it examines a cache organization called sub-block placement and the four innovations of the RISC II Instruction Cache.

In sub-block placement the address tags in the cache are associated with a larger amount of data than is usually brought into the cache on a single miss. This scheme allows smaller units, called sub-blocks, to be used in memory transfers to keep bus traffic from memory to a minimum. Address tags are associated with blocks so that the chip area devoted to address tags does not grow too large. This structure was used less effectively at a larger scale in the IBM System/360 Model 85 [2]. Basically, a block is composed of an address tag and two or more sub-blocks. When a reference is made to a totally new place, a block is allocated and only the missing sub-block is loaded. We define the sub-block size as the number of bytes moved to the cache by a minimum size data transfer. The block size is the maximum number of bytes of data that can be placed or associated with a single address tag in the cache. In most caches, the sub-block size and the block size are the same and collectively referred to as the block size. The expectation in sub-block placement is that saving in tag overhead makes up for the rigidity of the mapping.

In addition to exploring sub-block placement, this paper looks at an experiment in on-chip cache architectures in which a separate cache chip was built. This chip, the RISC II Instruction Cache, explores four ideas: multi-chip expansibility for flexibility, remote program counter for decreased access time, support for dynamic code expansion for increased effective size, and fault tolerance for increased yield. A key idea here is the remote program counter that is an example of using special-purpose cache control to improve performance. The remote program counter attempts to prefetch the next instruction out of the cache, based on its knowledge of RISC II instructions.

This paper is organized into six sections. This section concludes with a review of the literature. Section 2 looks at the design of on-chip caches. It outlines the effects of VLSI on cache design, proposes some examples, and presents the metrics of miss ratio and traffic ratio that we use to judge the performance of caches in this study. Section 3 justifies the use of trace-driven simulation, describes the features of the simulator that was implemented, and presents the traces used. Section 4 presents experimental results primarily of PDP-11 and 28000 traces. We found that sub-block placement allows miss and traffic ratios to be traded off, nibble and paged-mode memories work well with on-chip caches, and loading (called load-forward) the sub-blocks after a reference is a useful way of prefetching. Section 5 looks at the RISC II Instruction Cache results. Conclusions are presented in section 6.
1.1. Review of Literature

The first cache memory implementation was done for the IBM System/360 Model 85. Results were published by Liptay [2] in 1968. This cache is, to our knowledge, the only early cache to use a sub-block placement structure. This 16-Kbyte cache associates an address tag with a 1-Kbyte block, but does memory transfers with 64-byte sub-blocks.\footnote{Liptay refers to the block as a sector and to the sub-block as a block.} This cache was a large step forward. However, results in this paper show that it performs poorly by today's standards because at any one time, data in the cache can only come from one of 16 blocks of 1024 bytes. The cache was organized in this way to minimize expensive associative address-matching hardware. We look at using the System/360 Model 85's cache organization on a much smaller scale.

Few cache studies in the literature have been done for the small cache sizes (less than 2048 bytes of data) that we propose for initial on-chip caches. In 1974, Bell, Casasent, and Bell [3] published results on small caches with single-word (16-bit) blocks and direct-mapped placement for the DEC PDP-8 using traces of two scientific programs and the PDP-8 Assembler. They found numbers for small caches that can be converted into miss ratios that are worse than most results in this paper but are consistent with our worst runs. For example, a cache of 512 data bytes was found to have a miss ratio between 0.46 and 0.62.

In 1976, Strecker [4] summarized the research that led to the design of the cache memory in the DEC PDP-11/70. He used scientific applications and an assembler as traces. For direct-mapped caches with a block size of 4 bytes (2 PDP-11 words), the miss ratio dropped from 0.15 through 0.10, 0.05, and 0.02 as the cache data size was doubled from 256 bytes up to 2048 bytes. These numbers fit our data well. Strecker confirms our belief that there exists useful performance gain in increasing the associativity of a cache from 1 (direct-mapped) to 2, but that further increases in associativity buy marginal improvement. He also presented results showing that there is little difference in the performance of LRU, FIFO, and RANDOM replacement algorithms. The PDP-11/70 cache was designed to be 1024 data bytes, with 4-byte blocks, and two-way set associative with random replacement.

Recently, Smith and Goodman [5] have looked at instruction cache organizations with a stochastic model of instruction loop behavior and trace-driven simulation. They find that for small instruction caches, RANDOM performs better than LRU and FIFO. This is because, with LRU and FIFO, if a cache is slightly smaller than the loop it contains, every block will be replaced before it is re-used. The performance of RANDOM decays more gradually. Smith and Goodman also find that a direct-mapped instruction cache may perform as well or better than a fully-associative cache of the same size because the constrained mapping will sometimes prevent you
from replacing blocks just prior to their reuse.

More recent work by Goodman [6] examines more general, small caches within a multiprocessor environment. Among other issues, Goodman resurrects the structure in the System/360 Model 85's cache that permits a different size block for placement in the cache versus transfer from memory.\textsuperscript{2} Based on six UNIX\textsuperscript{3} traces, Goodman concludes that it is a good idea to have the block size larger than the sub-block size. In this paper we present extensive simulation results supporting the use of sub-blocks to move a cache's operating point to reduce either the miss ratio and the traffic ratio.

Patterson et al [7] present results on the RISC II Instruction Cache. Section 5 of shows additional data that supports the conclusions of Patterson et al. Some of the material from [7] is repeated here to document more of the author's experience enroute to this report. RISC is a acronym for Reduced Instruction Set Computer [8, 9]. RISC attempts to do frequent instructions fast and support procedure calls by implementing a simple hard-wired control and having multiple banks of general registers on chip.

\textsuperscript{2} Goodman refers to the block as a address block and to the sub-block as a transfer block.

\textsuperscript{3} UNIX is a trademark of Bell Laboratories.
2. Design of On-Chip Caches

This section discusses effects of VLSI on on-chip caches, including the disparity between on-chip and off-chip communication and the ability to inexpensively build specialized cache control. Next, two examples are proposed: the minimum cache and the smart cache. The minimum cache costs very little but can reduce off-chip accesses by one-third. The smart cache may perform better than current caches of the same size. The next sub-section discusses why we studied in detail only cache size, sub-block size, and block size. Lastly, we explain that we use miss ratio and bus traffic as the metrics of cache performance, and gross cache size (the size of address tags and data) as the metric of cost.

2.1. Effects of VLSI

On-chip caches will differ from MSI (Medium Scale Integration) memory caches for at least four reasons. First, block sizes will tend to be smaller because integrated circuit packaging limitations curtail the number of bits that can be handled in parallel. Large blocks require that data be time-multiplexed over limited pins. This time-multiplexing must be carefully designed so that it does not induce excessive latency in cache misses. Second, since most successful microprocessors today (1983) are bus-limited [10], on-chip caches will be increasingly concerned with minimizing bus traffic. This trend will reduce the processor-to-cache transfer size. Third, intra-chip communication is less expensive than inter-chip communication, therefore one can use more processor state information to control the on-chip cache. Data communication off-chip requires that signals be scaled to reasonable levels, such as TTL compatible (5 volts), and be able to drive large capacitive loads. This costs chip area, power, and delay. The cost of intra-chip transfer of information, like current instruction-in-execution or other hints to the cache, may be low enough to justify their incremental improvements to cache performance. Last, the current freedom from standard building blocks in VLSI can allow on-chip caches to use more complex custom control. Wide associative searches can be easily implemented in VLSI with content-addressable memory (CAM) cells.

2.2. Two Examples of On-Chip Cache Architectures

Two architectures for on-chip caches are proposed in this sub-section. These architectures were not explicitly studied but serve to illustrate the potential for VLSI on-chip caches. The first, called a minimum cache, is a cross between an instruction buffer and a cache. The second, a smart cache, is more like a traditional cache except that it uses custom VLSI to capitalize on the processor state information that is available on-chip.
Instructions buffers can be simple [11]. For example, the DEC VAX-11/780 and 750 have 8-byte instruction buffers that must contain eight contiguous bytes [12]. Simple instruction buffers differ from caches because they do not reduce the number of bytes required from the memory system; they only reduce latency by starting the instruction fetch before the processor needs the instruction. In contrast, more complex instruction buffers try to ease bandwidth requirements on the memory system. For example, the CRAY-1 has four instruction buffers that can each hold 64 contiguous 16-bit instructions for a total size of 512 bytes [13]. The CRAY-1's instruction buffers blur the distinction between an instruction buffer and an instruction cache because they are designed to reduce requests to main memory by holding entire loops. The CRAY-1's buffers are effectively an instruction cache of four blocks.

We believe that some of the scarce area on microprocessor chips should be devoted to at least simple, small caches to hold instructions, data, or both. These minimum caches can perform with miss ratios of 0.70 which still cuts the off-chip bandwidth requirement by over one-third. One possible architecture for a minimum cache is 32 data words broken into 16 2-word blocks, where only the requested word is loaded on a miss. For this architecture, it is reasonable to use 2-way set-associative placement with RANDOM replacement. A minimum cache for a 32-bit machine would require about 190 bytes of RAM (16 blocks * [29 tag bits + 2 valid bits + 64 data bits] / 8 bits/byte) plus some overhead for buffers and matching logic. We believe that the minimum cache will improve system performance most if it is used to cache instructions only.

The second cache proposal requires more area than the minimum cache, say enough area for 1024 to 2048 bytes of information. The smart cache's performance is enhanced by exploiting the ability in VLSI to design inexpensive special-purpose logic to utilize the processor state information available on-chip. Special purpose logic can examine reference patterns to prefetch instruction codes and operands from main memory or prefetch information out of the cache memory into even faster buffers. For example, the S-1 [14] attempts to dynamically predict jump targets and the RISC II instruction cache [7] prefetches instruction data from its private store. Complex logic might even be able to analyze and predict data referencing patterns. Lee and Smith [11] present empirical evidence for using a buffer to cache recent jump targets to reduce the detrimental effects of branches on pipelined machines. Another idea would be to explicitly cache the tops of certain stacks in a programming environment. This may work better than a method ignorant of the semantics of the data. Effective prefetching reduces latency at a cost of increased memory traffic and a risk of memory pollution (replacing data while it is still being used). Intelligent data replacement reduces memory pollution thereby reducing bus traffic and latency.
2.3. Cache Design Parameters

Cache size, sub-block size, and block size are important to cache performance. Cache size directly affects performance and cost in a way that dominates all other design decisions. The miss ratio of a cache can, at best, asymptotically approach zero as cache size is increased, but the cost of the cache is directly related to its size. For this reason, cache sizes will not grow without bound, and the size of cache that will give the optimum cost/performance cannot be calculated without looking at performance and cost data.

*Sub-block size* is the number of bytes moved to the cache by a minimum size data transfer. *Block size (line size)* is the maximum number of bytes of data that can be placed or associated with a single address tag in the cache. In most caches, the sub-block size and the block size are identical and are collectively referred to as the *block size*. When large blocks are divided into smaller sub-blocks, two effects take place. First, the miss ratio will increase because more than one miss can occur within a single block. Second, the traffic ratio will decrease because only the parts of the block that are used will be loaded. In VLSI, we expect the smaller sub-block size to reduce the cost of a miss, because less data is brought across limited pins. Alternatively, sub-block placement can be viewed as a way of grouping n small blocks under one address tag. This grouping reduces freedom of what can be resident in the cache at one time by restricting the n sub-blocks to consecutive locations. This loss in flexibility of placement will cause the miss and traffic ratios to increase. However, the cache can be more cost-effective since address tag overhead has been reduced by a factor of n.⁴

Associativity, replacement algorithms, fetch algorithms, and write-back algorithms were not studied. All experiments were performed with 4-way set-associative caches. Smith [15] and others have shown that 4-way set-associative establishes a tight lower bound on fully-associative performance. Changes in associativity have been extensively studied and induce lower order effects than other parameters studied here [1, 4]. Experiments were done using *LRU* replacement because reasonable replacement algorithms like *pseudo-LRU* and *Random* perform similarly to *LRU*. All cache fetches were done on demand although sub-block placement with load-forward acts like prefetching. Prefetching studies should be done but were beyond the scope of this study. Write-back issues were filtered out of results by plotting bus traffic and miss ratios for only data reads and instruction fetches.

