Benchmark Characterization

Thomas M. Conte Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
University of Illinois
conte@uiuc.edu

Abstract

The design of experimental systems ususally employ some form of simulation. The inputs to the simulation are typically standard benchmarks. This paper presents a method for finding a cost-effective design for each benchmark. This method is called *benchmark characterization.* Benchmark characterization is possible in-part due to the advent of high-performance architecture-independent compiler technology. To demonstrate the method, seven benchmarks are characterized. Five of the benchmarks are from the SPEC benchmark set (*gcc*, *espresso*, *spice*, *li* and *matrix300*) and two are popular synthetic benchmarks (*dhrystone* and *whetstone*). Benchmark characteristics are reported for the processor, memory system, and operating system.

1 Introduction

The design of an experimental system is a complex matter with a bewildering number of design choices. Consider the design of the instruction set in which adding a new instruction might improve overall performance by providing a better interface to the hardware, or degrade performance by reducing the clock speed. The choice of whether or not to include a new instruction depends on the system's speed with or without the instruction. The usefulness of the new instruction depends on the programs the system will run. In practice, designers often use benchmark programs and assume they represent the users' programs. Hence, the system's performance while running the benchmarks determines the design of the system. However, there is a problem of putting the cart before the horse: how can a system be designed to run a set of benchmarks efficiently if the benchmarks' performance on the machine isn't known until the design is completed?

The performance of a system can be simulated before it is ever built and the simulation results can be used to improve the design. Simulation involves selecting an initial design, simulating the design for each benchmark– often a lengthly process– and then adjusting the design and reiterating. If the initial design is far from what is required, the number of iterations will be large. Now imagine a reference work that lists several cost-effective system designs for each benchmark. Such a book could be used to design the initial system. Simulations of this prototype design could then be used for "fine tuning," cutting the number of simulation iterations significantly.

One method of finding cost-effective system designs for benchmarks would be to measure the benchmarks' performance on many systems. Of course, such a brute-force method would be highly time consuming and inconclusive. A more attractive method would be to define an abstract system that is general enough to include many system designs as special cases and then measure the benchmarks' performance in terms of this abstract system. We call this "abstract performance" the benchmark's *characteristics* and the process itself, *benchmark characterization* [1].

Until recently, benchmark characterization was impossible because defining a useful abstract system architecture was difficult. For example, the compiler technology often was an expensive, integral portion of a system (especially for RISC-based systems), where the better the compiler, the better the system's performance. Hence, running benchmarks often required combining the compiler's performance with the hardware's performance. Recently, however, high-quality architecture-independent compilers have emerged (GNU C [2] and IMPACT C [3]). These compilers contain an *intermediate code* that is essentially the machine language to an abstract architecture. This abstract machine language is widely machine-independent. For example, RTL, the GNU C intermediate language, is used for a wide spectrum of processors, including the Digital VAX, the MIPS

0073-1129/91/0000/0364$01.00 © 1991 IEEE

R2000, the Motorola 68000, among others [2].

With the pieces falling into place, the time has come to consider methods for benchmark characterization. We present such a method in this paper and then show the benchmark characteristics for a set of commonly-used benchmarks. The benchmark set we use includes some benchmarks from the SPEC benchmark set[1] (*gcc*, *espresso*, *spice*, *li*, and *matrix300*) and the two most popular synthetic benchmarks (*dhrystone* (version 1.1) and *whetstone*) [4][5].

2 The Method of Benchmark Characterization

Characterization of a benchmark requires measurement of its run-time behavior. The behavior we measure is in terms of an abstract system whose instruction set is the compiler's intermediate code. These "instructions" are interpreted by simulation and the results are used to produce the benchmark's characteristics. This process is illustrated in Figure 1. In the design of this process, we have built upon some clever ideas from the Computer Architecture Workbench by Mitchell and Flynn [6], which was a simulation engine for architecture comparisons.

