An Analysis of Instruction-Cached SIMD Computer Architecture

Todd E. Rockoff

December 1993

CMU-CS-93-218

School of Computer Science
Computer Science Division
Carnegie Mellon University
Pittsburgh, PA

Submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Thesis Committee:
Allan L. Fisher, Chairman
Norman P. Jouppi
Robert F. Sproull
Peter A. R. Steenkiste

Keywords: Data-parallel computation, SIMD computer architecture, VLSI computer architecture, total chip area, throughput-to-area ratio, SIMD instruction cache, generic SIMD computer, multi-clock SIMD computer, I-cached SIMD computer, instruction cache management

Abstract

In a single instruction-stream/multiple data-stream (SIMD) computer, calculations are performed by simple processing elements (PEs) that are not independently capable of program-control operations. In lock-step, the PEs execute one program that is sequenced by a single system controller. Large numbers of these simple PEs are obtained through replication of a PE chip containing many identical PEs.

A state-of-the-art SIMD computer is regulated by a single system clock that is distributed throughout the computer. On each system clock cycle, the system controller broadcasts the next instruction to be executed by the PEs. The system clock interval allows time to distribute a PE instruction throughout the computer, an action that typically requires more time than the minimum interval of a clock regulating the PEs themselves within the PE chips. The disparity between the highest rate of PE operation and the rate of global instruction broadcast gives rise to a heretofore un-compensated clock-rate limitation.

To overcome this limitation, *instruction-cached SIMD computer architecture* provides for a small instruction buffer to be placed within the replicated PE chip. This buffer stores repeated instruction sequences for subsequent retrieval at the relatively high rate attainable within the PE chip. The instruction buffer and its control mechanism comprise a SIMD instruction cache, or *I-cache*.

Throughput measures computer performance, while total chip area is a primitive measure of a computer's monetary cost that is most appropriate for high-PE-count multiprocessors. The ratio of throughput to area is a figure of merit expressing a multiprocessor's ratio of performance to hardware cost. My thesis is that even the simplest I-cache variants increase throughput-to-area ratios significantly.

I-cache increases the rate at which instructions are delivered to the PEs. However, I-cache takes up space in the PE chip that would otherwise have been used for PEs themselves. If the total chip area of the computer is fixed, then I-cache reduces the chip area used for PEs. For scalable data-parallel problems, reducing the number of PEs reduces the throughput. Therefore, the magnitude of the throughput-to-area ratio impact of I-cache depends on the balance between the competing factors of instruction execution-rate increase and displacement of PEs from the PE chip.

In analyzing SIMD instruction cache, the dissertation considers the interacting characteristics of data-parallel program properties and SIMD computer electrical characteristics in I-cache design. Designs of two simple I-cache variants are presented and their chip-area costs are estimated. For a diversity of data-parallel problems, these I-caches are evaluated over a range of underlying PE architecture and system characteristics. The evaluations are performed using register-transfer-level simulations of SIMD computations on a clock-phase by clock-phase basis.

The simple I-cache variants occupy negligible chip area and yet yield speedups ranging from 1.3 to 7.6 for the simple problems under realistic assumptions about the electrical characteristics of the SIMD computer. The computations for which these results are obtained include such problems as sorting and tree-reduction which are commonly thought to be inherently inter-PE-communication-bound because of their characteristic frequency and complexity of inter-PE communication. The simulation results suggest that there is a wide range of problem characteristics over which I-cache makes for the best use of chip area in a SIMD computer.

Reason with Love from Beauty.

Acknowledgments

I thank my thesis committee chairman Allan Fisher for providing an opportunity to make a SIMD PE chip, for getting me thinking about fast SIMD computers, and for helping me to identify what was of scientific value. Allan saw me safely through the Black Friday mine-field, when the odds were long against my making it through alive. I thank Allan especially for his stories about the less-restrained days back in Jersey, which stories were reassuring to me at important times.

Committee member Bob Sproull read very carefully and suggested numerous specific improvements. Bob understood the basic idea of I-cache very early on, and he encouraged me at times when my confused ramblings were easily construed as wrong-minded craziness. Bob's intuition is excellent, and his advice has been unerring throughout.

Committee member Peter Steenkiste required me to make sense. Peter asked probing questions whose answers clarified my results. Peter gave me the benefit of the doubt when I needed it.