⁴We neglect the lower-order effects of changes in the number of bits in the tag.
2.4. Relevant Metrics

The purpose of a cache is to improve memory system performance by reducing the effective access time to memory and processor-memory bandwidth requirements. Effective access time $t_{\text{eff}}$ is most simply modeled as:

$$t_{\text{eff}} = t_{\text{cache}} \cdot (1 - m) + t_{\text{mem}} \cdot m,$$

where $m$ is the miss ratio of the cache, $t_{\text{cache}}$ is the access time of the cache, and $t_{\text{mem}}$ is the access time of main memory. $t_{\text{cache}}$ and $t_{\text{mem}}$ are not easy to obtain through any method that does not intimately consider a particular implementation of a machine. $t_{\text{cache}}$ is a function of the technology, organization, and complexity of a cache. $t_{\text{mem}}$ is related to memory technology, organization, complexity, and the reference distribution (bursts of references can exceed the memory's bandwidth). Miss ratios are often quoted for caches because they are implementation-independent (if the processor-to-cache datapath width is held constant) and thus can be computed in architectural studies. Nevertheless, a reduction in miss ratio will not guarantee an improvement in memory system performance. For example, prefetching data before it is needed should reduce the misses when the data is actually requested. Still, $t_{\text{eff}}$ may grow if the additional logic increases $t_{\text{cache}}$. Other aspects of cache include the overheads of updating main memory and maintaining multicache consistency.

The two most important architectural performance metrics are miss ratio and traffic ratio. The miss ratio is the number of cache misses divided by the number of cache accesses. It is always less than or equal to 1 because the number of misses cannot be more than the number of cache accesses. It is of primary importance if sufficient bus bandwidth is available so that reducing latency is the overriding goal.

The traffic ratio is the number of bytes transferred from the memory to the cache divided by the number of bytes transferred from the memory to a processor without a cache. It results from two competing factors. First, repeated references to words already in the cache reduces the traffic ratio. For example, assuming word references, if words are brought into the cache and used $n$ times before they are removed, then the traffic ratio will be $\frac{1}{n}$. Second, block sizes that are bigger than a single word will tend to increase the traffic ratio, because some data brought into the cache will never be referenced. For example, if only one reference is made to a block of $w$ words while it is in the cache, then the traffic ratio will be $\frac{w}{1}$. Together, the traffic ratio for a cache with $w$-word blocks will be less than 1 if and only if more than $w$ references are made to blocks each time the block is resident in the cache. The traffic ratio is important if the memory bus is the bottleneck, either because the single processor is too fast for the bus, or because there are multiple processors on the same bus. It also becomes more important if the ratio of memory
access time to cache access time is low so that latency improvements are less dramatic.

The most important cost metrics are gross cache size and cache complexity. By gross cache size we mean the size of the data and tag area together. Historically, cache performance numbers have been given with respect to cache data size (net cache size). On-chip caches can have non-trivial tag areas because block sizes are smaller, associativity may be greater, and address spaces are larger. The effects of cache complexity are important but are not studied here because they are implementation-dependent and thus poorly measured by architecture-level, trace-driven simulation.
3. Methods

This section explains the use of trace-driven simulation, explains the simulator *Dinero*\(^5\) that was written to conduct the experiments, and introduces the trace data. First, the three methods of evaluating computer systems are explained. Then, trace-driven simulation and simulation with priority stacks are introduced. Last, the features of Dinero are presented.

3.1. Analytic Modelling, Hardware Monitoring, and Simulation

There are three methods of evaluating computer systems: analytic modeling, hardware monitoring, and simulation \[16\]. This sub-section explains these methods.

An analytic model of a computer system is a mathematical or algorithmic description of the system that does not, in contrast to a simulation, mimic the modeled system in the time domain. This description can range in complexity from a back-of-the-envelope calculation to an intricate queueing network. An analytic model should be made solvable through abstractions that hide detail. Both accurate, intractable models and inaccurate, solvable models are of little value. Queueing models represent the deterministic activity of digital computers with stochastically, or randomly, distributed events. The rationale is that numerous complex, confusing events looked at in aggregate act stochastically. Analytic models usually yield solutions for much less cost than other methods. Sometimes the solutions can be expressed in a compact closed-form that can yield more insight than numerical results. Often, the intersection between analytically tractable and useful is empty. Therefore, one must resort to numerical methods or model simulation.

Hardware monitoring and prototyping is a second set of computer system evaluation tools. Both require the existence of and access to the system in question. Thus, design efforts cannot profit from these methods until prototypes exist. Hardware experiments are expensive and time-consuming to set up but yield the most accurate results because they implicitly take into account complex interactions and correlations between such things as pipeline stages, prefetch units, write buffers, TLBs, and cache memories. Coupled with all this accuracy is the constraint that almost nothing can be changed. One can only evaluate how design choices perform absolutely, but not relative to other points in the design space. Hardware experiments are not repeatable in practice because there are many factors that change the state of a machine in a way impossible to control.

Simulation is a modeling technique in which the model’s state at time \(t\) corresponds to the (computer) system’s state at time \(t\). As time goes on, both the model and the system make state transitions into states that continue to correspond. Usually a single transition in the simulation

\(^{5}\) *Dinero* is the Spanish word for money (cash, cache).
corresponds to many transitions in the system. The cost and accuracy of simulation usually lies between that of analytic modeling and hardware monitoring. Simulation models can span a wide level of detail resolution. Simulations can make transitions at fixed or variable intervals. Simulations whose clock changes by fixed increments are called clock-driven. Those whose clock is updated by varying amounts are called event-driven because the simulation clock is updated to the time of the next event significant to the simulation model. Clock-driven simulations are preferred when the simulator's state is updated by many factors at virtually every clock increment. Event-driven is preferred when inter-event times vary greatly and are usually non-zero (i.e. few events are simultaneous).

3.2. Trace-Driven Simulation

Trace-driven simulation is chosen because it is of reasonable cost and experiments are repeatable and representative of at least one program-in-execution.

System performance experiments must always be driven with with a model of the system workload (the input stimuli). This model can be a simple or complex. Workload models in analytic studies tend to be simple and not look like "software." The models in hardware studies are always executable, but still can vary from a typical kernel loop to a set of benchmarks to a "live" production workload. Kernel loops are part of a real workload that execute for a sufficient fraction of the time that their performance is a good indicator of system performance. Benchmarks are a set of programs that serve the same function. We call a finite sample of a live workload a natural workload model. It is only a model because like any other model its representativeness must still be verified.

Simulation workload models fall into two categories: natural and synthetic. Natural models are samples of real workloads. Usually system operation is distilled down to events that are believed to be significant to the study at hand, and those events are recorded on a trace tape. This tape is used to play back system events. These experiments are deterministic and repeatable. Traces can contain relationships and correlations between event data only dimly suspected by those performing the study. Synthetic models are models that are not natural. Such models may be probability distribution-driven or parametric programs constructed to exercise features of a system.

Trace-driven simulation experiments were used in these studies because they are repeatable, and they allow for the easy change of cache design parameters so that effects can be isolated. Simulations are cheaper than hardware monitoring or prototyping and do not require access to or the existence of the machine being studied. Simulation results can be obtained in many situations where analytic model solutions are intractable without questionable simplifying assumptions. Workloads are "modeled" by samples of real workloads that contain complex embedded
correlations that synthetic workloads often do not have.

3.3. Simulation with Priority Stacks

Experiments in this study were performed with a priority-stack-based simulator so that many cache sizes could be evaluated concurrently.

The priority stack technique is an important and time-tested method for memory hierarchy simulation. This method is described in Mattson et al [17] and Coffman and Denning [18]. Some cache replacement algorithms work in such a way that a larger cache will always contain a superset of the cache blocks resident in a smaller cache. This property is called the inclusion property and algorithms that satisfy it are called stack algorithms [17]. Also important are a class of algorithms called priority algorithms which have three properties: 1) the last referenced cache block is on the top of the stack, 2) no blocks except the referenced one may move up in the stack, and 3) no blocks below the old position of the referenced block may move. Priority algorithms and stack algorithms are more or less equivalent and often confused in the literature. All priority algorithms are stack algorithms. All stack algorithms are equivalent to another stack algorithm that is a priority algorithm [19].

The simplest priority stack cache is one which replaces the least recently used block in the cache of \( b \) blocks (called LRU replacement). When a block is referenced, the stack is searched. If the block is not found or is found at a position greater than \( b \), then a miss is recorded. Regardless of where the block was, it is moved to the top of the stack. All blocks between the top and the referenced block's old position are implicitly moved down one position. The size of the cache \( b \) being studied has no effect on the position of blocks in the priority stack. We can record the number of misses in caches of many different sizes by recording a miss for all caches whose size is smaller than the position of the block referenced. FIFO (First In First Out) is not a stack algorithm because the size of the cache being studied affects the stack ordering.

Random replacement, where all blocks in the cache have an equal chance of being replaced, can be formulated as a stack algorithm. Mattson et al [17] describe a priority algorithm which realizes random replacement on caches of all sizes at once. It involves allowing an unreferenced block at position \( i \) in the priority stack to retain its position with probability \( \frac{i-1}{i} \). The block that falls to position \( i+1 \) competes with the block originally there to remain at position \( i+1 \). It can be shown that every block in the top \( b \) positions of the priority stack has a \( \frac{1}{b} \) chance of falling below position \( b \).

The priority stack method, with one priority stack, models cases where all blocks are eligible for replacement on each reference. This cache organization is called fully-associative. Many caches restrict eligibility to a single set of blocks. This is done to restrict the range of the search
in the cache when a reference is made. This scheme, called *set-associative* replacement, merely partitions the address space to yield several independent problems which can be handled by the priority stack method using a priority stack for each set.

### 3.4. Simulator Implementation: Dinero

A trace-driven cache simulator called *Dinero* was designed and implemented in C [20] to perform the studies contained in this paper. Dinero was built using the priority stack method for memory hierarchy simulation. Many features have been included in Dinero that were not used in the studies presented here. Dinero allows transfers from main memory to the cache to be made in smaller units than cache blocks. Support for these *sub-blocks* increases Dinero's complexity particularly with respect to prefetch and write-update policies. The discussion that follows explains Dinero's structure, and presents its features.

#### 3.4.1. Structure of Trace-Driven Simulator Dinero

Dinero is written hierarchically. At the highest level, there are three routines: *init*, *mainloop*, *outputmetric*. *Init* initializes the simulator state including reading command line arguments, allocating storage for priority stacks, and setting critical pointers to initial values. *Mainloop* oversees the entire simulation. *Outputmetric* presents output to the user in readable form.

*Mainloop* further subdivides into two alternately executing halves called *fetch* and *update*. *Fetch* decides what address and what type of access will next be presented to the cache. This can be as simple as reading the next trace address to doing prefetches, I/O's, or context switches. *Fetch* returns a reference's address tag, set number, displacement within a block, sub-block number, access type (read, write, instruction fetch, or I/O), process id, mode (user or supervisor), and a prefetch flag.

*Update* modifies the priority stacks to reflect the cache's response to the address presented and updates metrics to record cache performance. A cache block's entry on a priority stack contains an address tag, dirty bits, and referenced bits. The entries are more complex when sub-blocks are used.

#### 3.4.2. Features of Trace-Driven Simulator Dinero

Dinero features break up into four categories. 1) *Static structure parameters* specify the size and architecture of the cache(s) being studied. 2) *Dynamic characteristics* are the replacement and other policies that guide cache control in execution. 3) *I/O and multiprogramming parameters* are available to model these activities in the simulation. 4) *Input specification* parameters allow some flexibility in the form of the trace input and duration of simulation.
3.4.2.1. Static Structure Parameters

Static structure parameters (Table 1) specify the structure of the cache(s) being studied. They specify the cache size, block size, sub-block size (when necessary), the degree of associativity, whether the cache is partitioned, and the datapath width.

Under appropriate conditions, the priority stack method of cache simulation will allow multiple cache sizes to be simulated at once. The number of sizes followed by a list of the cache sizes in bytes must be specified. These conditions restrict larger caches to be super-sets of smaller ones. Multiple cache sizes should not be used in situations that violate priority stack conditions. These violating conditions include: FIFO replacement, tagged or miss-only prefetching, copy-back main memory update policy, or sub-block placement. The maximum number of sizes is an arbitrarily large compile-time constant.

The block size (or line size) is the size of the data in bytes that is contiguously stored in the cache and associated with a single address tag. If sub-block placement is not being used, the block size is also the size of the main memory to cache transfer unit. For implementation reasons, block size must be an integral power of 2.

In traditional caches, the unit of allocation within the cache and the unit of transfer from main memory are the same. Dinero supports situations where the transfer unit, called the sub-block, is smaller. Sub-block size must be a power of 2 smaller than the block size. Referenced and dirty bits must be associated with sub-blocks instead of blocks. In addition resident bits indicate which sub-blocks of a block have been loaded.

The placement policy is assumed to be set-associative. The number of sets corresponds to the number of internal priority stacks. A set is the range of associative searches. If all sets have four elements, then the cache is four-way set associative, has set size of 4, and has degree of associativity of four. The degree of associativity in a simple (non-partitioned) cache is the cache size divided by the product of the number of sets and the block size. A cache with one set is called fully-associative; one with one element per set is called direct-mapped.