The compiler that we use for our characterization process is GNU C version 1.37.1, which is public-domain software written by the members of the Free Software Foundation [2]. GNU C is an optimizing C language compiler that implements many traditional compiler optimizations (e.g., common subexpression elimination, jump and loop optimizations, peephole optimization, etc.). To instrument the intermediate code of GNU C, we used Larus' AE trace collection tool [7], modified to produce a trace of intermediate code. Step 1 in Figure 1 depicts the modified compiler. After the intermediate code representation is transformed by optimizations, it is normally converted to assembly code by the code generation phase of the compiler. We inserted a new phase ("instrument") that adds intermediate code to write dynamic behavior to a I/O channel when the compiled benchmark is run. (The I/O channel in our implementation is a Unix *socket*.) The dynamic behavior information that flows down the I/O channel while the benchmark runs is composed of intermediate code instructions, data memory references, and system call events. Instruction set design has a high impact on instruction memory referencing behavior, we therefore chose to exclude it from our measurements and consider only data memory behavior. Similarly, the number of registers available has a high impact on data memory behavior. The abstract system has an infinite number of registers. Methods exist to reconstruct the additional overhead of a finite register file [8]. The remaining memory references are caused by the benchmark and not an artifact of the system's design [9].

Our compilation method needed to be augmented to solve some minor problems. For example, some of the benchmarks were written in FORTRAN (e.g., *spice* and *matrix300*). We did not want to use a separate compiler back end for these benchmarks since it might produce inexplainable effects depending on how aggressive an optimizer the FORTRAN compiler was. Although C and FORTRAN are close enough to use the same compiler back end, no FORTRAN front end (essentially, a parser) existed for GNU C. To overcome this problem, we used a FORTRAN to C translator from AT&T Bell Laboratories, *F2C* [10]. *F2C* translates FORTRAN statements into C essentially at a statement-by-statement level. Therefore, it is equivalent to a true FORTRAN front end to GNU C. Another problem was that a significant fraction of our C language benchmarks' execution was spent in standard library functions. We solved this problem by compiling special instrumented versions of the library functions. Since *F2C* includes implementations of the FORTRAN libraries, we instrumented and included these in the tracing process also.

Abstract system behavior was measured by several tools (Step 2 of Figure 1). To characterize the memory referencing behavior, we used our recurrence-conflict method (RCM) [11]. RCM uses a modified single-pass LRU stack-based algorithm to record the number of recurrences and organizational misses for all cache dimensions in a design space. Our design space for data references included all caches up to 2GB, with block size ranging from 16B to 4KB, and associativity levels of one-way (direct-mapped), two-way, four-way, and fully associative (for a discussion of cache organization see Smith [12]). System calls and intermediate instruction frequencies were recorded dynamically.

3 A Benchmark Characterization

We have outlined a method for benchmark characterization. Below we present benchmark characteristics of the six benchmarks: *gcc*, *espresso*, *spice*, *li*, *matrix300*, *dhrystone* and *whetstone*.

3.1 Processor benchmark characteristics

Processor design involves providing execution resources (registers, function units, and supporting logic) to achieve high performance. The relative im-

[1] The University of Illinois is a member of SPEC.