Committee member Norm Jouppi, an experienced designer of very fast computers, woke me up initially to the possibilities of PLL-based clock generation for fast chips. More recently, Norm taught me about transmission lines, and thereby greatly improved the dissertation's treatment of fast instruction broadcast options.

My "shadow committee" — Chris Marlin, Skef Wholey, Marc Donner, and Wes Clark, provided well-timed support and continual encouragement that boosted me over a succession of obstacles, both internal and external. Skef provided much-needed information about commercial SIMD computers and their programming. I am especially indebted to Chris for his logistic support and strategic guidance through some rough patches. Chris loaned me the proximal courage to begin writing *chapters*. The top-down approach advocated by Chris came at just the right time to complement the bottom-up relevant-fact-enumeration method practiced by Wes. The goods lay in the middle. Wes Clark, Marc Donner, Doug Clark, and Richard Wallace together helped to sharpen my thesis defense talk during a halcyon afternoon in the Eagles' Nest.

Klaus Gross reminded me that I had to write an outline eventually. Klaus also demonstrated to me for years the importance of living in firm contact with one's animal nature: life is a bushwalk.

Brad White, Dean Rubine, Michael McCarthy, Bennet Yee, and Bert Enderton each gave generously in response to my recurrent pleas for C and Unix programming help. More competent, mellow crew would be hard to find. Michael McCarthy, in keeping my tools sharp, gracefully accepted far more whinging and far less praise than he deserved. Dean and Sid Chatterjee gave helpful LaTeX advice. Dale Moore and Roy McDougall helped me out with practical problems printing PostScript. James Tizard helped early on with plotting tools. Wes Clark provided a remarkable suite of custom PostScript plotting tools.

I enjoyed many varied interactions over a period of about 6 years at CMU Computer Science. I hope never to relinquish the Reasonable Person Principle. I thank my Pittsburgh office mates for years of tolerance: Randy Brost, Robert Stockton, Bert Enderton, Bennet Yee, Chris Lebiere, Dave Abramowitz, and Greg Nelson. For their help in maintaining my sanity, I am grateful to Blaine Burks, Charlie Krueger, Eric Nyberg, Dean Rubine, and Brad White, fine musos all who never turned off my mikes even during the wailingest of jams. For providing an opportunity to vent my frustrations in a relatively safe manner, I am indebted to Tom Sullivan and to all the stalwart fore-checkers who turned up for the weekly hockey games.

Most of this research was conducted while I was an Honorary Visiting Scholar with the Computer Science Discipline at Flinders University in Adelaide, Australia. Ivan Sutherland suggested that Australia would be a good place to develop perspective. The initial move was facilitated by a generously funded CSIRO position, boldly tendered by Craig Mudge. The gentle folks at Flinders Uni were extremely tolerant through the last year and a half of work. I am especially grateful to my supportive friends downunder, especially Chris Marlin and Paul Calder, for helpful discussions, for

providing advice, and for asking insightful questions. Gary Glonek generously helped me through the statistical analysis of the I-cache speedup data. Bill McFarlane posed important questions that were hard to ask. Dennis Jarvis, Jon Baxter, Neil Bergmann, Jim Entwisle, and Stuart Beck read some of what I'd written and offered helpful comments.

I am grateful to my parents, Maxine L. Rockoff and S. David Rockoff. They have always made it possible for me to pursue the ideal of truth without compromise. I discovered on my own the apparently unavoidable necessity of compromise, but I count myself among the luckiest for the latitude I have almost always had in choice-making. I further thank Maxine for maintaining such close contact, through frequent email, and for her sharp editorial comments on chapter drafts. Thanks, Ma!

I also thank the remainder of my family, especially Lisa, Kevin, and Jacqueline for their constant love and support, of which I am always aware.

This work has benefited enormously from the comprehensive mentoring generously provided by Wes Clark. With his unique clarity of thought, Wes gently guided me in distilling an idea soup into the bullion of a thesis proposal. Over the subsequent 3 years, he persistently asked hard questions about the details and insisted on clear consistency. Wes' PostScript programming made some of the figures in the dissertation look good, while his design sense helped me to improve others. It is not possible for me to overstate the improvements in prose and in modes of exposition that resulted from Wes' tireless editing and suggesting. Any reader who is sufficiently committed to work through the details herein should hesitate not to believe that the going would have been far more difficult had Wes not graciously intervened. Thanks, Wes!