Most caches hold instructions and data from user and supervisor domains. It is possible to split caches along either of these lines. The split can increase the bandwidth of the cache if independent accesses are made to both halves simultaneously. Splitting the cache reduces flexibility of where data can be placed, but it can protect well-behaved data spaces from misbehaved ones. The partition halves need not be of identical size or ability. For example, instruction caches may not have to provide write-back capability, and supervisor caches may not have to worry about multiple virtual address spaces. Dinero only supports cache partitioning into halves of identical size and characteristics. Caches can be split by instruction/data or user/supervisor. This partitioning is done by having separate priority stacks for the different sides of the partition.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache size(s)</td>
<td>powers of 2 bytes</td>
</tr>
<tr>
<td>block size</td>
<td>power of 2 bytes</td>
</tr>
<tr>
<td>sub-block size</td>
<td>power of 2 bytes*</td>
</tr>
<tr>
<td>degree of associativity</td>
<td>power of 2</td>
</tr>
<tr>
<td>cache partitioning</td>
<td>unified</td>
</tr>
<tr>
<td></td>
<td>data/instruction</td>
</tr>
<tr>
<td></td>
<td>user/supervisor</td>
</tr>
<tr>
<td>datapath width</td>
<td>power of 2 bytes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>replacement</td>
<td>LRU</td>
</tr>
<tr>
<td></td>
<td>Random</td>
</tr>
<tr>
<td></td>
<td>FIFO*</td>
</tr>
<tr>
<td>fetch</td>
<td>demand</td>
</tr>
<tr>
<td></td>
<td>always-prefetch</td>
</tr>
<tr>
<td></td>
<td>tagged-prefetch* [1]</td>
</tr>
<tr>
<td></td>
<td>miss-prefetch*</td>
</tr>
<tr>
<td></td>
<td>nonwrite-miss-prefetch*</td>
</tr>
<tr>
<td></td>
<td>load-forward*</td>
</tr>
<tr>
<td>write-update</td>
<td>write-through</td>
</tr>
<tr>
<td></td>
<td>copy-back*</td>
</tr>
<tr>
<td>write-allocate</td>
<td>write-allocate</td>
</tr>
<tr>
<td></td>
<td>no-write-allocate</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O</td>
<td>number of addresses between I/O's</td>
</tr>
<tr>
<td>switch time</td>
<td>number of addresses between task switches (round-robin multiprogramming)</td>
</tr>
<tr>
<td>on switch</td>
<td>flush cache</td>
</tr>
<tr>
<td></td>
<td>do not flush cache</td>
</tr>
<tr>
<td></td>
<td>(prepend process ID to address tags)</td>
</tr>
</tbody>
</table>

* This option violates the assumptions of priority algorithms so only one cache size can be simulated at a time.
Therefore, a 4-way set-associative cache that is partitioned into data and instruction halves will need eight priority stacks.

The *datapath width* can be set to any power of 2. All reads to and writes from memory operate on data the size of the datapath width. Thus, logical reads and writes require the ceiling of \( \frac{\text{logical size}}{\text{datapath width}} \) physical memory operations.

### 3.4.2.2. Dynamic Characteristics

*Dynamic characteristics* (Table 2) are the policies that guide the cache in execution. They specify policies for replacement, fetch, write-update, and write-allocate.

Since a cache has finite capacity, old data must be replaced when new data enters. The block to be thrown out is chosen by the *replacement* policy. Policies implemented in Dinero are LRU, FIFO, and RANDOM. Many machines use LRU or an approximation to it, but FIFO and RANDOM have advantages in hardware implementation and only small performance differences from LRU for small caches.

The *fetch* policy determines when new data will be added to the cache. Demand fetching is the policy under which data is brought in only when a request is made for it. Policies that bring data in before it is requested are called *prefetch* policies. Prefetch policies must decide what and when to prefetch. Since caches are hardware managed, the most reasonable datum to prefetch is the (sub-)block sequentially after what is currently being accessed. The following prefetch policies are implemented in Dinero. *Always-prefetch* fetches one unit ahead after every request. *Tagged-prefetch*\(^1\) fetches one unit ahead only after data in a block has actually been used for the first time. *Miss-prefetch* fetches one unit ahead only after a miss to the (sub-)block preceding it. *Nonwrite-miss-prefetch* fetches a single unit after misses on instruction fetches and data reads. *Load-forward* fetches all sub-blocks in a block forward of the sub-block requested.

Processor writes must eventually be reflected in main memory. Dinero supports the two most common *write update* policies: *write-through* and *copy-back*. Write-through modifies main memory immediately, independent of whether the block referenced is in the cache. Copy-back does not modify main memory until the cache block is removed from the cache.

When a write to memory is made to a datum not in the cache, a decision must be made whether to bring the block into the cache and update it or simply update main memory. Dinero supports both *write-allocate* and *no-write-allocate*. 
3.4.2.3. I/O and Multiprogramming

I/O and multiprogramming (Table 3) are modeled with two parameters that specify the time until next I/O and time to next context switch and one parameter that says how a context switch affects data already in the cache.

I/O is modeled by an access to a location outside the current processes after a fixed number of non-I/O addresses.

The simulator supports simulation of round-robin multiprogramming with up to a compile-time constant of separate input traces. Task switches are modeled to occur after a fixed number of memory references. This interval is called the inter-task switch time. This is similar to but not exactly the same as a fixed number of instructions.

A reference to the cache by a process after a context switch should not accidentally hit on data left in the cache by former process. A cache can protect against this by flushing all valid data following a context switch or prepending a process id on all data that is in the cache. Dinero supports both the flush and no-flush options.

3.4.2.4. Input Specifications

Input specifications (Table 4) provide information to the simulator on the input trace record format. This information is the address size, the type of address, the file format, and the maximum number of addresses to be processed.

The address size defaults to 22 bits; this can be varied by a small implementation-dependent amount.

Displacement is the number of binary zeros that must be appended to input addresses to recreate byte-addressing. 32-bit-word addresses have a displacement of 2, 16-bit words have displacement of 1, and byte addressing has displacement of 0.

The input trace file format can be either ASCII or binary.

The simulation can run until one of the traces is exhausted or for a fixed number of addresses called the count.

3.5. Traces Studied

This section presents the traces used to produce the results of this paper. They are a normal production programs. No synthetic benchmarks have been used. See Tables 5 to 8.

OPSYS, PLOT, SIMP, TRACE, ROFF, and ED are the traces used in PDP-11 results (Table 5). OPSYS (C) is a trace of a program that implements a toy operating system. PLOT (Fortran) is a trace of a printer plotter program. SIMP (Fortran) is a trace of pipeline simulation
program. TRACE (PDP-11 Assembly) is a trace of the tracing program tracing ED. ROFF (PDP-11 Assembly) is a trace of a text output and formatting program. ED (C) is a trace of a text editor.

CPP, C1, C2, OD, GREP, SORT, LS, and PR are Unix programs ported to the Z8000 from the PDP-11 (Table 6). CPP, C1, C2, the first three of five passes of the Z8000 C compiler, were used in the load-forward work. OD, GREP, SORT, LS, NM, and PR are UNIX utility programs written in C. They are used in the other Z8000 averages. Since these programs are small and believe to be compiled with a poor compiler, they may yield optimistic results. OD dumps octal files in ASCII. GREP searches a file for a pattern. SORT is a sorting package. LS lists files. NM prints a specially compiled object file's symbol table. PR formats text files for printing.

FGO1, FGO2, FGO3 (Fortran Go Steps), and FCOMP1 (Fortran Compile) are from a scientific workload with 4-byte datapath (Table 7). FGO2 and FCOMP1 are used in the IBM System/360 Model 85 comparisons. FGO1 and FGO3 are used in other FORTRAN results. FGO1 is a single-precision factor analysis. FGO2 is a double-precision analysis of satellite information. FGO3 is a double-precision numerical analysis program. FCOMP1 is a compile of a program that solves Reynolds partial differential equations. All of FGO1, FGO3, and FGO3 were compiled using the IBM Fortran G compiler.

CGO1, CGO3 (Cobol Go Steps), and PGO2 (PL/I Go Step) are from a business workload (Table 8). CGO1, CGO3, and PGO2 are used in the IBM System/360 Model 85 comparisons. CGO1 and CGO3 are used in other COBOL results. CGO1 is a fixed-assets program doing tax transaction selection. CGO3 projects depreciation of assets. PGO2 does CCW analysis.

PGO1 (PL/I) represents miscellaneous workload and was used in the IBM System/360 Model 85 comparisons.
### Table 6. Z8000/Unix Workload

<table>
<thead>
<tr>
<th>Trace</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPP</td>
<td>First phase of C compiler; C; used in load-forward results.</td>
</tr>
<tr>
<td>C1</td>
<td>Second phase of C compiler; C; used in load-forward results.</td>
</tr>
<tr>
<td>C2</td>
<td>Third phase of C compiler; C; used in load-forward results.</td>
</tr>
<tr>
<td>OD</td>
<td>Unix utility for dumping files in ASCII; C; used in Z8000 results.</td>
</tr>
<tr>
<td>GREP</td>
<td>Unix utility for string searching; C; used in Z8000 results.</td>
</tr>
<tr>
<td>SORT</td>
<td>Unix utility for sorting; C; used in Z8000 results.</td>
</tr>
<tr>
<td>LS</td>
<td>Unix utility for listing files; C; used in Z8000 results.</td>
</tr>
<tr>
<td>NM</td>
<td>Unix utility for printing a specially compiled object file's symbol table; C; used in Z8000 results.</td>
</tr>
<tr>
<td>PR</td>
<td>Unix utility for formatting text files for printing; C; used in Z8000 results.</td>
</tr>
</tbody>
</table>

### Table 7. Scientific Workload

<table>
<thead>
<tr>
<th>Trace</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>FGO1</td>
<td>Single precision factor analysis; Fortran go step compiled with IBM Fortran G compiler; used in Fortran results.</td>
</tr>
<tr>
<td>FGO2</td>
<td>Double-precision analysis of satellite information; Fortran go step compiled with IBM Fortran G compiler; used in IBM System/360 Model 85 comparisons.</td>
</tr>
<tr>
<td>FGO3</td>
<td>Double-precision numerical analysis program; Fortran go step compiled with IBM Fortran G compiler; used in Fortran results.</td>
</tr>
<tr>
<td>FCOMP</td>
<td>Compile of a program that solves Reynolds partial differential equation; used in IBM/360 Model 85 comparisons.</td>
</tr>
</tbody>
</table>

### Table 8. Business Workload

<table>
<thead>
<tr>
<th>Trace</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGO1</td>
<td>Fixed-assets program doing tax transaction selection; Cobol Go Step; used in Cobol results and IBM/360 Model 85 comparisons.</td>
</tr>
<tr>
<td>CGO3</td>
<td>Program projects depreciation of assets; Cobol Go Step; used in Cobol results and IBM/360 Model 85 comparisons.</td>
</tr>
<tr>
<td>PGO1</td>
<td>PL/1 Go Step; used in IBM/360 Model 85 comparisons.</td>
</tr>
<tr>
<td>PGO2</td>
<td>Program does CCW analysis; PL/1 Go Step; used in IBM/360 Model 85 comparisons.</td>
</tr>
</tbody>
</table>

### Table 9. IBM System/360 Model 85 Results

<table>
<thead>
<tr>
<th>Cache Organization</th>
<th>Miss Ratio</th>
<th>Relative to 360/85</th>
</tr>
</thead>
<tbody>
<tr>
<td>360/85</td>
<td>0.0258</td>
<td>1.000</td>
</tr>
<tr>
<td>4-way</td>
<td>0.0088</td>
<td>0.341</td>
</tr>
<tr>
<td>8-way</td>
<td>0.0081</td>
<td>0.314</td>
</tr>
<tr>
<td>16-way</td>
<td>0.0076</td>
<td>0.294</td>
</tr>
</tbody>
</table>

*Based on six IBM System/360 programs: 1 Fortran Go Step, 1 Fortran Compile, 2 Cobol Go Steps, and 2 PL/1 Go Steps.*
4. Results

This section presents the results of trace-driven simulations of small memory caches with various block and sub-block sizes. First, the original use of sub-block placement in the IBM System/360 Model 85 (the first machine with a cache memory) is examined. Next, the results from PDP-11 and Z8000 traces show small caches perform well. Caches are shown do work well with nibble and paged-mode memories, and a way of prefetching sub-blocks called load-forward is introduced. Last, preliminary Fortran and Cobol runs show Fortran performs well and Cobol less well.

4.1. Early Sub-Block Placement

The first memory cache implementation was done in the IBM System/360 Model 85 [2]. The designers felt that it was not feasible to build a main memory that could keep up with Model 85's 80 nanosecond cycle time. Putting a cache between memory and the processor allowed designers to use older main storage units with access times of 1.04 microseconds. The 360/85 Cache is 16 K bytes divided into 16 blocks of 1 K bytes each. Each block was divided into 16 sub-blocks of 64 bytes. Data was transferred between main memory and the cache in single sub-block units by a four-way interleaved access across a 16-byte datapath, beginning with the address the processor requested.