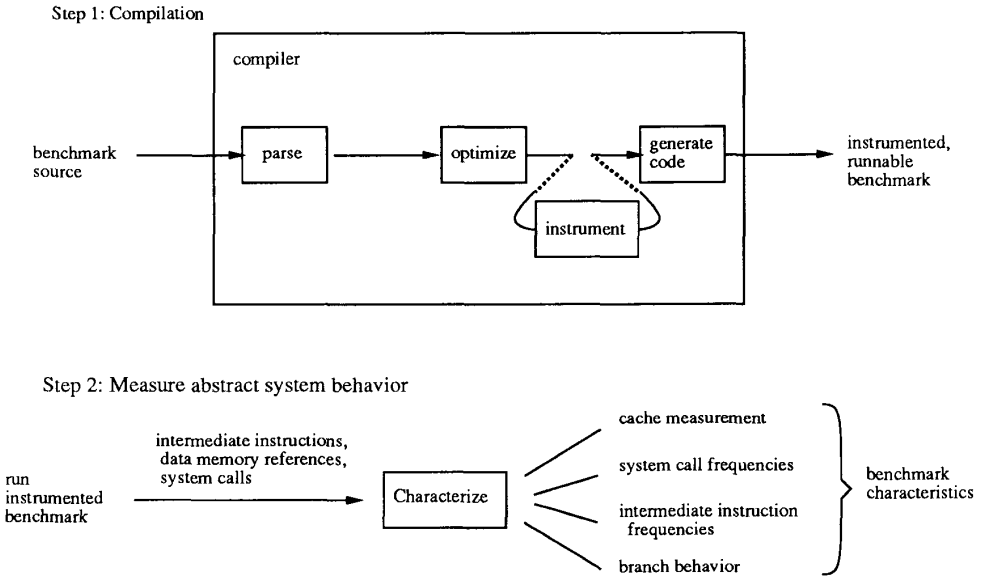

Figure 1: The benchmark characterization process.

Table 2: Intermediate code instruction frequency.

Benchmark	itAlu	itMul	itDiv	bShft	load	store	fpAdd	fpMul	fpDiv	fpCvt	brnch
dhrystone	42%	*	*	2%	20%	15%	0%	0%	0%	0%	21%
whetstone	40%	*	0%	9%	18%	6%	5%	4%	*	*	17%
gcc	44%	*	*	6%	14%	11%	0%	*	*	*	24%
espresso	38%	*	*	12%	27%	2%	0%	0%	*	*	21%
spice	25%	*	*	11%	40%	9%	2%	1%	*	*	12%
li	23%	*	*	1%	31%	12%	0%	0%	0%	0%	33%
matrix300	28%	11%	*	11%	22%	11%	5%	5%	*	0%	6%

(* value is $< 1\%$)

portance of various operations can be extremely useful in designing the processor. Consider the example of whether to add an instruction that was given in the introduction: the relative use of different instruction classes in benchmarks could answer this question. Such relative frequencies could also answer how important floating-point hardware is, or whether a barrel shifter is worthwhile to implement. To gather the intermediate code instruction frequencies, we chose eleven categories of important instruction types (see Table 1) and separated GCC's intermediate instructions into these categories. The dynamic frequencies of each instruction type is presented in Table 2. The value of high-performance integer hardware (itAlu) cannot be argued, as it is its use represents at least a quarter of the instructions for all benchmarks. However, integer division (itDiv), floating-point division (fpDiv), and floating-point conversions (fpCvt) are hardly used and could safely be implemented by software. Shifting (bShft) is used a considerable amount by *espresso*, which performs bit manipulations. Floating-point intensive programs such as *whetstone*, *spice*, and *matrix300* use shifting for array index calculation. Hence, a barrel shifter will find use in both integer-intensive and floating-point-intensive environments. Integer multiplication (itMul) is also used by *matrix300* for array index calculation and suggests hardware support for integer multiplication, although perhaps not a full-scale integer multiplier. The frequency of control transfers (brnch) agrees with the rather widely-held belief that floating-point programs have longer basic blocks. Therefore, if the system will be using applications close to *whetstone*, *spice*, and *matrix300*,

Table 1: The intermediate code instruction types.

Type	Description
itAlu	Integer ALU operations
itMul	Integer multiply
itDiv	Integer divide, mod
bShft	Barrel shifting
load	Load
store	Store
fpAdd	Floating point addition
fpMul	Floating point multiplication
fpDiv	Floating point division
fpCvt	Floating point conversion
brnch	Control transfers

branch prediction hardware is optional.

3.2 Memory system benchmark characteristics

Modern memory systems consist of one or more levels of caching above a virtual memory system. The entire set of miss ratios for the benchmarks constitutes a vast amount of data. Instead of presenting that data, we present several cost-effective memory system designs for the benchmarks. We describe these designs first by discussing the top-level cache and second-level cache requirements, then several good TLB designs, and we close by discussing main-memory designs. Throughout we exclude the results for 4-way set associative caches because we discovered that the required cache sizes where the same.

