Special Brief Papers

Multilevel Optimization in the Design of a High-Performance GaAs Microcomputer

O. A. Olukotun, R. B. Brown, R. J. Lomax, T. N. Mudge, and K. A. Sakallah

Abstract—**The design of microelectronic systems has traditionally been carried out at several levels of abstraction. Partitioning the design process into levels makes it more manageable, but usually results in a suboptimal design. When high performance is the goal, optimization should be done across multiple abstraction levels. This paper illustrates multilevel optimization in the design of an instruction cache for a high-performance GaAs microprocessor. Performance of the system is maximized by concurrently considering the interrelationships of: 1) the time of flight of signals across the multichip module on which the processor and cache chips are mounted; 2) the clocking scheme that synchronizes these signals; and 3) the size of the cache. These three design issues are normally considered independently because they arise in different abstraction levels. Design automation tools developed to facilitate this multilevel optimization are described. This process, applied to various subsystems, has been used to gain substantial performance improvement in the GaAs microcomputer.**

I. Introduction

MANAGING the design of a computer system is greatly simplified by partitioning the design into a hierarchy of abstraction levels (e.g., transistor, logic, architecture, and language levels [1]) which can be treated with some degree of independence. Designs are typically carried out by several groups of designers, each having responsibility for one of these levels. The use of abstractions is necessary for dealing with the complexity of microelectronic system design; however, the indiscriminate application of this approach leads to suboptimal computer designs.

As system performance goals increase, the inefficiencies introduced by treating the various abstraction levels independently become significant. Without abandoning the advantages of partitioning the design process, selected optimizations across traditionally separate abstraction levels can be performed to achieve better overall system performance. This approach is referred to herein as *multilevel optimization*. Multilevel optimization requires concurrent consideration of all design levels over which the optimization is to take place, rather than allowing, for example, a top-down methodology, in which the design at each level dictates the specifications of the next level down the hierarchy. Multilevel optimization relies heavily on the ability to simulate performance at each abstraction level in order to evaluate the impact of design trade-offs on system performance, and is characterized by the need for iteration as the design takes form on several levels at once.

This paper describes a typical example of multilevel optimization in the design of an instruction cache for a high-performance GaAs microcomputer currently under development at the University of Michigan [2]. In particular, the performance of this system, as defined by its instruction execution rate, has been maximized by concurrently optimizing the circuit, timing, and architectural levels of the design. The instruction cache subsystem is used to illustrate this multilevel optimization; numerous other examples of the design technique could be cited from this project.

Section II provides background information which puts the discussion into perspective. The optimization problem is formulated in Section III. Section IV details the multilevel optimization procedure and some of the CAD tools developed to support it. Section V summarizes the benefits and challenges of multilevel optimization.

II. Background

The University of Michigan GaAs microcomputer implements the MIPS instruction-set architecture. This RISC architecture is appropriate for implementation in GaAs because of its simplicity, and it is well-suited to executing the programs expected in the target environment, that of an engineering workstation. Use of the standard instruction set enables the computer to use the MIPS Computer Systems, Inc. operating system and language compilers with few modifications, as well as allowing it to execute application programs written for MIPS processors. The desire for software compatibility narrows the design space,

Manuscript received September 24, 1990; revised January 10, 1991. This work was supported by the U.S. Army Research Office under the URI Program, Contract DAAL03-87-K-0007, and by the Defense Advanced Research Projects Agency under DARPA/ARO Contract DAAL03-90-C-0028.

The authors are with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122.

IEEE Log Number 9143200.

0018-9200/91/0500-0763$01.00 ©1991 IEEE

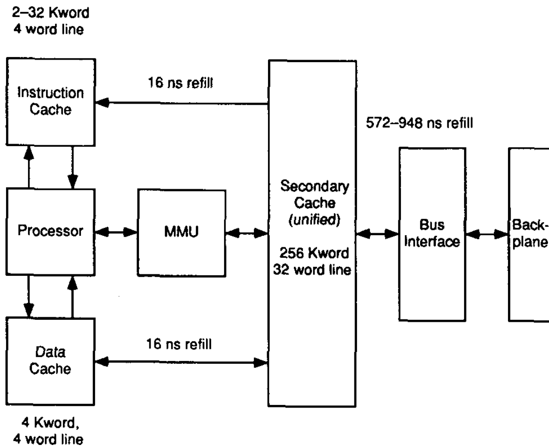

Fig. 1. System block diagram.

constraining the architectural options, and in many ways reducing design effort and risk in the project.