Finally, Trudie, my love. May kindness and tenderness continue to bind us, always.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Some computational problems demand the fastest possible computers. The two main technologies used for attaining maximum throughput are VLSI and parallelism. VLSI is a fabrication technology that exploits the inherent speed and cost advantages arising from packing computer system components together inside chips. Parallelism is an organization technology that exploits the speed advantages of harnessing the collective efforts of large numbers of computational units. These computational units are processing elements (or *PEs*), each of which is physically small enough to fit inside a modern VLSI chip.

The inherent performance and cost advantages of placing systems within chips impel both reductions in transistor dimensions and increases in the physical sizes of chips. The size of a chip, as measured in the number of transistors it contains, increases over time.

Parallel computers containing large numbers of coordinated PEs are the fastest computers for some problems. The *data-parallel problems* [41] constitute one class of computational problem for which the solution speed is roughly proportional to the number of PEs. For a scalable data-parallel problem, the fastest parallel computer is that containing the greatest number of PEs, all else being equal.

There is an interaction between VLSI and parallelism that warrants caution on the part of the computer architect: VLSI implementation technique imposes an upper limit on the number of PEs that may be packaged together in a chip. The compromise with respect to this technical reality is to decompose the target system into modules of manageable size, each containing as many PEs as will fit on one chip when accompanied by whatever circuitry may be needed to facilitate the decomposition. Coordinating large numbers of PEs requires inter-chip communication, which typically occurs at a lower rate than communication within chips. The speed advantages of parallel computers are tempered by the extent to which the activity of a PE can no longer be confined to a single chip. Coordination among PEs, access by PEs to non-integral storage or functional components, and movement of input and output data sets to and from the PEs, all incur significant speed penalties. These speed penalties tend to lessen as VLSI implementation technique improves, because an increasing fraction of the total activity occurs at the relatively high rates achievable within the confines of a chip.

This dissertation addresses managing one interaction between VLSI and parallelism in the design of Single Instruction-stream/Multiple Data-stream (or *SIMD*) computers. A SIMD computer is a parallel computer whose PEs are as simple as possible, so as to occupy as little chip area as possible. Because SIMD computer architecture packs a maximum number of PEs into the available total chip area, one might expect a SIMD computer to be among the fastest possible for data-parallel problems. Unfortunately, generic SIMD computer architecture introduces a throughput limitation by requiring that a new instruction be broadcast to the PEs on each clock cycle. Instruction broadcast to large numbers of chips occurs at a lower rate than the PEs' maximum rate of intra-chip operation.

Of the many cost factors juggled by computer designers, including monetary cost, power consumption, size, and cooling, total chip area is not often the most important. However, there are application contexts wherein making the best use of chip area is very important. Making good use of chip area is paramount where the greatest throughput is sought for a given implementation budget with respect to total chip area. Examples of this type of application include physical simulations, scientific models, and commercial data analysis. Alternatively, it is important to make the best use of chip area where the physical size of the computer is limited. Examples of computer systems for which size is critical include portable computers, commodity signal processors, and embedded systems.

Because of its chip-area parsimony, generic SIMD computer architecture would be appealing, if not for its inherent throughput limitation. The importance of making good use of chip area leads one to wonder: Is it possible to re-formulate SIMD computer architecture slightly, without relinquishing all of the chip-area advantage, to allow the PEs to operate at their highest rate?

The stakes are high enough for the answer to this question to be interesting, because while a PE in a SIMD computer occupies between 20% and 50% of the chip area of a microprocessor with the same calculation components, PEs in existing SIMD computers operate at clock rates 8 to 12 times below the maximum achievable using their VLSI implementation techniques. While a factor of up to 5 decrease in total chip area may be compelling, surrendering a factor of 8 in operation rate to achieve it is a poor exchange where speed is the ultimate objective.