The unit of placement within the 360/85 cache was the large 1-Kbyte block. Whenever a request was made to a sub-block within a block not in the cache, the block least recently used was invalidated, and a memory was started to get the requested sub-block. This placement scheme is less flexible than allowing the 256 sub-blocks to come from anywhere in the address space. The rigidity is harmful if much of the 1 K block is never accessed, but it only requires one address tag per block. Thus the associative search logic, built in the limited technology of the day, needed to operate on only 16 entries.

The technology used to build mainframe computers has changed greatly in the last fifteen years, including making associative search logic comparatively less expensive. To our knowledge, there are no public studies comparing the 360/85's mapping scheme to today's set-associative mapping. Before we resurrect this idea for on-chip caches, we will show that the idea performs poorly for mainframe caches. Figure 1 (Table 9) compares the 360/85 to 4-, 8-, and 16-way set associative mappings for several System/360 benchmarks. Our 360/85 miss ratio numbers are consistent with Liptay's [2] finding for the average miss ratio of 0.032. The average miss ratio for the 360/85 cache is three times greater than the miss ratio for a 4-way set associative cache with LRU replacement using 64-byte blocks for placement and transfer. The reason the 360/85 cache performs poorly is data can only be resident in the cache from one of 16 blocks that are much too
large (1 Kbytes). In fact, on the average 72 percent of the sub-blocks in a block, 11.52 of 16 sub-blocks, are never referenced in the period a block is resident.

4.2. PDP-11 Results

The results for the balance of Section 4 examine miss ratio, traffic ratio, and gross cache size for small caches with various block and sub-block sizes. The miss ratio, the number of cache misses divided by the number of cache accesses, is a measure of the effectiveness of a cache in reducing memory access latency. The traffic ratio, the number of bytes transferred to the cache divided by the number of bytes transferred to a processor without a cache, is a measure of the effectiveness of a cache in reducing memory bandwidth. The gross cache size, the size of the tag and data area of the cache, is a measure of the cost of a cache. Gross size is used because the different schemes presented here require radically varying amounts of address tag area.

Three categories of two-dimensional figures illustrate results: Miss Ratio vs. Gross Cache Size (e.g. Figure 2), Traffic Ratio vs. Gross Cache Size (e.g. Figure 5), and Miss Ratio vs. Traffic Ratio (e.g. Figure 7). Figures in the first two categories contain lines that span a range of gross cache sizes. These lines are labeled $bz-sy$ where $z$ is the block size in bytes and $y$ is the sub-block size in bytes. Figures in the last category show the miss ratio versus the traffic ratio. The labeled regions enclose caches of a single net (data) cache size. (Net cache size is related, but not the same as, the gross cache size). Within these regions, solid lines connect the set of caches with block size $bz$, and dashed lines connect the set of caches with sub-block size $sy$. A cache with block size $bz$ and sub-block size $sy$ is found at the intersection the solid line labeled $bz$ and the dashed line labeled $sy$. For example, in Figure 7, the 512-byte cache with block size 16 and sub-block size 4 is found at the intersection of $b16$ and $s4$ in the region labeled 512.

Six 1 million-address DEC PDP-11 runs have been run individually and averaged together to produce the results in Thus, the results quoted are warm-start ratios. Warm-start ratios do not count the misses taken to initially fill the cache with relevant data. Tables 10 and 11 and Figures 2 through 10. Figure 2 shows results from caches with sub-block size of 4 bytes (transfer size). Storage is allocated in the cache according to the block size that varies from 4 to 64 bytes. Block sizes of 4, 8, and 16 bytes work comparably. One would normally suspect that the larger block sizes should have higher miss ratios than small blocks because the mappings in the cache are more constrained. This can be verified by plotting the miss ratios against the net cache size (the size of the data only). Here, larger blocks reduce the tag bits required for a cache, so plots of miss ratios of large blocks versus gross cache size are shifted to the left. Thus, when curves of two different block sizes are superimposed, the effect of the constrained mapping is undone by the savings in tag area.
Figure 3 shows miss ratios for runs with a block size of 32 bytes. Transfers are made in sub-block sizes from 2 to 32 bytes. The number of misses increases when the sub-block size is decreased because it takes more misses to bring in the same amount of data. Sometimes decreasing the main memory transfer size will not cause more misses because the decreased size will reduce the amount of data prematurely replaced (memory pollution). Here, the misses will increase because it is the block size, not the sub-block size, that determines the degree of memory pollution. A whole block is invalidated when replacement is necessary. Therefore, all curves in the figure have the same memory pollution characteristic, that is, all invalidate 32 bytes when a block-miss occurs.

Figure 4 shows the miss ratios for runs where the sub-block size is equal to the block size as is the case for most cache memories today. The b2-s2 performs worse than the b32-s2 curve of Figure 3 because of the large amount of tag area required by 2-byte blocks, but, for a given net size, b2-s2 has a lower miss ratio than b32-s2 (see Tables 10 and 11).

Figures 5 and 6 show the traffic ratio vs. gross cache size for the PDP-11 runs. Figure 5 looks at runs with a constant sub-block size of 4 bytes. The best block size for minimum bus traffic changes for different values of gross cache size. The block size of the minimum curve varies from 4 to 8 to 16 as gross cache size is increased. Still, all three curves are close together. Once again, plotting against gross cache size moves caches with more constrained, larger blocks to the left, allowing three of the curves to be near each other.

Figure 6 shows runs with constant block size. At a given cache size, the smaller the sub-block size, the lower the bus traffic because the chance of bringing in data that is never used is reduced.

Figures 7 and 8 show miss ratio vs. traffic ratio for different net cache sizes. Figure 7 shows results for net cache size of 32, 128, and 512 bytes; Figure 8 shows results for net cache size of 64, 256, and 1024 bytes. Some organizations of caches actually increase the bus traffic, i.e. their traffic ratio is greater than 1.0. For example, the 128-byte cache (net) in Figure 7 with block and sub-block size of 16 bytes almost doubles bus traffic. This happens because the sub-block size is too large. Caches alter the traffic ratio through two competing effects. They reduce the traffic ratio whenever many references are made to a word already in the cache. They increase the traffic ratio whenever some word in a sub-block brought into the cache is never referenced. Thus, the traffic ratio for a cache with \( w \) words per sub-block is greater than 1.0 whenever the average sub-block is referenced less than \( w \) times whenever it is resident in the cache.

Changing cache block size significantly affects gross cache size if cache blocks are already small. This is because doubling the block size halves the total address tag area which is a large fraction of cache area if blocks are small. For example, the 512-byte cache with block and sub-block size of 2-bytes (b2-s2) occupies 50.0 percent more area than the 512-byte cache with 4-byte
<table>
<thead>
<tr>
<th>Net, Block, Gross Cache Block Size</th>
<th>Miss Traffic Ratio</th>
<th>Traffic Ratio</th>
<th>Traffic Ratio (paged)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>32</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>47 8,4</td>
<td>0.580</td>
<td>1.160</td>
<td>0.773</td>
</tr>
<tr>
<td>63 4,4</td>
<td>0.546</td>
<td>1.092</td>
<td>0.728</td>
</tr>
<tr>
<td>64 4,2</td>
<td>0.846</td>
<td>0.846</td>
<td>0.846</td>
</tr>
<tr>
<td>96 2,2</td>
<td>0.824</td>
<td>0.824</td>
<td>0.824</td>
</tr>
<tr>
<td><strong>128</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>157 16,16</td>
<td>0.215</td>
<td>1.720</td>
<td>0.717</td>
</tr>
<tr>
<td>158 16,8</td>
<td>0.291</td>
<td>1.164</td>
<td>0.582</td>
</tr>
<tr>
<td>160 16,4</td>
<td>0.418</td>
<td>0.836</td>
<td>0.557</td>
</tr>
<tr>
<td>164 16,2</td>
<td>0.649</td>
<td>0.649</td>
<td>0.649</td>
</tr>
<tr>
<td>188 8,8</td>
<td>0.251</td>
<td>1.004</td>
<td>0.502</td>
</tr>
<tr>
<td>190 8,4</td>
<td>0.367</td>
<td>0.734</td>
<td>0.489</td>
</tr>
<tr>
<td>194 8,2</td>
<td>0.578</td>
<td>0.578</td>
<td>0.578</td>
</tr>
<tr>
<td>252 4,4</td>
<td>0.320</td>
<td>0.640</td>
<td>0.427</td>
</tr>
<tr>
<td>256 4,2</td>
<td>0.509</td>
<td>0.509</td>
<td>0.509</td>
</tr>
<tr>
<td>384 2,2</td>
<td>0.461</td>
<td>0.461</td>
<td>0.461</td>
</tr>
<tr>
<td><strong>512</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>542 64,16</td>
<td>0.164</td>
<td>1.312</td>
<td>0.547</td>
</tr>
<tr>
<td>546 64,8</td>
<td>0.229</td>
<td>0.916</td>
<td>0.458</td>
</tr>
<tr>
<td>554 64,4</td>
<td>0.335</td>
<td>0.670</td>
<td>0.447</td>
</tr>
<tr>
<td>568 32,32</td>
<td>0.081</td>
<td>1.299</td>
<td>0.487</td>
</tr>
<tr>
<td>570 32,16</td>
<td>0.112</td>
<td>0.896</td>
<td>0.373</td>
</tr>
<tr>
<td>574 32,8</td>
<td>0.164</td>
<td>0.656</td>
<td>0.328</td>
</tr>
<tr>
<td>582 32,4</td>
<td>0.248</td>
<td>0.496</td>
<td>0.331</td>
</tr>
<tr>
<td>598 32,2</td>
<td>0.402</td>
<td>0.402</td>
<td>0.402</td>
</tr>
<tr>
<td>628 16,16</td>
<td>0.078</td>
<td>0.626</td>
<td>0.261</td>
</tr>
<tr>
<td>632 16,8</td>
<td>0.116</td>
<td>0.464</td>
<td>0.232</td>
</tr>
<tr>
<td>640 16,4</td>
<td>0.179</td>
<td>0.358</td>
<td>0.239</td>
</tr>
<tr>
<td>656 16,2</td>
<td>0.290</td>
<td>0.290</td>
<td>0.290</td>
</tr>
<tr>
<td>752 8,8</td>
<td>0.087</td>
<td>0.348</td>
<td>0.174</td>
</tr>
<tr>
<td>760 8,4</td>
<td>0.137</td>
<td>0.274</td>
<td>0.183</td>
</tr>
<tr>
<td>776 8,2</td>
<td>0.223</td>
<td>0.223</td>
<td>0.223</td>
</tr>
<tr>
<td>1008 4,4</td>
<td>0.102</td>
<td>0.204</td>
<td>0.136</td>
</tr>
<tr>
<td>1024 4,2</td>
<td>0.168</td>
<td>0.168</td>
<td>0.168</td>
</tr>
<tr>
<td>1536 2,2</td>
<td>0.140</td>
<td>0.140</td>
<td>0.140</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Net, Block, Gross Cache Block Size</th>
<th>Miss Traffic Ratio</th>
<th>Traffic Ratio</th>
<th>Traffic Ratio (paged)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>64</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64 16,8</td>
<td>0.399</td>
<td>1.596</td>
<td>0.798</td>
</tr>
<tr>
<td>80 16,4</td>
<td>0.557</td>
<td>1.114</td>
<td>0.743</td>
</tr>
<tr>
<td>94 8,8</td>
<td>0.339</td>
<td>1.356</td>
<td>0.678</td>
</tr>
<tr>
<td>95 8,4</td>
<td>0.479</td>
<td>0.958</td>
<td>0.639</td>
</tr>
<tr>
<td>97 8,2</td>
<td>0.739</td>
<td>0.739</td>
<td>0.739</td>
</tr>
<tr>
<td>126 4,4</td>
<td>0.425</td>
<td>0.850</td>
<td>0.567</td>
</tr>
<tr>
<td>128 4,2</td>
<td>0.666</td>
<td>0.666</td>
<td>0.666</td>
</tr>
<tr>
<td>192 2,2</td>
<td>0.620</td>
<td>0.620</td>
<td>0.620</td>
</tr>
<tr>
<td><strong>256</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>284 32,32</td>
<td>0.146</td>
<td>2.336</td>
<td>0.876</td>
</tr>
<tr>
<td>285 32,16</td>
<td>0.191</td>
<td>1.528</td>
<td>0.637</td>
</tr>
<tr>
<td>287 32,8</td>
<td>0.291</td>
<td>1.164</td>
<td>0.582</td>
</tr>
<tr>
<td>291 32,4</td>
<td>0.418</td>
<td>0.836</td>
<td>0.557</td>
</tr>
<tr>
<td>299 32,2</td>
<td>0.599</td>
<td>0.599</td>
<td>0.599</td>
</tr>
<tr>
<td>314 16,16</td>
<td>0.144</td>
<td>1.152</td>
<td>0.480</td>
</tr>
<tr>
<td>316 16,8</td>
<td>0.294</td>
<td>0.816</td>
<td>0.408</td>
</tr>
<tr>
<td>320 16,4</td>
<td>0.302</td>
<td>0.604</td>
<td>0.403</td>
</tr>
<tr>
<td>328 16,2</td>
<td>0.478</td>
<td>0.478</td>
<td>0.478</td>
</tr>
<tr>
<td>376 8,8</td>
<td>0.168</td>
<td>0.672</td>
<td>0.336</td>
</tr>
<tr>
<td>380 8,4</td>
<td>0.254</td>
<td>0.508</td>
<td>0.339</td>
</tr>
<tr>
<td>388 8,2</td>
<td>0.407</td>
<td>0.407</td>
<td>0.407</td>
</tr>
<tr>
<td>504 4,4</td>
<td>0.218</td>
<td>0.436</td>
<td>0.291</td>
</tr>
<tr>
<td>512 4,2</td>
<td>0.351</td>
<td>0.351</td>
<td>0.351</td>
</tr>
<tr>
<td>768 2,2</td>
<td>0.297</td>
<td>0.297</td>
<td>0.297</td>
</tr>
<tr>
<td><strong>1024</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1084 64,16</td>
<td>0.081</td>
<td>0.646</td>
<td>0.269</td>
</tr>
<tr>
<td>1092 64,8</td>
<td>0.118</td>
<td>0.472</td>
<td>0.236</td>
</tr>
<tr>
<td>1108 64,4</td>
<td>0.178</td>
<td>0.356</td>
<td>0.237</td>
</tr>
<tr>
<td>1136 32,32</td>
<td>0.033</td>
<td>0.533</td>
<td>0.200</td>
</tr>
<tr>
<td>1140 32,16</td>
<td>0.049</td>
<td>0.391</td>
<td>0.163</td>
</tr>
<tr>
<td>1148 32,8</td>
<td>0.075</td>
<td>0.298</td>
<td>0.149</td>
</tr>
<tr>
<td>1164 32,4</td>
<td>0.116</td>
<td>0.232</td>
<td>0.155</td>
</tr>
<tr>
<td>1196 32,2</td>
<td>0.190</td>
<td>0.190</td>
<td>0.190</td>
</tr>
<tr>
<td>1256 16,16</td>
<td>0.033</td>
<td>0.265</td>
<td>0.110</td>
</tr>
<tr>
<td>1264 16,8</td>
<td>0.052</td>
<td>0.206</td>
<td>0.103</td>
</tr>
<tr>
<td>1280 16,4</td>
<td>0.081</td>
<td>0.162</td>
<td>0.108</td>
</tr>
<tr>
<td>1312 16,2</td>
<td>0.133</td>
<td>0.133</td>
<td>0.133</td>
</tr>
<tr>
<td>1504 8,8</td>
<td>0.039</td>
<td>0.156</td>
<td>0.078</td>
</tr>
<tr>
<td>1520 8,4</td>
<td>0.061</td>
<td>0.122</td>
<td>0.081</td>
</tr>
<tr>
<td>1552 8,2</td>
<td>0.101</td>
<td>0.101</td>
<td>0.101</td>
</tr>
<tr>
<td>2016 4,4</td>
<td>0.048</td>
<td>0.096</td>
<td>0.054</td>
</tr>
<tr>
<td>2048 4,2</td>
<td>0.081</td>
<td>0.081</td>
<td>0.081</td>
</tr>
<tr>
<td>3072 2,2</td>
<td>0.072</td>
<td>0.072</td>
<td>0.072</td>
</tr>
</tbody>
</table>
Figure 1. IBM System/360 Model 85-like Caches. The 360/85 has 1 Kbytes blocks and 64 byte sub-blocks. The other organizations use 4-, 8- and 16-way set-associative LRU replacement of 64-byte blocks.