Due largely to the increased speed and density of scaled CMOS technologies, microprocessors have sustained an increase in performance of 2–3 times every three years (compared to 1.5 times for supercomputers) [3]. Switching speed and integration levels were important considerations in selecting GaAs direct-coupled FET logic (DCFL) from Vitesse Semiconductor Corporation as the technology for this design. The 32-b CPU and 64-b floating-point accelerator (FPA) defined by the MIPS architecture will be implemented in this project on a single chip. As microprocessors such as this one move into the performance range of supercomputers, advanced packaging technology is required to prevent unavoidable chip crossings from dominating system delay. Multichip module (MCM) technology is used to reduce delay on the critical path between the processor and cache memory, allowing the speed of GaAs to be reflected in system performance.

Fig. 1 shows a block diagram of the GaAs microcomputer. This organization resulted from extensive architectural studies using a multilevel cache simulator, **cacheUM** (see Section IV-C). The main components are the processor chip (CPU and FPA), direct-mapped primary and secondary caches, a memory management unit (MMU), and a bus interface chip that connects the processor to the primary memory and I/O bus. Bandwidth considerations dictated that the primary cache be split into data and instruction sections that can be accessed simultaneously. The MMU chip combines the functions of a write buffer between primary and secondary cache, virtual-to-physical address translation, and cache control.

III. The Problem

Our objective is to maximize the performance of the GaAs microcomputer in terms of the average number of instructions it can execute per second. This figure is usually expressed in units of MIPS, millions of instructions per second, defined by

$$\text{MIPS} = \frac{10^3}{T_c \times \text{CPI}} \tag{1}$$

where T_c is the period of the processor clock in nanoseconds and CPI is clock cycles per instruction for a representative mix of application programs. The optimization problem is therefore equivalent to minimizing $T_c \times \text{CPI}$. These variables arise in different abstraction levels, and changes in design parameters can affect them in opposite ways. To optimize system performance, the effects must be considered concurrently.

In general, the principal parameters affecting T_c and CPI are the total time taken to fetch instructions from the primary instruction cache, the size of the cache, and the number of branch delay slots. These parameters are all related. Clearly, T_c is an increasing function of cache access time. Access time is, in turn, an increasing function of cache size in a given technology, due to longer access times in larger memory chips and longer time of flight in memory systems having more chips. If access times of secondary cache and main memory remain constant, then CPI decreases with increasing T_c because fewer cycles are required to access these levels of memory. CPI is generally a decreasing function of cache size, because larger caches have lower miss rates.

The effect of branch delay slots on T_c and CPI is less obvious. Fig. 2(a) illustrates how branch delay slots arise. On the left side of the diagram, the two possible targets of the branch instruction, **PC + 1** and **PC + Branch Offset**, are calculated, but selection of the appropriate address must wait for the output of the comparator on the right (evaluation of the branch condition). Delay slots allow this evaluation and selection to extend beyond a single cycle. The MIPS instruction set architecture (ISA) being implemented in this project dictates one branch delay slot. In general, CPI is an increasing function of the number of delay slots because techniques such as branch prediction and reordering of instructions to fill these slots are not always successful. T_c tends to be inversely proportional to the number of branch delay slots because delay slots create a pipeline loop through the instruction cache and the processor, as illustrated in Fig. 2(b) for a one-slot branch delay.

Our desire to maintain software compatibility with the MIPS instruction set architecture [4] constrains our design to having exactly one branch delay slot, leaving cache size and access time as the only designable parameters. As mentioned above, cache access time is a function of cache size, so the performance model can be stated in terms of the following three equations:

$$\text{CPI} = f(T_c, S_{\text{cache}}) \tag{2}$$

$$T_c = g(t_{\text{cache}}) \tag{3}$$

$$t_{\text{cache}} = h(S_{\text{cache}}) \tag{4}$$

where t_{cache} is the total time taken to fetch instructions from the primary instruction cache and S_{cache} is primary