SIMD instruction cache (or *I-cache*), introduced in this dissertation, is an explicitly managed instruction buffer added to the PE chips of a SIMD computer. Globally broadcast instruction sequences that are to be repeated are stored in I-cache for subsequent re-broadcast at the high rate attainable within the PE chip. Whereas SIMD computer architecture simplifies PEs by eliminating program control that is redundant for some problems, adding I-cache re-introduces some redundant program control into the PE chip. How much chip area does the re-introduced program control occupy? Further, by how much does I-cache increase throughput? Clearly, the answers to these questions depend on the characteristics of computational problems, of PE architecture, of VLSI implementation technique, and of the target computer's decomposition into chips. The analysis of I-cache reveals how these characteristics interact. Detailed evaluations of the simplest I-cache variants over a set of diverse sample problems show that throughput increases are two to three orders of magnitude larger than the concomitant chip-area cost of I-cache in a modern PE chip.

1.1 The Thesis

The ratio of throughput to total chip area is a useful computation metric in a world of finite resources: For a given total chip area used in a computer, a higher throughput-to-area ratio implies higher throughput. Equivalently, for a given required throughput, a higher throughput-to-area ratio implies lower total chip area. For scalable data-parallel problems, high throughput-to-area ratios are exploitable either as high throughput or as low chip area, depending upon application requirements. Throughput-to-area ratio is an especially important metric in designing computers for problems demanding the highest possible throughput given a limited implementation budget with respect to total chip area.

This section introduces my thesis by providing the background for understanding the throughput-to-area ratio consequences of I-cache. Figure 1.1 illustrates a sequence of improvements to computer architecture leading from generic uniprocessors to maximum-throughput programmable VLSI-based multiprocessors. Each step in the sequence increases the throughput or decreases the cost of the computer. Each step is applicable only to a subset of the computations at that step, so the domain of problems narrows as successive steps are taken. This particular sequence describes one path towards the goal of fast, inexpensive computations, rather than all such paths.

The last two steps in the sequence shown in Figure 1.1 are new, resulting in I-cached SIMD