Figure 2. PDP-11 4-byte Sub-Block Miss Ratios. Lines connect caches of a block size $bz$ and sub-block size 4 bytes ($sz$) for varying gross (address tags and data) cache sizes.

Figure 3. PDP-11 32-byte Block Miss Ratios. Lines connect caches of a block size 32 bytes ($b32$) and sub-block size $sz$ for varying gross (address tags and data) cache sizes.

Figure 4. PDP-11 Block Size Equal Sub-Block Size Miss Ratios. Lines connect caches of a block size $bz$ and equal sub-block size $sz$ for varying gross (address tags and data) cache sizes.
Figure 5. PDP-11 4-byte Sub-Block Traffic Ratios. Lines connect caches of a block size $b_k$ and sub-block size $s_4$ for varying gross (address tags and data) cache sizes.

Figure 6. PDP-11 32-byte Block Traffic Ratios. Lines connect caches of a block size 32 bytes ($b_{32}$) and sub-block size $s_8$ for varying gross (address tags and data) cache sizes.

Figure 7. PDP-11 Results-Part I. Results for caches with net sizes of 32, 128, and 512 bytes. Solid lines connect caches with constant block size $b_k$. Dashed lines connect caches with constant sub-block size $s_k$.

Figure 8. PDP-11 Results-Part II. Results for caches with net sizes of 64, 256, and 1024 bytes. Solid lines connect caches with constant block size $b_k$. Dashed lines connect caches with constant sub-block size $s_k$. 
blocks and 2-byte sub-blocks \(b4-s2\) (Table 10: 1536 bytes vs. 1024 bytes). Yet, it only performs
16.7 percent better in miss ratio and 16.9 percent better in traffic ratio. Therefore, the \(b4-s2\)-cache is more cost-effective.

Changing cache sub-block size does not significantly affect gross cache size. In fact, dou-
bling the sub-block size only halves the number of sub-block-valid bits in a cache block. This does
not significantly affect cache size because the number of sub-block-valid bits in a block is small
compared to the tag and data area. For example, going from a \(b32-s4\) to a \(b32-s8\)-cache
decreases the total size by only 1.4 percent. A cache with the capability of varying sub-block size
can be set to run at different operating points depending on the relative importance of miss ratio
and traffic ratio. Such a cache requires somewhat more control and enough sub-block-valid bits
for the the smallest sub-block size. For example, Figure 8 shows a solid line labeled \(b32\) with net
cache size 1024 that intersects all sub-block sizes between 2 and 32 bytes. These various sub-
block sizes allow a system implementor to trade the miss ratio against the traffic ratio. In a sys-
tem with considerable unused bus bandwidth, the sub-block size could be set to 32 bytes to real-
ize miss and traffic ratios of 0.033 and 0.542 (Table 11). In another system that is bus-limited
either by a slower bus or more devices (processors) using the bus, the sub-block size could be set
as low as 2 bytes. This would increase the miss ratio by a factor of 6 to 0.190 but decrease the
traffic ratio by a factor of 3 to 0.197.

These results show that small caches can be effective at reducing delay, off-chip bandwidth,
or both. Designers can trade off miss ratio and traffic ratio by varying the sub-block size. Once
the sub-block size has been fixed in a cache implementation, the traffic ratio is the miss ratio
times \(w\), where \(w\) is the number of words in a sub-block.

4.3. Paged-Mode PDP-11 Results

The above traffic ratio results assume that the cost of a memory access is directly propor-
tional to the number of bytes read. This assumption is accurate for many current microprocessor
systems, often because of bus protocol compatibility constraints. However, some new memory
chips provide paged or nibble addressing modes. These modes reduce the average access time if
several words are simultaneously fetched from adjacent locations. On-chip caches can exploit
these addressing modes for loading sub-blocks. Caches with larger sub-blocks derive more benefit
than those with smaller sub-blocks. Burns [21] reports that typical access times are 160 ns for
the first word and 55 ns for subsequent words. If we approximate the ratio of 160 to 55 as 3 to 1
and assign unit-cost to getting one word, then the cost of getting \(s\) sequential words is
\[1 + \frac{1}{3}(s - 1)\]. A cache with a sub-block size of \(s\) words will reduce memory costs by a factor
\(g(s)\) where
Figure 9. PDP-11 Results for Paged Mode Part I. Results for caches with net sizes of 32, 128, and 512 bytes. Paged-mode assumes that an access for $s$ sequential words costs $1 + \frac{1}{3}(s - 1)$.

Figure 10. PDP-11 Results for Paged Mode Part II. Results for caches with net sizes of 64, 256, and 1024 bytes. Paged-mode assumes that an access for $s$ sequential words costs $1 + \frac{1}{3}(s - 1)$.
\[ g(s) = \frac{1}{s} \left[ 1 + \frac{1}{3} (s - 1) \right] \]

because the cache always fetches \( s \) sequential words on a miss. The standard traffic ratio multiplied by the above factor produces a scaled traffic ratio for paged-mode memories. This equation also reflects the cost of using a bus cycle to present an address to the memory before waiting 105 ns for the first word and 55 ns for subsequent words.

Figures 9 and 10 show PDP-11 results scaled to reflect the economies-of-scale of transferring more than one 16-bit word in a single access. Scaling with 16-bit words should put an upper bound on the improvement that can be expected from scaling with 32-bit words. Curves of constant block size and varying sub-block size (solid lines) minimize the traffic ratio at a sub-block size of 4 or 8 bytes rather than 2 bytes result from the standard memory interface. Since, the traffic ratio penalty for bringing in additional data is smaller than in the case of a standard memory interface, a cache designed for these addressing modes will tend to have a larger sub-block size.

4.4. Z8000 Results

Figures 11 through 17 and Tables 11 and 13 show results from five UNIX utilities written in C and compiled for the Z8000. These traces yield better performance than the PDP-11 traces; they are plotted on the same scale for comparison. Figures 11 through 15 show miss ratio vs. gross cache size for: sub-block sizes of 2 and 4 bytes, block sizes of 8 and 32 bytes, and block size equal to sub-block size. The results quoted are warm-start ratios. These ratios are slightly optimistic if some of the utilities do not run long enough to amortize the initial misses. All runs with a sub-block size smaller than the block size were performed with a 4-way set-associative cache. Some runs with a sub-block size equal to the block size were run with greater associativity, so that multiple caches could be simulated concurrently.

4.5. Load-Forward Results

Load-forward is a mechanism for combining the miss ratio benefits of a large block size with the low bus traffic of sub-block placement. It is being used in the 256-byte, on-chip cache of the Z80,000 [22,23]. The Z80,000 cache has 16 blocks of 16 bytes. The blocks are replaced using an LRU stack on-chip. Data is fetched from memory in one-word (two-byte) sub-blocks with an option to load-forward (to be described).