3.2.1 Top-level and second-level cache requirements

Figure 2 shows the minimum dimensions required for a top-level cache which is backed up by a second-level cache. The design criterion we used was the smallest (i.e., least expensive) cache that provides a 10% miss ratio. The required cache size ranges from 128B (*matrix300*, fully-associative) to 32KB (*spice*, all associativities) An 8KB direct-mapped cache or a 4KB 2-way set associative cache achieves the design goal for all benchmarks except for *spice*. These two dimensions result in 13.0% and 12.7% miss ratio respectively for *spice*.

Most benchmarks show a decrease in cache size requirement as the set associativity increases. Misses due to cache dimensions account for a significant portion of the misses in this performance range. For integer benchmarks, these misses are mostly due to accesses to the stack and heap locations whose addresses tend to be far way from each other. For numerical benchmarks, they are due to the access to large arrays where several frequently accessed elements may contend for the same cache set. The only exceptions are *spice* and *whetstone*. The misses for these two benchmarks are mostly due to insufficient cache size to hold the working set. Large block size (64B) helps to reload the spilled working set with fewer misses for *spice*. The reduced number of reloading misses causes misses due to cache dimensions to dominate the miss ratio, thus making the benefit of increased set associativity more obvious.

We now turn our attention to the design of second-level caches. The design criterion we used depended in-part on a benchmark's *intrinsic miss ratio* for a given block size. The intrinsic miss ratio corresponds to the miss ratio for a cache with an infinite number of blocks. Our criterion was to design the second-level caches with either a very small overall miss ratio of 1% or the intrinsic miss ratio if 1% was not achievable. Figure 3 shows the cache dimensions required for such a second level cache. The cache sizes required for all benchmarks are 1MB, 256KB, and 128KB for direct mapped, 2-way, and fully associative caches, respectively. It is once again clear that much smaller caches can be designed to cover all benchmarks except *spice*. Some compromised sizes would be 256KB, 128KB, and 64KB for the set associativities. They offer 1.59%, 1.84%, and 2.47% miss ratios (respectively) for *spice*.

3.2.2 Translation lookaside buffer requirements

Translation lookaside buffer (*TLB*) is a memory designed to cache the virtual memory translation results for the frequently accessed pages in the virtual space. Without a TLB, it takes a significant number of cycles to translate a virtual address into a physical address (typically 10-40 cycles). On the other hand, the cost of translation can be eliminated if the translation results are found in the TLB. A small increase in the miss ratio of the TLB usually results in a significant loss in performance. Therefore, it is important to design the TLB to provide a very small miss ratio (e.g., 0.1% or intrinsic).

Table 3 presents the TLB dimensions required to guarantee a 0.1% miss ratio for each benchmark. (The sizes are presented in units of *page-table entries* instead of *bytes*, since the size of a page-table entry is dependent on the size of the virtual memory system.) In general the TLB size requirement decreases with the increase in page size. This is due to the fact that larger page sizes result in a fewer number of active