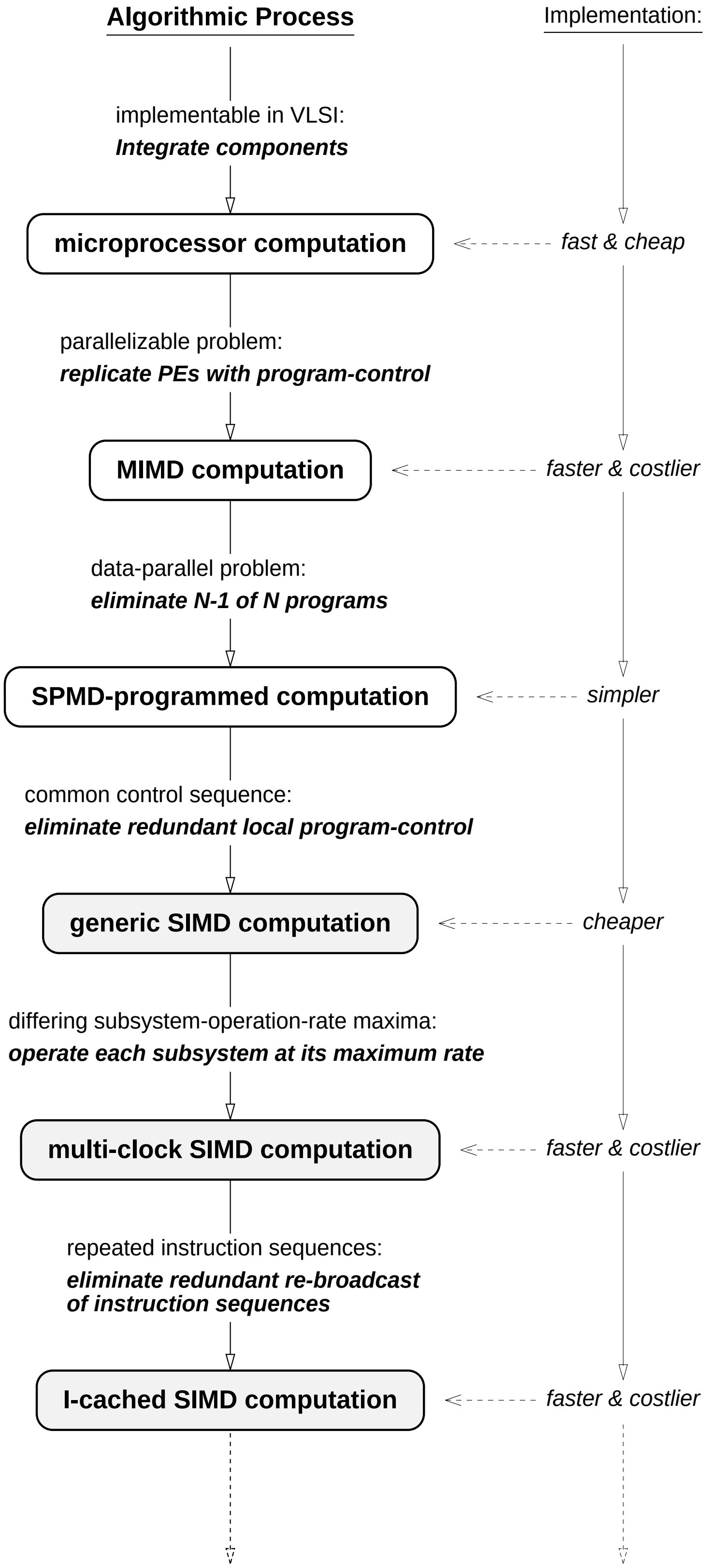


Figure 1.1: Successive Computer Architecture Improvements

computers. My thesis is that significant speedups are exhibited even for simple I-cache variants over a broad range of data-parallel computations.

The first step in Figure 1.1 characterizes the advent of the microprocessor in the 1970s [76], whereupon it became possible to place substantial parts of all of the main subsystems of a computer on a single chip, thus enabling appreciably fast computation at a relatively low cost.

The second step in Figure 1.1 reflects the possibility of multiprocessor computers being faster, though of course more expensive, than their single-processor counterparts. This step can be taken for computations that admit parallel solutions. It is interesting historically to note that the second step shown in Figure 1.1 was in fact applied first, in the construction of non-VLSI multiprocessors including Solomon [74] and ILLIAC IV [3] in the 1960s. This variance with historical progress underscores the fact that the sequence represented in Figure 1.1 is not prescriptive of the development of inexpensive fast computers, but rather merely suggestive of one path leading in that direction.

The most general form of parallel computer is a Multiple Instruction-stream/Multiple Data-stream (or *MIMD*) computer, wherein each PE comprises counterparts of the uniprocessor's calculation and program-control components, adapted for inter-PE communication. It is increasingly common for MIMD computers to contain microprocessors and memory chips in their replicated building blocks. Because they are used in a very large number of applications, microprocessors and memory chips are manufactured in high volumes. High-volume manufacture typically leads to low monetary cost. Low monetary PE cost underlies the popularity of this style of MIMD computer design.

The common parallel-programming practice of replicating a single program in all PEs of a MIMD computer is called Single Program/Multiple Data-stream (or *SPMD*) programming. Although SPMD programming is a method-of-use specialization of a MIMD computer rather than an improvement to the computer itself, the simplicity of SPMD in some cases reduces the programming costs associated with multiprocessor computation. This step can be taken only for computations solving data-parallel problems.

The next step in the sequence shown in Figure 1.1 can be followed for SPMD computations wherein all PEs' executions of the single program happen to proceed in a common sequence. In these cases, the replication of program storage and program sequencing in every PE is redundant. Eliminating the redundant program control from the PE, so that instead a system controller provides a single sequence of instructions for all PEs via a global instruction broadcast network, significantly reduces the chip area occupied by a PE. This control sharing characterizes SIMD computer architecture.

In general, a SIMD computer is less convenient to program than a MIMD computer, because of the requirement that all PEs receive a common sequence of instructions. This inconvenience restricts the class of problems for which SIMD computers are appropriate. Primarily, these problems are the scalable data-parallel problems. The low production volume of SIMD PE chips makes the monetary fabrication cost per chip characteristically high. A consequence of high PE chip cost is that the applications for which SIMD computers are practical are further restricted to those in which making the best use of chip area is paramount.

The shading of the "generic SIMD computer" box in Figure 1.1 indicates that it is the starting point for the modifications suggested in this dissertation.

Inter-chip wire delays tend to be large compared to intra-chip wire delays. The broadcast of a new instruction every clock cycle from a central repository, as occurs in a generic SIMD computer, means that every instruction's execution involves inter-chip signaling. Therefore, instruction execution proceeds at a relatively low inter-chip signaling rate. Generic SIMD computers suffer from an inherent instruction delivery-rate limitation.