Program and data references within a cache block exhibit a forward bias. A program typically branches to a location within a cache block, proceeds sequentially forward, and then branches again. Data references also tend to proceed forward because of processing of arrays, character strings, and individual variables whose storage is defined by the programmer in order of
<table>
<thead>
<tr>
<th>Net, Gross, Block, Cache Size</th>
<th>Miss Ratio</th>
<th>Traffic Ratio</th>
<th>Traffic Ratio (paged)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 Block, 8.4 Sub-Block Size</td>
<td>0.539</td>
<td>1.078</td>
<td>0.719</td>
</tr>
<tr>
<td>47 Block, 8.2</td>
<td>0.905</td>
<td>0.905</td>
<td>0.905</td>
</tr>
<tr>
<td>63 Block, 4.4</td>
<td>0.506</td>
<td>1.012</td>
<td>0.675</td>
</tr>
<tr>
<td>64 Block, 4.2</td>
<td>0.855</td>
<td>0.855</td>
<td>0.855</td>
</tr>
<tr>
<td>96 Block, 2.2</td>
<td>0.850</td>
<td>0.850</td>
<td>0.850</td>
</tr>
<tr>
<td>128 Block, 16.8</td>
<td>0.206</td>
<td>0.824</td>
<td>0.412</td>
</tr>
<tr>
<td>160 Block, 16.4</td>
<td>0.321</td>
<td>0.642</td>
<td>0.428</td>
</tr>
<tr>
<td>164 Block, 16.2</td>
<td>0.544</td>
<td>0.544</td>
<td>0.544</td>
</tr>
<tr>
<td>188 Block, 8.8</td>
<td>0.157</td>
<td>0.628</td>
<td>0.314</td>
</tr>
<tr>
<td>194 Block, 8.2</td>
<td>0.257</td>
<td>0.514</td>
<td>0.343</td>
</tr>
<tr>
<td>252 Block, 4.4</td>
<td>0.222</td>
<td>0.444</td>
<td>0.296</td>
</tr>
<tr>
<td>256 Block, 4.2</td>
<td>0.392</td>
<td>0.392</td>
<td>0.392</td>
</tr>
<tr>
<td>384 Block, 2.2</td>
<td>0.361</td>
<td>0.361</td>
<td>0.361</td>
</tr>
<tr>
<td>512 Block, 64.16</td>
<td>0.089</td>
<td>0.712</td>
<td>0.297</td>
</tr>
<tr>
<td>542 Block, 64.8</td>
<td>0.134</td>
<td>0.536</td>
<td>0.268</td>
</tr>
<tr>
<td>546 Block, 64.4</td>
<td>0.216</td>
<td>0.432</td>
<td>0.288</td>
</tr>
<tr>
<td>554 Block, 64.2</td>
<td>0.380</td>
<td>0.380</td>
<td>0.380</td>
</tr>
<tr>
<td>570 Block, 32,32</td>
<td>0.042</td>
<td>0.672</td>
<td>0.252</td>
</tr>
<tr>
<td>570 Block, 32,16</td>
<td>0.062</td>
<td>0.496</td>
<td>0.207</td>
</tr>
<tr>
<td>574 Block, 32,8</td>
<td>0.095</td>
<td>0.380</td>
<td>0.190</td>
</tr>
<tr>
<td>582 Block, 32,4</td>
<td>0.158</td>
<td>0.316</td>
<td>0.211</td>
</tr>
<tr>
<td>598 Block, 32,2</td>
<td>0.283</td>
<td>0.283</td>
<td>0.283</td>
</tr>
<tr>
<td>628 Block, 16.16</td>
<td>0.045</td>
<td>0.360</td>
<td>0.150</td>
</tr>
<tr>
<td>632 Block, 16.8</td>
<td>0.070</td>
<td>0.280</td>
<td>0.140</td>
</tr>
<tr>
<td>640 Block, 16.4</td>
<td>0.117</td>
<td>0.234</td>
<td>0.156</td>
</tr>
<tr>
<td>656 Block, 16.2</td>
<td>0.211</td>
<td>0.211</td>
<td>0.211</td>
</tr>
<tr>
<td>752 Block, 8.8</td>
<td>0.056</td>
<td>0.224</td>
<td>0.112</td>
</tr>
<tr>
<td>760 Block, 8.4</td>
<td>0.093</td>
<td>0.186</td>
<td>0.124</td>
</tr>
<tr>
<td>776 Block, 8.2</td>
<td>0.170</td>
<td>0.170</td>
<td>0.170</td>
</tr>
<tr>
<td>1008 Block, 4.4</td>
<td>0.075</td>
<td>0.150</td>
<td>0.100</td>
</tr>
<tr>
<td>1024 Block, 4.2</td>
<td>0.136</td>
<td>0.136</td>
<td>0.136</td>
</tr>
<tr>
<td>1536 Block, 2.2</td>
<td>0.120</td>
<td>0.120</td>
<td>0.120</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Net, Gross, Block, Cache Size</th>
<th>Miss Ratio</th>
<th>Traffic Ratio</th>
<th>Traffic Ratio (paged)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 Block, 16.8</td>
<td>0.330</td>
<td>1.320</td>
<td>0.660</td>
</tr>
<tr>
<td>79 Block, 16.4</td>
<td>0.508</td>
<td>1.016</td>
<td>0.677</td>
</tr>
<tr>
<td>80 Block, 16.2</td>
<td>0.857</td>
<td>0.857</td>
<td>0.857</td>
</tr>
<tr>
<td>94 Block, 8.8</td>
<td>0.298</td>
<td>1.192</td>
<td>0.596</td>
</tr>
<tr>
<td>95 Block, 8.4</td>
<td>0.461</td>
<td>0.922</td>
<td>0.615</td>
</tr>
<tr>
<td>97 Block, 8.2</td>
<td>0.762</td>
<td>0.762</td>
<td>0.762</td>
</tr>
<tr>
<td>126 Block, 4.4</td>
<td>0.432</td>
<td>0.864</td>
<td>0.576</td>
</tr>
<tr>
<td>128 Block, 4.2</td>
<td>0.671</td>
<td>0.671</td>
<td>0.671</td>
</tr>
<tr>
<td>192 Block, 2.2</td>
<td>0.583</td>
<td>0.583</td>
<td>0.583</td>
</tr>
<tr>
<td>256 Block, 32,32</td>
<td>0.079</td>
<td>1.264</td>
<td>0.474</td>
</tr>
<tr>
<td>284 Block, 32,16</td>
<td>0.107</td>
<td>0.856</td>
<td>0.357</td>
</tr>
<tr>
<td>285 Block, 32,8</td>
<td>0.156</td>
<td>0.624</td>
<td>0.312</td>
</tr>
<tr>
<td>291 Block, 32,4</td>
<td>0.245</td>
<td>0.490</td>
<td>0.327</td>
</tr>
<tr>
<td>299 Block, 32,2</td>
<td>0.421</td>
<td>0.421</td>
<td>0.421</td>
</tr>
<tr>
<td>314 Block, 16,16</td>
<td>0.082</td>
<td>0.656</td>
<td>0.273</td>
</tr>
<tr>
<td>316 Block, 16,8</td>
<td>0.124</td>
<td>0.496</td>
<td>0.248</td>
</tr>
<tr>
<td>320 Block, 16,4</td>
<td>0.203</td>
<td>0.406</td>
<td>0.271</td>
</tr>
<tr>
<td>328 Block, 16,2</td>
<td>0.355</td>
<td>0.355</td>
<td>0.355</td>
</tr>
<tr>
<td>376 Block, 8,8</td>
<td>0.108</td>
<td>0.432</td>
<td>0.216</td>
</tr>
<tr>
<td>380 Block, 8,4</td>
<td>0.175</td>
<td>0.350</td>
<td>0.233</td>
</tr>
<tr>
<td>388 Block, 8,2</td>
<td>0.312</td>
<td>0.312</td>
<td>0.312</td>
</tr>
<tr>
<td>504 Block, 4,4</td>
<td>0.157</td>
<td>0.314</td>
<td>0.209</td>
</tr>
<tr>
<td>512 Block, 4,2</td>
<td>0.287</td>
<td>0.287</td>
<td>0.287</td>
</tr>
<tr>
<td>768 Block, 2,2</td>
<td>0.273</td>
<td>0.273</td>
<td>0.273</td>
</tr>
<tr>
<td>1024 Block, 64,16</td>
<td>0.041</td>
<td>0.328</td>
<td>0.137</td>
</tr>
<tr>
<td>1084 Block, 64,8</td>
<td>0.063</td>
<td>0.252</td>
<td>0.126</td>
</tr>
<tr>
<td>1092 Block, 64,4</td>
<td>0.104</td>
<td>0.208</td>
<td>0.139</td>
</tr>
<tr>
<td>1108 Block, 64,2</td>
<td>0.104</td>
<td>0.104</td>
<td>0.104</td>
</tr>
<tr>
<td>1136 Block, 32,32</td>
<td>0.013</td>
<td>0.208</td>
<td>0.078</td>
</tr>
<tr>
<td>1140 Block, 32,16</td>
<td>0.021</td>
<td>0.192</td>
<td>0.040</td>
</tr>
<tr>
<td>1148 Block, 32,8</td>
<td>0.039</td>
<td>0.156</td>
<td>0.078</td>
</tr>
<tr>
<td>1164 Block, 32,4</td>
<td>0.065</td>
<td>0.130</td>
<td>0.087</td>
</tr>
<tr>
<td>1204 Block, 32,2</td>
<td>0.047</td>
<td>0.047</td>
<td>0.047</td>
</tr>
<tr>
<td>1256 Block, 16,16</td>
<td>0.013</td>
<td>0.104</td>
<td>0.043</td>
</tr>
<tr>
<td>1280 Block, 16,8</td>
<td>0.023</td>
<td>0.092</td>
<td>0.045</td>
</tr>
<tr>
<td>1312 Block, 16,2</td>
<td>0.072</td>
<td>0.072</td>
<td>0.072</td>
</tr>
<tr>
<td>1604 Block, 8,8</td>
<td>0.015</td>
<td>0.060</td>
<td>0.030</td>
</tr>
<tr>
<td>1520 Block, 8,4</td>
<td>0.030</td>
<td>0.060</td>
<td>0.040</td>
</tr>
<tr>
<td>1552 Block, 8,2</td>
<td>0.055</td>
<td>0.055</td>
<td>0.055</td>
</tr>
<tr>
<td>2016 Block, 4,4</td>
<td>0.022</td>
<td>0.044</td>
<td>0.029</td>
</tr>
<tr>
<td>2048 Block, 4,2</td>
<td>0.045</td>
<td>0.045</td>
<td>0.045</td>
</tr>
<tr>
<td>3072 Block, 2,2</td>
<td>0.037</td>
<td>0.037</td>
<td>0.037</td>
</tr>
</tbody>
</table>
Figure 11. Z8000 2-byte Sub-Block Miss Ratios. Lines connect caches of a block size $b$ and sub-block size 2 bytes ($s_2$) for varying gross (address tags and data) cache sizes.

Figure 12. Z8000 4-byte Sub-Block Miss Ratios. Lines connect caches of a block size $b$ and sub-block size 4 bytes ($s_4$) for varying gross (address tags and data) cache sizes.

Figure 13. Z8000 8-byte Block Miss Ratios. Lines connect caches of a block size 8 bytes ($b_8$) and sub-block size $s_8$ for varying gross (address tags and data) cache sizes.

Figure 14. Z8000 32-byte Block Miss Ratios. Lines connect caches of a block size 32 bytes ($b_{32}$) and sub-block size $s_4$ for varying gross (address tags and data) cache sizes.
Figure 15. Z8000 Block Size Equal Sub-Block Size Miss Ratios. Lines connect caches of a block size $b_x$ and equal sub-block size $s_x$ for varying gross (address tags and data) cache sizes.

Figure 16. Z8000 Results-Part I. Results for caches with net sizes of 32, 128, and 512 bytes. Solid lines connect caches with constant block size $b_x$. Dashed lines connect caches with constant sub-block size $s_x$.

Figure 17. Z8000 Results-Part II. Results for caches with net sizes of 64, 256, and 1024 bytes. Solid lines connect caches with constant block size $b_x$. Dashed lines connect caches with constant sub-block size $s_x$. 
use [24]. For this reason, data just before a reference is less likely to be referenced than data just after it. References tend to enter a block either at its beginning or at a random location. The targets (entry points) of branches should be uniformly distributed over the length of the block; the mean entry location is the middle. Thus, a larger block size will result in more data being loaded from memory locations that are before the point of reference. Load-forward breaks blocks into sub-blocks and, on a miss, only loads sub-blocks from the reference forward. For example, when there are four sub-blocks in a block and a reference is made to sub-block 3, only sub-blocks 3 and 4 will be loaded. Sub-blocks 1 and 2 will remain invalid, because they are unlikely to be needed. Subsequently, if a reference is made to sub-block 2, then either sub-block 2 alone, or 2 and everything forward can be loaded. The former scheme minimizes bus traffic, but requires that the main memory system know what sub-blocks the cache has loaded so far. The latter scheme has redundant bus traffic in the few instances of a backwards reference within a cache block, but allows the main memory system to function autonomously.

Load-forward was studied with traces CPP, C1 and C2 using the latter redundant-load scheme. Since results (Figure 18, 19, and 20 and Table 14) show that few redundant loads were made, there was not enough gain to justify experimenting with the smarter scheme. The points labeled b16-s2-LF in Figures 18 and 19 and b16-s2-LF-g328 in Figure 20.\(^6\) on the 256-byte cache curve in Figure 20 corresponds to the performance of the Z80,000 cache on these particular benchmarks.

The load-forward mechanism reduces bus traffic at a small cost in miss ratio. Changing from a sub-block size equal to the block size to a 2-byte sub-block size with load-forward reduces the traffic ratio by 16.3 percent for a cost of 4.0 percent in the miss ratio. Thus, load-forward is useful if it is easy to implement and the traffic ratio is of some concern. We would expect that load-forward will be especially effective in the instruction space.

Figures 18 and 19 show the miss and traffic ratios on a limited number of runs. Two runs, b16-s2 and b16-s2-LF, appear as single points because the runs were made at only one cache size. The lowest miss ratio occurs on the b16 run because the large block size brings in more data per miss. The load-forward run with block size of 16 bytes is only slightly above the b16 run. The run for block size of 8 is almost on top of the b8 run. Not loading sub-blocks behind the reference does not hurt the miss ratio much.