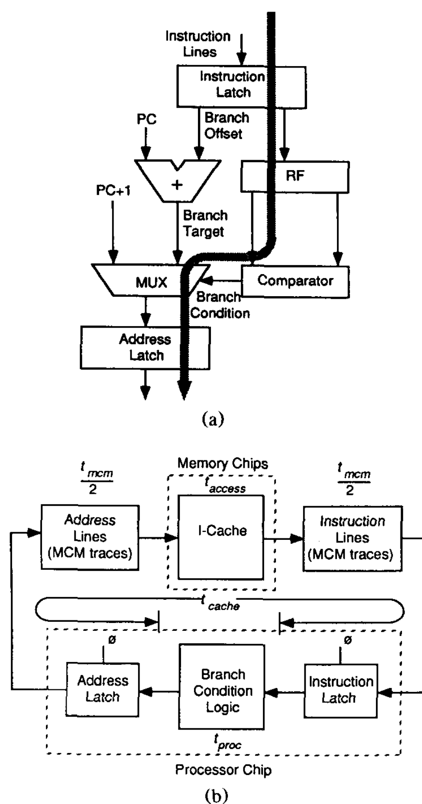

Fig. 2. (a) Branch instruction logic. (b) Critical path through the instruction cache.

cache size. These three functions correspond, respectively, to the architectural, timing, and circuit levels. CPI decreases as cache size increases, but t_{cache} and T_c increase with cache size. The remainder of this paper is devoted to the development of these three functions, and to minimizing the product $T_c \times \text{CPI}$.

IV. The Levels of Optimization

A. Circuit Level

T_c is critically dependent on t_{cache}, so multichip module packaging is used to minimize the off-chip delay time while simultaneously maintaining signal integrity. The semiconductor chips are mounted on the MCM and interconnected by conductors embedded in low-permittivity dielectrics on a rigid substrate material. The MCM also provides photolithographically defined terminating resistors. The signal lines are sandwiched between ground and power planes, and run predominantly in orthogonal directions. These interconnections are designed to behave as 50–70-Ω stripline transmission lines.

Fig. 2(b) identifies the delay of each component in the two-stage pipeline loop connecting the processor and the instruction cache: t_{proc} represents the critical delay through the branch condition logic on the processor chip; t_{cache} models the overall memory access time, and is the sum of t_{mcm}, the round-trip signal propagation time on the MCM, and t_{access}, the access time of the cache itself. Both the processor delay and the round-trip signal propagation time on the MCM have been determined by simulation using a version of SPICE 2G.6 with Vitesse-proprietary models for enhancement and depletion MESFET's [2]. The 1-kb register file and the arithmetic logic unit from the processor data path have been fabricated and tested. Measured delay times from these circuits, composed of 16 085 and 3419 transistors, respectively, have verified the circuit simulations. For example, propagation times for the 32-b adder, predicted to be 2.5 ns, were measured on four chips at 2.0, 2.3, 2.5, and 2.5 ns [5].

SPICE simulation of the processor chip showed that the worst-case delay for the branch condition logic is 3.0 ns, arising from the read setup (1.2 ns), read from the register file (0.4 ns), a quick compare in the arithmetic logic unit (1.3 ns), and a multiplexing operation (0.1 ns). This is the shaded path shown in Fig. 2(a). The primary cache will be made of custom GaAs 1K×32-b SRAM chips, which are expected to have access times of 3 ns, exclusive of buffer delays. (In this analysis, I/O drivers and receivers are included in the parameter t_{mcm}.)

Preliminary layouts of the MCM were used to estimate t_{mcm}. At a clock rate of 250 MHz, interconnects on the MCM behave predominantly as *RC* transmission lines, but inductive effects are not completely negligible. Transmission-line effects were included by modeling the interconnect for 1 b of a bus with a ladder network consisting of a lumped series inductor and resistor together with a shunt capacitor for each millimeter of interconnect [2]. Input pads were represented by diode-connected MESFET's and parallel capacitors. The chips were interconnected using stubs branching off the transmission line, which was terminated with a 50-Ω resistor. While these layouts were only first approximations and the SPICE models are imperfect representations of MCM interconnect, the simulation results are reasonably accurate, and they allow the impact of cache size on t_{cache} to be evaluated. Such preliminary analysis is common in multilevel optimization because trade-offs are evaluated before the design is fully specified at any of the abstraction levels. At each iteration of the design, the simulations yield more accurate results.

The top line in Fig. 3 is the total cache access time t_{cache} for cache sizes ranging from 2K words to 32K words. This curve completely characterizes the function h in (4). It is the sum of the constant cache access time (t_{access}) of 3 ns and the time of flight on the MCM for each cache size.