The maximum operation rates of a SIMD computer's subsystems are determined principally by VLSI implementation technique and by the electrical propagation characteristics of inter-chip wires. The next step in Figure 1.1 is to adapt a generic SIMD computer so that clocks are available to regulate the various subsystems, including the PEs, each at its maximum rate. Such a computer is a *multi-clock SIMD computer*. A multi-clock SIMD computer might incorporate a clock-rate multiplier in the PE chip resembling the high-rate clock generators that are increasingly commonly used in microprocessors [5, 88]. A multi-clock generator provides a PE clock within the PE chip.

Note that in a multi-clock SIMD computer, even when the rate of the PE clock exceeds that of the clock regulating global instruction broadcast, PEs still receive instructions at the relatively low instruction broadcast rate. I-cache is one means to overcome the instruction delivery-rate limitation. I-cache is an explicitly managed instruction buffer inside the PE chip. In an I-cached SIMD computer, repeated instruction sequences are stored within the PE chip for subsequent retrieval at the relatively high PE clock rate.

While it is reasonable to expect an I-cached SIMD computer to be at least somewhat faster than a generic SIMD computer, it is not clear *a priori* just how costly are multi-clocking and I-cache. Indeed, some computer scientists might reasonably object to the improvement claimed for the final two steps in Figure 1.1: How can multiple clocks and I-cache speed up SIMD computation, which we suspect to be inherently limited in the rates of inter-PE communication or local external memory access? Such objections may be lodged, and indeed need to be addressed, with respect to every subsystem of a SIMD computer in relation to the global instruction broadcast subsystem, which is the focus of the thesis.

This dissertation provides evidence that adding I-cache is very attractive: while occupying less than 1% of the area of a modern PE chip, I-cache significantly increases the throughput of a diversity of data-parallel computations. Detailed simulations of simple I-cache variants reveal that the I-cache tradeoff has many facets, including the following:

- I-cache speedups depend strongly on the relative clock rates of the computer's subsystems. Where the disparity between PE clock rate and global instruction broadcast rate is a factor of 8, these simple I-cache variants yield speedups ranging between factors of 1.3 and 7.9 for a diverse set of sample programs. For sample programs with simple loop structure, simple I-cache variants do nearly as well as possible. Some of the sample programs have more complicated loop structures, for example wherein repeated instruction sequences alternate execution. These programs with complex loop structures demand more complex I-cache variants.

- Surprisingly, the interaction between I-cache speedup and PE chip pin time-sharing for multi-chip subsystems depends on the relative clock rates of the subsystems. When a multi-chip subsystem's clock rate is as low as the instruction broadcast rate, I-cache speedup decreases with the degree of PE chip pin time-sharing. However, if the subsystem's clock rate is higher than the instruction broadcast rate, I-cache speedup in fact increases with the degree of PE chip pin time-sharing. I-cache acts in some cases to reduce communication bottlenecks that typically occur at chip boundaries.

- Appropriate management of I-cache has important consequences for speedup. For some sample problems, straightforward static management of I-cache yields near-maximum speedups. Other sample problems, for example those wherein loop iterations are data-dependent or loop-index-dependent, demand dynamic cache management mechanisms.

- A variety of strategies for providing chip area for I-cache in a PE chip of fixed size are presented. Empirical evaluations of these strategies indicate that while no one strategy works best in all cases, it is very likely that small I-cache can be accommodated with slight impact on the operational structure of a SIMD computation.

1.2 The Structure of the Dissertation

The reader may currently be using (or thinking about using) data-parallel computation to solve a particular problem. Such a reader is likely interested in learning how much faster an I-cached SIMD computer would be than its generic counterpart for that problem, and at what cost. Unfortunately, there is a huge number of cases, and the dissertation does not provide a closed-form analytic tradeoff equation. The dissertation does characterize a heretofore unexplored region of the design space for the fastest possible computers for scalable data-parallel problems. This characterization takes the form of analysis and evaluation, grounded in detailed examples, that elucidate the issues and tradeoffs relating to I-cache.