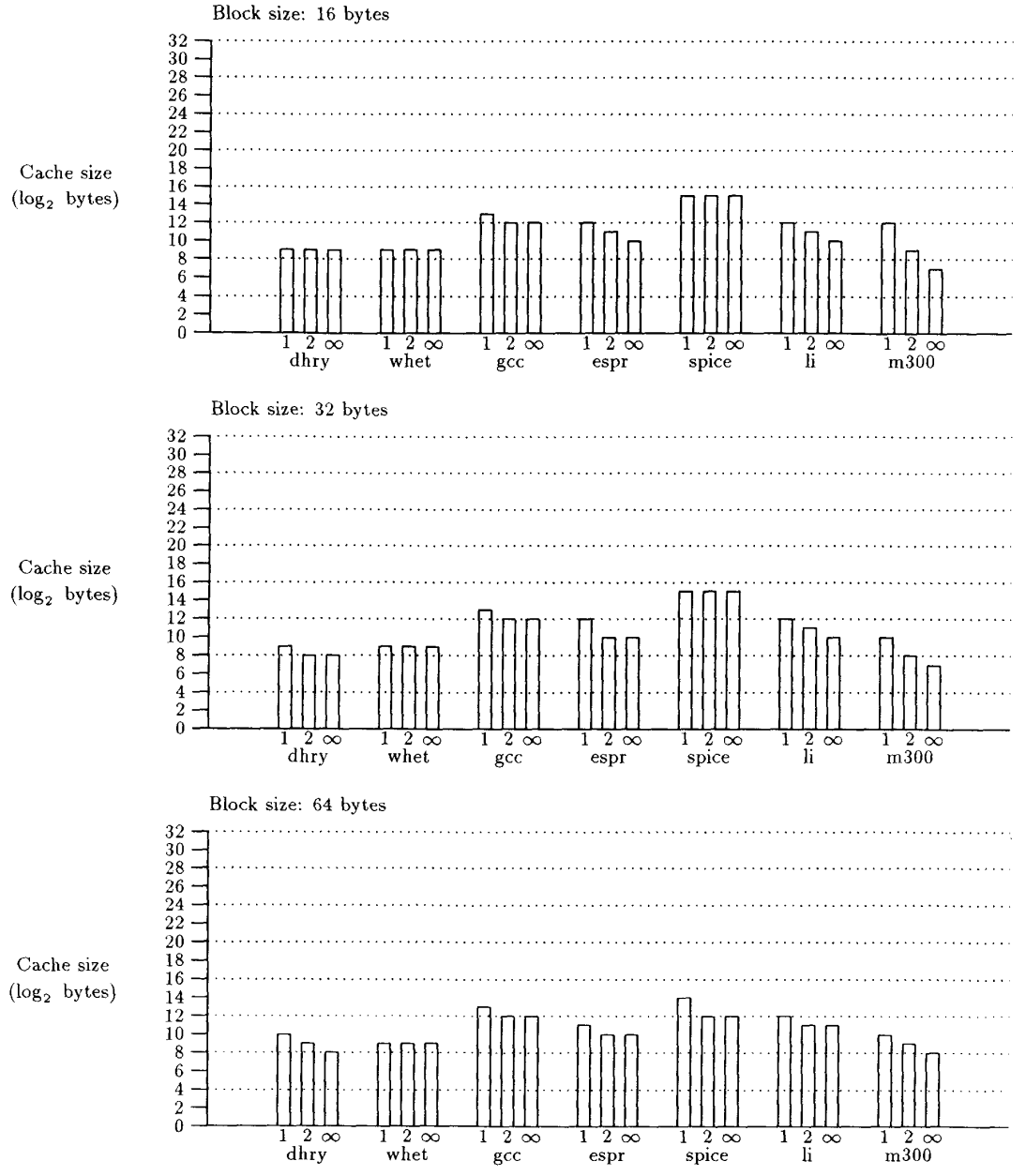

Figure 2: Minimum cache sizes that guarantee a maximum 10% miss ratio.