B. Timing Analysis

The performance level of this processor requires that the MCM and all of the MCM-mounted components be viewed from a timing perspective as a single entity. While

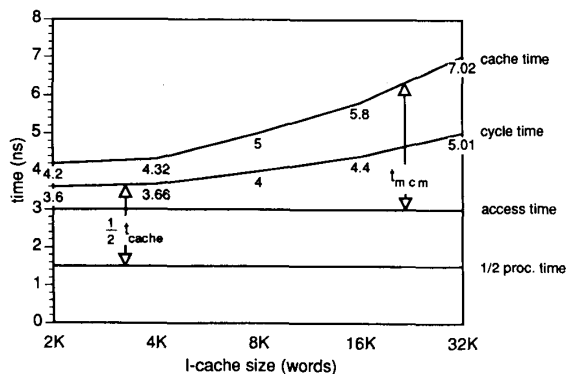

Fig. 3. The relationship between cache time, cycle time, and cache size.

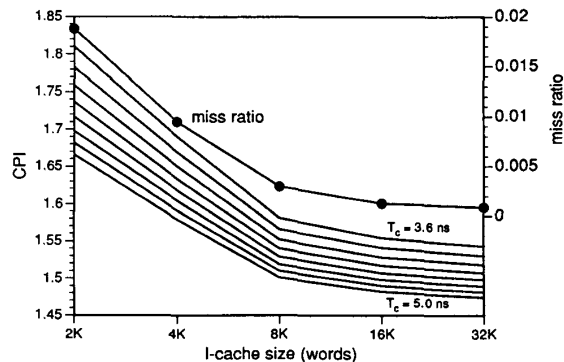

Fig. 4. Architectural performance versus cache size for various values of system cycle time that vary from 3.6 to 5.0 ns in 0.2-ns intervals. The left axis shows CPI and the right axis shows the instruction cache miss ratio.

elaborate interchip signaling protocols simplify the task of combining off-the-shelf chips, they also reduce performance by increasing chip-crossing delays. In this design, with all custom chips, the interfaces can be tailored to achieve the highest possible performance. Such optimization requires accurate timing analysis techniques. We have developed two new timing analysis tools, *check*T_c and *min*T_c for this purpose [6]. The first of these, *check*T_c, is a timing verifier which examines a circuit to see if it satisfies a specified clock schedule, and reports setup- and hold-time violations. The second tool, *min*T_c, is a clocking optimizer which determines the optimal clock schedule (i.e., the schedule with the minimum cycle time) that satisfies all timing constraints of a given circuit.

The cycle time curve of Fig. 3 shows the relationship between optimal cycle time and cache size. This relationship was derived by running *min*T_c on the critical path between the processor and the primary instruction cache (Fig. 2(b)) using the appropriate value of t_{cache} for each cache size. To achieve the optimal cycle time, the address and instruction latches in Fig. 2(b) must be clocked so that a signal arriving at the input of a latch is gated through the latch immediately. If the signal is delayed by the clock, a nonoptimal cycle time will result. *Min*T_c ensures that signals never wait along the critical path and guarantees that the signals at all latches in the system meet setup- and hold-time requirements.

The optimal cycle time is inversely proportional to the number of stages in the critical path pipeline. Therefore, the two-stage critical path of Fig. 2(b) results in a proportionality constant of one-half between increases in t_{cache} and increases in T_c. This factor is clearly seen in the piecewise linear slopes of the cache time (t_{cache}) and cycle time (T_c) curves of Fig. 3. The horizontal line at 1.5 ns represents $t_{\text{proc}}/2$. The interval between this line and the T_c curve is $t_{\text{cache}}/2$. The function g in (3) can now be defined as

$$T_c = \frac{t_{\text{proc}} + t_{\text{cache}}}{2}. \tag{5}$$

C. Architectural Level

To determine CPI for a particular instruction cache size, the cache is simulated using address traces from real application programs as input stimuli. The trace-driven cache simulator we have developed to do this is called **cacheUM** [2]. **CacheUM** models all aspects of the memory system shown in Fig. 1. Performance of a memory system varies with the application program. To make these simulations as realistic as possible, a mix of integer and floating-point applications was used to represent the work load of a high-performance workstation used in a technical environment. The benchmark suite consisted of these 16 programs: Matrix500, awk, diff, doduc, dhrystone2, espresso, gnuchess, grep, integral, linpack, Livermore loops, nroff, small, spice2g6, Timberwolf, and yacc. Cache performance for each of these benchmarks is listed in [2]. Their execution required about 2.5×10^9 processor cycles. To simulate a multiprogramming environment, context switches among benchmarks are scheduled whenever a system call is executed or a fixed-process time slice of 500 000 cycles expires. The CPI value of a memory system is calculated by dividing the total number of cycles by the number of instructions executed by all benchmarks.