Figure 19 illustrates the bus traffic for these cache parameters. A block size of 2 bytes (b2) yields the lowest traffic because only the 16 bits that have been requested are brought in on a miss. No data is ever fetched that is not used. The lowest curve in Figure 19 is b8-s2 because even though this option has higher bus traffic owing to a more constrained mapping, it does not

\(^6\)LF stands for load-forward, and g328 for gross cache size of 328 bytes.
<table>
<thead>
<tr>
<th>Net, Gross Cache Size</th>
<th>Block, Sub-Block Size</th>
<th>Miss Ratio</th>
<th>Traffic Ratio</th>
<th>Traffic Ratio (paged)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>94 8,8</td>
<td>0.257</td>
<td>1.028</td>
<td>0.514</td>
</tr>
<tr>
<td></td>
<td>97 8,2,LF</td>
<td>0.263</td>
<td>0.865</td>
<td></td>
</tr>
<tr>
<td></td>
<td>97 8,2</td>
<td>0.678</td>
<td>0.678</td>
<td>0.678</td>
</tr>
<tr>
<td></td>
<td>192 2,2</td>
<td>0.612</td>
<td>0.612</td>
<td>0.612</td>
</tr>
<tr>
<td>256</td>
<td>314 16,16</td>
<td>0.120</td>
<td>0.960</td>
<td>0.400</td>
</tr>
<tr>
<td></td>
<td>328 16,2,LF</td>
<td>0.128</td>
<td>0.772</td>
<td></td>
</tr>
<tr>
<td></td>
<td>328 16,2</td>
<td>0.489</td>
<td>0.489</td>
<td>0.489</td>
</tr>
<tr>
<td></td>
<td>376 8,8</td>
<td>0.164</td>
<td>0.656</td>
<td>0.328</td>
</tr>
<tr>
<td></td>
<td>388 8,2,LF</td>
<td>0.169</td>
<td>0.567</td>
<td></td>
</tr>
<tr>
<td></td>
<td>388 8,2</td>
<td>0.454</td>
<td>0.454</td>
<td>0.454</td>
</tr>
<tr>
<td></td>
<td>768 2,2</td>
<td>0.402</td>
<td>0.402</td>
<td>0.402</td>
</tr>
</tbody>
</table>

*LF stands for Load-Forward.*
Figure 18. Z8000 Load-Forward Miss Ratios. Lines connect caches of a block size $b_z$ and sub-block size $s_y$ for varying gross (address tags and data) cache sizes. $LF$ stands for load-forward.

Figure 19. Z8000 Load-Forward Traffic Ratios. Lines connect caches of a block size $b_z$ and sub-block size $s_y$ for varying gross (address tags and data) cache sizes. $LF$ stands for load-forward.

Figure 20. Z8000 Load-Forward Results. Results for caches with net sizes of 64 and 256 bytes. $b_z$ is block size, $s_x$ is sub-block size, LF stands for load-forward, and $g_z$ is the gross cache size.
have the tremendous tag overhead of the b2 scheme. Almost two-thirds of the gross area of the b2 cache is devoted to address tags.

Figure 20 shows traffic ratio versus miss ratio for cache sizes of 64 and 256 bytes. If the overriding constraint is the miss ratio, then the best cache can be found by moving a horizontal line (slope = 0) up from the x-axis until it hits b8. For minimum bus traffic, a vertical line (slope = −∞) should be moved right until b2 is intercepted. If both miss ratio and bus traffic are critical then a line with slope between 0 and −∞ can be used to weigh their relative importance. b2 is a larger cache than the other three because it has a tag for every 2 bytes instead of every 8 bytes.

Load-forward caches will perform better with paged-mode RAMs. Paged-mode RAMs will reduce the cost of transfers by an amount related to the distribution of number of sub-blocks brought in per miss. Traffic costs decrease when the number of sub-blocks loaded per miss is increased. The number of sub-blocks loaded must be less than the number of sub-blocks per block s. Using the work in section 4.3, traffic costs will be reduced by a factor g(s) which bounded from below by:

\[ g(s) \geq \frac{1}{s} \left( 1 + \frac{1}{3} (s - 1) \right) = \frac{(s + 2)}{3s}. \]

An upper bound on g(s) can be established by assuming that the first reference to all blocks is at a random location so that the number of sub-blocks loaded on a miss is uniformly distributed between 1 and s. Since, some streams enter block sequentially, the distribution function of the number of sub-blocks loaded on a miss will never be greater than the uniform distribution function. Thus, the number of sub-blocks loaded on a miss is bounded from below, and g(s) will be bounded from above by:

\[ g(s) \leq \left( \sum_{w=1}^{s} w \right)^{-1} \sum_{w=1}^{s} \left( 1 + \frac{1}{3} (w - 1) \right) = \frac{(s + 5)}{3(s + 1)}. \]

Thus, using paged-mode RAMs in a 616-s2-LF cache should reduce traffic costs to between 42 and 48 percent (s=8) of the costs of using standard RAMs.

The load-forward mechanism reduces bus traffic at a small cost in miss ratio. It is useful if it is easy to implement and bus traffic is of some concern. We would expect that load-forward will be especially effective in the instruction space.

4.6. Other Results

IBM/370 traces showed worse locality than Z8000 and PDP-11 traces. On the 370, Cobol benchmarks performed worse than all of the other runs, among which performance was comparable.
Figures 21 and 22 show miss ratios for two typical IBM/370 Fortran runs with block size equal to the sub-block size. Figures 23 and 24 exemplify IBM/370 Cobol. These can be compared to PDP-11 results in Figure 4 and Z8000 results in Figure 15. We show two runs each for Fortran and Cobol instead of the means to show the reader that individual trace results do not look radically different from mean results.

Figures 25, 26, 27, and 28 show traffic ratios for those IBM/370 runs.

Not surprisingly, the PDP-11 and Z8000 microprocessor traces benefit from a smaller cache than the IBM Fortran traces. Cobol traces have poor miss ratios, but reasonable traffic ratios. Nevertheless, the bus traffic per instruction is about three times that of the PDP-11 runs.

Small cache performance was generally good, but decayed slightly from Z8000 to PDP-11 to IBM Fortran runs. Cobol runs have poor miss ratios, but reasonable traffic ratios. Thus, we believe that some area on microprocessor chips is best used in an on-chip cache rather than providing additional special-purpose logic. A very small 64-byte cache with a block size of 4 bytes and a sub-block size of 2 bytes reduced memory accesses and bus traffic by one-third in both PDP-11 and Z8000 runs. 32-bit IBM/370 runs need more than the 64-byte cache size that work for the 16-bit runs. In fact, larger 1024-byte caches perform better for all traces and can be designed to run with miss ratios as low as 0.033 for large 32-byte blocks or with traffic ratios as low as 0.072 for small 2-byte blocks. Additional improvements can be achieved with intelligent control. Paged and nibble mode memories can reduce traffic costs by one-third with a sub-block size of 2 words and by one-half with a sub-block size of 4 words. Load-forward, which loads data from the reference forward, can reduce misses for a small increase in the traffic ratio.
Figure 21. Fortran (FGO1) Block Size Equal Sub-Block Size Miss Ratios. Lines connect caches of a block size bZ and equal sub-block size sZ for varying gross (address tags and data) cache sizes.

Figure 22. Fortran (FGO3) Block Size Equal Sub-Block Size Miss Ratios. Lines connect caches of a block size bZ and equal sub-block size sZ for varying gross (address tags and data) cache sizes.

Figure 23. Cobol (CGO1) Block Size Equal Sub-Block Size Miss Ratios. Lines connect caches of a block size bZ and equal sub-block size sZ for varying gross (address tags and data) cache sizes.

Figure 24. Cobol (CGO3) Block Size Equal Sub-Block Size Miss Ratios. Lines connect caches of a block size bZ and equal sub-block size sZ for varying gross (address tags and data) cache sizes.
Figure 25. Fortran (FGO1) Block Size Equal Sub-Block Size Traffic Ratios. Lines connect caches of a block size $b_x$ and equal sub-block size $s_x$ for varying gross (address tags and data) cache sizes.

Figure 26. Fortran (FGO3) Block Size Equal Sub-Block Size Traffic Ratios. Lines connect caches of a block size $b_x$ and equal sub-block size $s_x$ for varying gross (address tags and data) cache sizes.

Figure 27. Cobol (CGO1) Block Size Equal Sub-Block Size Traffic Ratios. Lines connect caches of a block size $b_x$ and equal sub-block size $s_x$ for varying gross (address tags and data) cache sizes.

Figure 28. Cobol (CGO3) Block Size Equal Sub-Block Size Traffic Ratios. Lines connect caches of a block size $b_x$ and equal sub-block size $s_x$ for varying gross (address tags and data) cache sizes.
5. RISC II Instruction Cache

The RISC II Instruction Cache is a single-chip instruction cache implementation used to study architectural ideas for on-chip memory caches. The cache includes four architecture innovations that are of general interest. The cache performs with an average miss ratio of 0.15 on limited benchmarks.

The RISC II Instruction Cache is a single nMOS chip designed specifically to work with the RISC II implementation of the RISC Architecture. The cache is designed to return data to RISC II 200 nanoseconds after the instruction is presented in order for the system to operate at a 400 nanosecond cycle time. It holds 0.5 Kbytes arranged in 64 direct-mapped blocks of 8 bytes each. Addresses and instruction data are passed to and from the the cache chip across a single 32-bit system bus. The architecture does not allow instructions to cross page or block boundaries. This significantly reduces the number of cache control states. The cache has no memory-update problems because the RISC Architecture does not allow writes into the instruction stream. We felt the complexity of a general cache was not needed since most of the data needs are served by the large register file. The cache is described in detail in two papers [7, 25].

Global control in the cache chip is simple. The RISC II instruction cycle is divided into four clock phases. Cache control was implemented with a finite state machine with six states that changes state every machine cycle. We call this state the superstate. Control signals change within a superstate because they are logically ANDed with processor clock phases and other inputs to produce the cache’s state. The cache goes through four superstates to handle a miss during which the RISC II processor waits.

The cache was designed with the Mead-Conway [26] design style in nMOS with one layer of metal to have 45 K transistors and 63 pins. It was fabricated in nMOS with 4.0-micron channels (lambda of 2.0 micron). Chips from the first fabrication run work correctly with a 250 nanosecond access time, consuming 1.3 watts, and of size 7.0 millimeters by 7.6 millimeters.

The three sub-sections to follow present some background on RISC, outline the ideas in the chip, and evaluate their performance.

5.1. RISC Background

Researchers at U.C. Berkeley have proposed a Reduced Instruction Set Computer Architecture called RISC. Reduced Instruction Set Computers use simple straight-forward instructions as primitives to build high-level semantics. The RISC architecture is presented in several papers by Patterson et al [8, 9, 27]. The RISC Architecture is a Load-Store architecture. All other instructions operate on data that is kept in registers. Instructions typically have two source registers
and a destination. Thirty-two general registers are accessible at one time. High-Level Language computers must support procedure calls including saving and restoring registers. This is typically done by writing and reading the registers across the von Neumann bottleneck to a stack in main memory. RISC saves state by allocating a new bank of registers. Thus, data must only be saved in memory when all banks of an implementation are full. Register banks are overlapped so that most parameter passing can be done without putting things in main memory.

RISC II is the second implementation of the RISC architecture. The RISC II microarchitecture and datapath are described in two papers [28, 29]. RISC II is a VLSI nMOS implementation that realizes control without the indirection of microcode. It uses a three-stage pipeline to execute a register-to-register instruction every 400 nanosecond (micro) cycle. Loads and Stores require two cycles. RISC II has 8 overlapping banks of registers.

5.2. Four Architectural Ideas

The RISC II Instruction Cache's architecture includes four innovations: multi-chip expansibility for flexibility, remote program counter for decreased access time, support for dynamic code expansion for increased effective size, and fault tolerance for increased yield.

It is often prudent to retain flexibility in designs until more information can be gathered to properly make a decision. For example, silicon design time is a poor time to specify cache size because the optimal size depends on the workload envisioned for a particular system. Our cache chip allows a system designer to build an instruction cache from any number of cache chips. Therefore, the instruction cache can be any multiple of 0.5 Kbytes. This Multi-Chip Expansibility is a by-product of not fitting the instruction cache on the processor chip. The multiprocessor cache is built by hanging all the cache chips off the system bus and using some mechanism for selecting which chip will handle a given request. This chip selection can be as simple as a decoder operating on select address bits much like what is done in memory systems. This will produce a cache that is still direct-mapped. A simple, unimplemented extension will allow associativity between chips. For example, four chips could be used to produce a four-way set associative cache. In this scheme, all chips would try to handle instruction requests in parallel with at most one hitting. One chip would be designated to receive data on a miss. Random replacement could be simply implemented by passing a single token between chips. At first glance this may look like FIFO replacement between chips. But for any given set, the replacement is pseudo-random.

Memory cells in a cache can only be made so fast. The Remote Program Counter is an architectural innovation that decreases the effective access time to these cells. The Remote Program Counter attempts to guess the next instruction address so that the cache can begin its data fetch before the processor presents the instruction fetch address. Of course a speed-up is realized only if the guess is correct. Guesses are made by special-purpose logic on the cache chip which
has limited instruction-decode ability as follows: The next instruction address after a sequential instruction is assumed to be the current program counter plus the instruction's length. Targets after unconditional relative branches and relative calls are produced by adding the program counter to the immediate displacement. Conditional relative branches are treated as either sequential instructions or unconditional relative branches, depending on whether the compiler has set the static jump-likely bit. Thus, miss-predictions most commonly occur on conditional relative branches that go the other way and on procedure returns. Miss predictions will occasionally occur because of absolute jumps, indexed jumps, absolute procedure calls (all of which are rarely generated by RISC C compiler), and context switches caused by traps and interrupts. When the cache mispredicts, it will try again to fetch the data using the actual address. On this second fetch, data can still be provided to the processor in less than half the time of a cache miss.