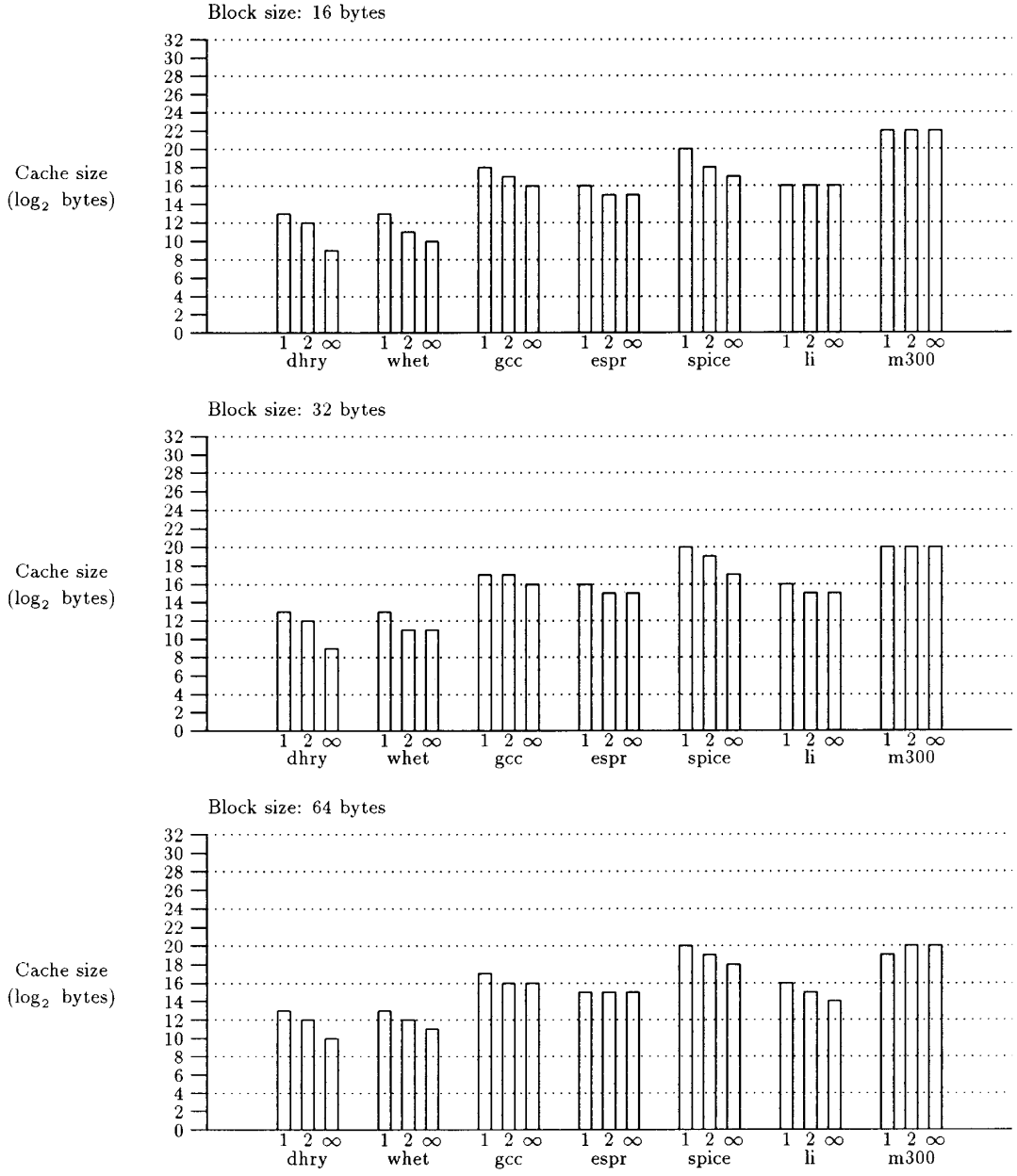

Figure 3: Minimum cache sizes that guarantee a maximum 1% miss ratio.

Table 3: TLB sizes (in number of page-table entries) for page sizes 512 bytes, 1KB, 2KB, and 4KB (see text).

	Number of page-table entries							
	Page size: 512B				Page size: 1KB			
	Set size				Set size			
Benchmark	1	2	4	∞	1	2	4	∞
dhrystone	32	32	16	8	16	16	16	8
whetstone	128	32	32	16	64	16	16	16
gcc	4096	1024	1024	1024	2048	512	512	512
espresso	512	128	128	64	256	64	64	32
spice	2048	2048	2048	2048	2048	1024	1024	1024
li	256	128	128	64	128	64	64	64
matrix300	2048	2048	2048	2048	1024	256	16	16

	Number of page-table entries							
	Page size: 2KB				Page size: 4KB			
	Set size				Set size			
Benchmark	1	2	4	∞	1	2	4	∞
dhrystone	32	16	16	8	16	8	8	8
whetstone	64	64	8	8	32	32	8	8
gcc	1024	512	256	256	1024	256	128	128
espresso	256	64	32	32	128	32	16	16
spice	1024	512	512	512	512	256	256	256
li	256	64	32	32	128	32	32	16
matrix300	512	128	16	16	512	64	16	8

pages. Therefore, the TLB has to accommodate a fewer number of address translation results.

An interesting phenomenon is that *spice* is not the most demanding benchmark for TLB design. Although *spice* consistently requires more than four times the cache size than *gcc*, their TLB requirements are very comparable. This is due to the fact that *gcc* accesses a comparable number of active pages to *spice* but accesses a much smaller number of blocks within these active pages. This is a good example of a demanding benchmark for TLB design not being the demanding benchmark for cache design.