Every line in the cache is accompanied by a tag word that specifies which line of main memory is currently in the cache. A primary- or secondary-cache miss causes a new cache line to be read from the next memory level, and the corresponding tag word to be updated. In this design, the primary instruction cache is refilled from the secondary cache one four-word line at a time using a wide data bus. Because it allows the cache to be refilled quickly, this scheme significantly reduces the miss penalty.

The values of CPI are shown in Fig. 4 for five instruction cache sizes (not including tag words) and for system cycle times varying from 3.6 to 5.0 ns. As will be seen below, some of the cycle time/cache size combinations are not physically realizable. Fig. 4 shows that CPI decreases with increasing cache size and increasing system cycle time as expected. The dependence of CPI on cycle

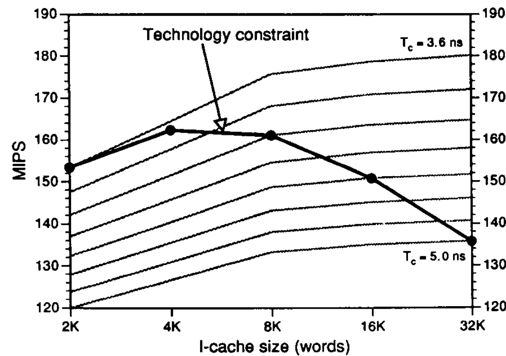

Fig. 5. System performance versus cache size. The system performance for each cache size is shown by the *Technology Constraint* curve.

time is less pronounced at larger cache sizes because the number of cycles it takes to refill cache is only a factor when there is a cache miss. The miss rate is smaller for large caches, so system cycle time has a smaller effect on CPI. Though the plot only shows instruction cache sizes to 32K words, the slopes of the curves are still decreasing. This indicates that larger caches will have even lower miss rates. The function f in (2) is thus given by Fig. 4.

D. Multilevel Optimization

With functions f, g, and h of (2)–(4) specified, all the information is available to determine the design point that has the best overall system performance. The optimal design point is found by selecting the cache size S_{cache} that minimizes the product $T_c \times \text{CPI}$. Fig. 5 plots the MIPS rating as a function of primary instruction cache size, for various cycle times. These plots show the trade-off between system cycle time and instruction cache size. The *Technology Constraint* curve of Fig. 5 is the inverse of $T_c \times \text{CPI}$, or the MIPS rating of the system constrained by all of the circuit, timing, and architectural effects described. Only the values below this line are physically realizable. The optimal cache size is readily seen from this curve to be 4K words. The figure suggests that, for the benchmarks used in this analysis, the system will have a MIPS rating of 162. It is interesting to note that this cache size is much smaller than that suggested by architectural considerations alone (see Fig. 4), and larger than that which yields the smallest cycle time T_c suggested solely by circuit/timing considerations (see Fig. 3).

V. Concluding Remarks

In this paper we have demonstrated, with one example, that optimization of high-performance microelectronic systems can only be achieved by concurrent consideration of multiple design levels. This design process is more demanding than the traditional top-down or bottom-up approaches because it requires a global view of the design space, and calls for iterative simulations of design trade-offs as the design evolves to its final form. In the example detailed here, architectural, timing, and circuit levels are concurrently evaluated to optimize the instruction cache size of a GaAs microcomputer. We believe that multilevel optimization will become increasingly necessary as demands for ever-increasing performance continue. Tools and methodologies that go beyond traditional approaches must be developed to support this extension to microelectronic circuit design.

Acknowledgment

The authors thank MIPS Computer Systems for supporting this project with technical assistance under a special licensing arrangement. We also gratefully acknowledge the assistance of Vitesse Semiconductor Corporation, Seattle Silicon Corporation, and Mentor Graphics Corporation.

References

[1] A. S. Tanenbaum, *Structured Computer Organization*, 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1990.
[2] T. N. Mudge *et al.*, "The design of a microsupercomputer," *Computer*, pp. 57–64, Jan. 1991.
[3] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann, 1990.
[4] G. Kane, *MIPS RISC Architecture*. Englewood Cliffs, NJ: Prentice-Hall, 1989.
[5] J. A. Dykstra, "High-speed microprocessor design with gallium arsenide very large scale integrated digital circuits," Ph.D. dissertation, Univ. of Michigan, Ann Arbor, 1990.
[6] K. A. Sakallah, T. N. Mudge, and O. A. Olukotun, "Analysis and design of latch-controlled synchronous digital circuits," in *Proc. 27th ACM/IEEE Design Automation Conf.*, June 1990, pp. 111–117.