Once the number of memory cells in a cache is fixed, there are still ways to increase the effective size of a cache. The effective size of an instruction cache can be measured by the number of instructions that are typically resident in the cache. This size can be increased with Code-Compaction. Standard RISC instructions are all 32-bit for easy decodability. Our support for compact code allows code to be stored in a more dense format that can be re-expanded "on-the-fly" before it is given to the processor. Studies showed that Huffman encoding of the standard RISC instructions would yield code 43 percent smaller [30]. Selected half-word instructions showed a 30 percent savings. We decided to implement support for half-word instructions because this accomplished 70 percent of the density increase with much less added complexity than would be required to support bit-variable Huffman-encoded instructions. A little of this compaction is lost because we do not allow full-word instructions to cross cache block boundaries. For this reason, the last of an odd number of half-word instructions to be placed into a block is instead incarnated in its full-word format. Block-crossing instructions significantly increase cache control complexity because, among other things, a single access can cause two misses. The first implementation of the cache chip did not include the logic to expand 16-bit instructions back into 32-bit ones. It was left out so that we could complete the chip design in a single quarter.

Fabrication errors destroy integrated circuits and drive up the cost of the chips that do work. At a RISC Instruction Cache design review, Gaetano Borriello and Alan Paeth of Xerox PARC suggested Fault Tolerant Bits as a way to salvage chips that would otherwise be thrown out. Architecturally every block of a cache chip must have a valid bit so that blocks can be tagged empty after power-up and other events that may flush the cache. The Fault Tolerant Bit is an additional valid bit that can be set to zero to permanently invalidate a cache block in the event of some fabrication errors in the memory array. When a block is disabled, all references to it miss. Instruction data is given to the processor before it is loaded into the cache's internal memory array. Thus, the processor will always get correct data even though it may have to wait
a little longer. Fault Tolerant Bits must be serially loaded when the system is powered up, or they can be disabled if all blocks are good.

The cache's memory array is 50 percent of the area of the cache chip. We can recover from any fabrication error in it that does not short power to ground or erroneously drive bit lines. We estimate that if we assume we can recover from two-thirds of cache memory array errors, then we will see relative yield improvements of 30, 67, and 135 percent for defects per square inch of 2, 4, and 8, respectively. This estimate is conservative because although the memory array is 50 percent of the chip area, it is 85 percent of the active transistors. All of this is worthwhile since whatever the numbers, the Fault Tolerant Bits add only 0.6 percent chip area.

Invalidating cache blocks increases the cache's miss ratio. All valid blocks miss at the same rate as the intact cache and all invalid blocks miss on every reference. If we assume that references are uniformly distributed over the 64 direct-mapped cache blocks, which is a much weaker restriction than requiring references to be uniform over the entire address space, then the following calculates the miss ratio, no matter which $n$ blocks are invalidated. If the references are not uniform, the following calculation is the average miss ratio when a random $n$ blocks are invalidated. Non-uniform reference distributions would imply that some groups of $n$ errors would produce better caches than other groups. Assuming the miss ratio for a intact cache is $m(0)$, the average miss ratio for a cache with $n$ bad blocks $m(n)$ is defined by:

$$m(n) = \frac{m(0)}{64} * (64 - n) + \frac{n}{64},$$

where the first term is the miss ratio of the intact cache blocks and the second is the bad blocks always missing. The performance loss is:

$$m(n) - m(0) = \frac{n}{64} * (1 - m(0)).$$

Since $0 \leq m(0) \leq 1$, the average performance loss is bounded by $\frac{n}{64}$. For example, a typical miss ratio of $m(0)=0.15$ (hit ratio of 0.85) will yield a loss of 1.3 percent for each block invalidated. No simulations were run to test that the references are indeed uniform on the cache blocks because this assumption is not critical to the usefulness of the Fault Tolerant Bits.

5.3. RISC II Simulation Results

Some simulations were run to test cache performance with respect to multi-chip expansibility, remote program counter, and code compaction. Cache performance was tested using the largest program we can run on the RISC Simulator: the Portable C Compiler $[31]$ for RISC. Results show that the cache missed 0.148 of the time. Doubling the number of cache chips from one to two changes the miss ratio from 0.148 to 0.125 which is probably not worth the cost of an extra
VLSI chip. The remote program counter works well, correctly predicting the next instruction address 88.9 percent of the time. Code compaction helps performance by increasing the average number of instructions brought in on a miss from 2.0 to 2.3. This decreases misses to 0.108.

The RISC II Instruction Cache was studied using an instruction-level simulator of RISC II to pipe information to a modified, early version of Dinero. Programs to be run on the RISC simulator had to survive the RISC C compiler (PCC) and were easier to run if they required no input. The limited results presented here are based on four small benchmarks and the RISC C compiler compiling these four benchmarks. The benchmarks are Ackermann's function with argument (2,6), Towers of Hanoi for 9 levels, Erathoshene's Sieve [32] (for prime numbers), and BenchE (a string jump search program). The benchmarks themselves are too small to adequately test more than the jump prediction mechanism.

The multi-chip expansibility feature was tested in two ways: completely direct-mapped and set-associative between chips. Figure 29 shows the miss ratios for direct-mapped caches of 1, 2, 4, and 8 chips. Doubling the number of chips in the cache produces a cache with a 19 percent miss ratio improvement. Figure 30 also shows caches that have random replacement for sets spanning chips so that a n-chip cache in n-way set-associative. Doubling the number of chips in the cache produces a cache with a 25 percent miss ratio improvement. Set-associative caches have lower miss ratios than direct-mapped caches by the following differences in miss ratio: two chips 0.0060, four chips 0.0035, and eight chips 0.0158.

These results imply two things. First, multiple cache chips give only a small performance gain per chip. For example, eight direct-mapped chips have half the miss ratio of a single chip for eight times the cost. Multiple chips are justified only if the cost of the cache chips are small with respect to system cost and the cache's miss ratio is critical to system performance. Second, associativity across chips increases the performance only slightly and is therefore only worthwhile if it is easy to implement.

The Remote Program Counter performs well. In tests it correctly predicted 89.9 percent of next-instruction addresses. The prediction rate ranged from 79.2 to 93.7 percent (See Figure 31). Let $t_{eff}$ be the effective access time, $m$ be the miss ratio, $p$ the prediction rate, and $t_{hp}$, $t_{hm}$, and $t_{miss}$ be the time for hits predicted correctly, hits mispredicted, and misses. Without prediction:

$$t_{eff} = (t_{hm}) * (1 - m) + (t_{miss}) * (m)$$

With prediction:

$$t_{eff} = (t_{hp}) * (1 - m) * (p) + (t_{hm}) * (1 - m) * (1 - p) + (t_{miss}) * (m)$$

The cache is designed to return data in 200, 600, and 1400 ns for hits predicted correctly, hits mispredicted, and misses. Prediction improves effective access time from 719 to 413 ns.
719 = 600*(1 - 0.1485) + 1400*(0.1485)

413 = 200*(1 - 0.1485)*(0.899) + 600*(1 - 0.1485)*(1 - 0.899) + 1400*(0.1485)

The Remote Program Counter works as an internal prefetch mechanism. We believe the idea is good for instruction caches, instruction fetch units, and instruction memories.

The RISC II Instruction Cache also supports code compaction. Compact RISC code is 20 percent smaller after corrections have been made to avoid block-crossers. Compact instructions were modeled by increasing the cache simulator's block size from 8 to 10 bytes. Compact instructions yield a miss ratio lower than that uncompacted code. For varying amounts of chips, the miss ratios, with relative improvement in parentheses, are: one chip 0.0401 (27.0 %), two chips 0.0326 (26.1 %), four chips 0.0425 (43.3 %), and eight chips 0.0372 (47.7 %). The miss ratios are plotted in Figure 32. The curve with caches using compacted code are labeled CPACT.

People often claim that code density is important. This evidence at the cache level of the memory hierarchy supports this idea. One observation is that dense code in our standard 0.5 Kbyte-size cache performs better than a cache holding the same number of instructions with more blocks of uncompacted code. This is because a miss of compacted code brings into the cache more instructions than an uncompacted code miss (2.3 versus 2.0; 2.3 is 8 bytes divided by the product of 85 % and 4 bytes per instruction). To get invariant results on compacted and uncompacted code, one must change the block size, not the number of blocks.
Figure 29. RISC Direct-Mapped Miss Ratios. Lines connect runs of the Portable C Compiler (PCC) for RISC compiling selected benchmarks for varying number of instruction cache chips (0.5 Kbyte each) connected in a direct-mapped (DM) organization.

Figure 30. RISC Random Replacement Miss Ratios. Lines connect runs of PCC compiling selected benchmarks for varying number of cache chips (0.5 Kbyte each) connected in a direct-mapped (DM) or random-replacement (RAND) organization.

Figure 31. RISC Jump Prediction Accuracy. Histogram that shows the percent of next-instruction-addresses correctly predicted by the remote program counter.

Figure 32. RISC Compact Code Miss Ratios. Lines connect runs of PCC compiling selected benchmarks for varying number of direct-mapped cache chips (0.5 Kbyte each). Lines labeled CPACT are results with compacted RISC instructions.
6. Conclusions

We believe that some of the limited silicon area on microprocessor chips is best used as the top of the memory hierarchy rather than providing additional special-purpose logic.

Small on-chip caches are effective in reducing latency and off-chip traffic. Larger caches do a better job of both, but it is surprising how effective minimum caches can be. A 64-byte cache with a block size of 4 bytes and a sub-block size of 2 bytes reduced memory accesses and bus traffic by one-third in both PDP-11 and Z8000 runs. This cache requires only 128 bytes of RAM cells to hold address tags and data for a 16-bit machine.

Slightly larger caches perform better and can be designed to run at different operating points depending on system considerations. For example, a 1024-byte cache with fixed 32-byte blocks and sub-blocks that can be varied from 2 to 32 bytes allows miss and traffic ratios to be traded off. 32-byte sub-blocks yield miss and traffic ratios of 0.033 and 0.542; 2-byte sub-blocks miss six times as often but initiate one-third the bus traffic. Additional improvements can be achieved with intelligent control.

Miss ratio and bus traffic can be traded off against each other. Increasing the sub-block size increases the amount of data that is brought in per miss and improves the miss ratio as long as most of the data brought in is used. When the amount of data brought in on a miss is increased, the likelihood that some of it will never be used is also increased. Therefore increases in the bus traffic occur.

Paged and nibble mode memories allow reads of sequential words to proceed more quickly than reads to random words. Cache memories whose sub-block size is greater than a machine's word size can use these memories to load sub-blocks. Traffic cost can be reduced by one-third with a sub-block size of 2 words and by one-half with a sub-block size of 4 words.

Load-forward overcomes a traditional disadvantage of large cache blocks, namely, the loading of data into the cache that is not likely to be used because it lies before the first reference to a block. Load-forward gives away a little performance in its miss ratio when compared to a cache with large blocks, but reduces bus traffic significantly. It is recommended in those cases in which bus traffic is of some concern and the technology allows it implemented cheaply. VLSI implementations will only be inexpensive if the memory interface, not usually VLSI, can be modified without arduous compatibility constraints.

Small cache performance was generally good, but decayed slightly from Z8000 to PDP-11 to IBM Fortran runs. 32-bit runs performed poor relative to 16-bit runs for extremely small cache sizes (e.g. 64 bytes). Cobol runs have poor miss ratios, but reasonable traffic ratios.
The RISC II Instruction Cache illustrates three good ideas for on-chip caches. (Multi-chip expansibility is inherently an off-chip idea.) The remote program counter used knowledge of RISC II's instruction set to reduce effective access time by 42 percent. By storing RISC II instructions compacted in memory but expanding them before they go to the processor, we decreased the miss ratio of a single cache chip by 27 percent with no performance penalty to the processor. Fault tolerance helped reduce the cost of working chips by increasing yield for only 0.6 percent extra area.
7. Acknowledgements

This paper is based upon work supported by the Nation Science Foundation under Grant MCS82-02591.

I would like to thank my advisor Alan Jay Smith for support and guidance, David A. Patterson for his suggestions as my second reader, and Susan Eggers and Susan Dentinger for their useful comments on drafts of this paper. Also, thanks to John Lee for PDP-11 traces, John Lee and the Amdahl Corporation for IBM System/370 traces, and Juan Porcar and the Zilog Corporation for Z8000 traces. The RISC II Instruction Cache was designed and implemented with Dimitris Lioupis, Chris Nyberg, Tim Sippel, and David A. Patterson.
References