3.2.3 Main memory requirements

The design of main memory differs from that of cache memory in several ways. First of all, the main memory design is fully associative rather than set associative as a result of the nature of modern virtual memory systems. Secondly, the page size is usually much larger than the cache block size due to address translation and disk access considerations. Thirdly, the page fault penalty is much higher than the cache miss penalty. A common goal in main memory design is to achieve the intrinsic page fault rate of programs. The intrinsic page fault rate is the fault rate of an infinite main memory. All intrinsic page faults are due to the loading of accessed pages into the infinite main memory. The intrinsic page fault rate of program is a function of the page size.

The main memory sizes required to achieve intrinsic page fault rate for each benchmark are shown for four page sizes (512B, 1KB, 2KB, and 4KB from left to right) in Figure 4. The sizes required to cover all benchmarks are 4MB, 4MB, 4MB, and 8MB for page sizes 512B, 1KB, 2KB, and 4KB respectively. The main memory requirement increases with the page size due to internal fragmentation. However, the increase may be justified by smaller TLB sizes and simpler physical cache design.

3.3 Operating system benchmark characteristics

The operating system's interface to any program is through the system call mechanism, which is a procedure call into the kernel. How often particular calls are used can impact how the system software is designed for high-performance and also provide insight into I/O system design. The most-frequently used system calls are the best candidates for streamlining. Table 4 lists the system calls used by each benchmark

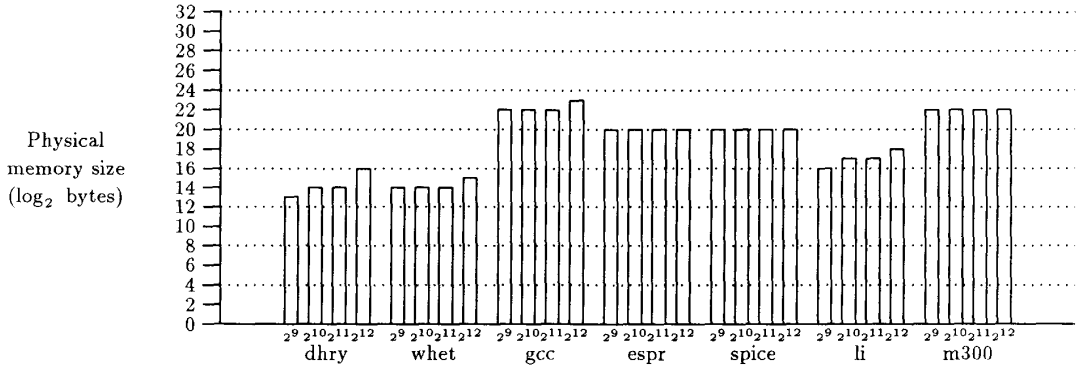

Figure 4: Main memory sizes required to achieve intrinsic miss ratio for page sizes 512B (2^9), 1KB (2^{10}), 2KB (2^{11}), and 4KB (2^{12}).

Table 4: Dynamic system call usage.

Benchmark	System calls
dhrystone	*getrusage (8), sbrk (3), fstat (1), getpagesize (1), ioctl (1)*
whetstone	*none*
gcc	*getrusage (6072), write (164), sbrk (66), read (15), open (2), close (2), fstat (2), ioctl (2), getpagesize (1)*
espresso	*write (27), sbrk (18), read (4), ioctl (2), fstat (2), open (1), getpagesize (1)*
spice	*write (784), lseek (6), fstat (5), sbrk (3), close (3), read (3), ioctl (3), getpagesize (1)*
li	*sbrk (7), ioctl (2), fstat (2), open (1), getpagesize (1), read (1)*
matrix300	*lseek (12), fstat (5), close (5), stat (3), sbrk (3), write (3), ioctl (2), open (2), getdtablesize (1), getpagesize (1)*

and the number of dynamic occurrences of each call (in parentheses). Note that the benchmark *whetstone* makes no system calls whatsoever. This is understandable due to its synthetic nature. The *getrusage* system call that is prominent for *dhrystone* and *gcc* is used by the *times* library call to report run times. The *sbrk* call is used in the heap space allocator. Heap space is used considerably in *li*, *espresso*, and *dhrystone*. I/O intensive benchmarks reveal themselves here by the use of the *read, write, open, close, fstat, lseek,* and *ioctl* calls. These benchmarks are *spice*, and *gcc*.

4 Concluding Remarks

Benchmark characterization reduces the cost of exploring the design space, focuses the experimental system towards the intended workload, and lessens the amount of simulation and redesign required. This paper presents designs for a system based on the benchmark characteristics of a set of popular benchmarks. The construction of an abstract system model and the characterization method were made possible in-part by the advance of architecture-independent compiler technology.

It is interesting to wonder how to apply benchmark characterization to parallel machine design. Benchmarks themselves are specific to a particular type of parallel architecture. For example, programs written for message-passing multicomputers would perform poorly on tightly-coupled shared-memory multiprocessors. Therefore, benchmark characterization must be done for each type of architecture. Consider a tightly-coupled shared-memory multiprocessor. Extensions to our abstract system would include a distributed shared memory hierarchy and synchronization events. We are currently implementing such extensions using the *Perfect Club* [13] as a sample

benchmark set.

Acknowledgements

The authors would like to thank Sadun Anik, Andy Glew, Dave Griffith, Isadora Parrini, and all members of the IMPACT research group for their support, comments and suggestions. Special thanks to Jim Larus for use of AE.

This research has been supported by Dr. Lee Hoevel at NCR, the National Science Foundation (NSF) under Grant MIP-8809478, and by an equipment donation from the Hewelett-Packard company.

References

[1] T. M. Conte and W. W. Hwu, "Benchmark characterization for experimental system evaluation," in *Proc. Hawaii Int'l Conf. on System Sciences*, vol. 1, (Kona, Hawaii), pp. 6–18, Jan. 1990.

[2] R. M. Stallman, *Using and porting GNU CC.* Free Software Foundation, Inc., 1989.

[3] P. P. Chang and W. W. Hwu, "Inline function expansion for compiling C programs," in *Proc. 1989 ACM Conf. on Prog. Lang. Design and Implementation*, (Portland, OR), June 1989.

[4] J. Mashy, "Your milage may vary," tech. rep., MIPS Computer Systems, Inc., Sunnyvale, CA, 1990.

[5] "Spec newsletter," Feb. 1989. SPEC, Fremont, CA.

[6] C. L. Mitchell and M. J. Flynn, "A workbench for computer architects," *Design & Test*, pp. 19–29, Feb. 88.

[7] J. R. Larus, "Abstract execution: a technique for efficiently tracing programs," tech. rep., Computer Sciences Department, University of Wisconsin–Madison, Feb. 1990.

[8] G. D. McNiven and E. S. Davidson, "Analysis of memory reference behavior for design of local memories," in *Proc. 15th. Annu. Int'l Symp. on Comput. Arch.*, (Honolulu, HI), pp. 56–63, June 1988.

[9] D. W. Wall and M. L. Powell, "The Mahler experience: using an intermediate language as the machine description," in *Proc. Second Int'l Conf. on Architectural Support for Prog. Lang. and Operating Systems.*, (Palo Alto, CA), pp. 100–104, Oct. 1987.

[10] S. I. Feldman, D. M. Gray, M. W. Maimore, and N. L. Schryer, "A Fortran-to-C converter," Computing Science Tech. Report 149, AT&T Bell Laboratories, Murray Hill, NJ, June 1990.

[11] T. M. Conte and W. W. Hwu, "Single-pass memory system evaluation for multiprogramming workloads," Tech. Rep. CSG-122, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, May 1990.

[12] A. J. Smith, "Cache memories," *ACM Computing Surveys*, vol. 14, no. 3, pp. 473–530, 1982.

[13] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck, "Supercomputer performance evaulation and the Perfect Club," in *Proc. Int'l Conf. on Supercomputing*, (Amsterdam, The Netherlands), pp. 254–266, June 1990.