Chapter 2 presents the essential facets of the I-cache idea. The instruction delivery-rate limitation of SIMD computers is developed in a qualitative manner, and I-cache is proposed as a means to surmount that limitation. Although the dissertation focusses specifically on I-cache added to SIMD computers' PE chips, the central problem of distributing instructions through a relatively slow channel to many PEs arises also in loading programs into the PEs of MIMD computers. The use of the term "I-cache" to describe an explicitly managed PE chip instruction buffer may surprise those who are familiar with the direct-mapped instruction caches ordinarily used in microprocessors. A clear distinction is drawn between SIMD instruction cache and ordinary instruction cache. A space of possible I-cache designs is then painted in broad strokes, to provide a framework in which to consider the mechanisms and functions. To illustrate the essential functions of I-cache, an example program for a SIMD computation is presented and its I-cache speedup is calculated and discussed. Chapter 2 reveals the complexity of the interactions between I-cache and the various characteristics of computations, so Chapter 2 motivates the detailed analysis to follow.

Chapter 3 provides the definition of a SIMD computer that is the starting point in the analysis of I-cache. Chapter 3 introduces the SIMD computer's components and their general functions. A small set of abstractions are incorporated in the model of a generic SIMD computer that allow the model to encompass a broad range of existing and foreseeable SIMD computers. A detailed example of a SIMD computation grounds the discussion. I-cache is added to the local controller of a PE chip. To provide a basis for assessing the chip-area impact of I-cache, a physical model for the PE chip is presented. A practical estimate for the disparity between PE clock rate and global instruction broadcast rate is developed which shows that it is very likely for the disparity to exceed a factor of 2. Chapter 3 concludes with a consideration of the options for overcoming the throughput limitation that arises from that disparity in generic SIMD computers.

Chapter 4 describes the I-cache design elements and introduces a family of related single-port I-cache variants. The chip area occupied by members of this family is estimated. These I-cache variants occupy as little as 1% of the chip area originally occupied by PEs in a PE chip made with present-day VLSI implementation technique. Interactions of I-cache speedup with program properties are considered, as are interactions with electrical characteristics of the computer. These considerations provide clues as to what effects to look for in the empirical evaluations of I-cache. Finally, the I-cache management problem is discussed in detail, illustrated with examples for the family of I-cache variants.

Chapter 5 describes the empirical method used to measure throughput of SIMD computations. Measurements are taken from register-transfer-level simulations of a basis computer that is parameterized to represent a broad range of generic, multi-clock, and I-cached SIMD computers. Chapter 5 also describes the set of SIMD computer variants on which speedups are measured and the set of sample problems for which evaluations are performed. Each simulated computation comprises a problem running on a SIMD computer variant assumed to contain an inter-PE communication network topology appropriate for the problem. After giving an overview of I-cache speedups for the various problems, Chapter 5 presents summaries of the measured speedups. The measured speedups are compared against estimates for the greatest possible single-port I-cache speedups, and the chapter concludes with measurements that reveal the sensitivities of I-cache speedups to the intensiveness with which programs utilize the SIMD computer's multi-chip subsystems.

The I-cache evaluations in Chapter 5 are performed under the assumption that any additional chip area needed for I-cache is provided without changing the operational structure of the subject computation. Such would be the case, for example, if the physical size of the PE chip were increased to accommodate I-cache. Chapter 6 considers alternative strategies for providing the chip area occupied by I-cache, under the realistic assumption that the PE chip's physical size is limited. Chapter 6 presents I-cache speedup measurements wherein I-cache chip area is provided by limiting cache size, reducing PE register count, reducing PE chip PE count, or reducing PE function unit complexity. The results show that while none of these strategies for providing chip area for I-cache is universally desirable, it is important to make the I-cache large enough to contain entire loop bodies.

Appendices are included that describe in detail the basis computer, the assembly language used to describe the computations for the sample problems, an example of developing a sample problem's solution and then adding I-cache to the computation, the set of sample programs, and the complete set of measured I-cache speedups.

The dissertation shows that for diverse problems, even simple I-cache variants yield significant throughput-to-area ratio increases over generic SIMD computers. The factors by which I-cache increases throughput-to-area ratio depend on the properties of programs, of VLSI implementation technique, of PE architecture, and of the architecture of the replicated chip containing the PEs. Significant speedups are measured for each of a diverse collection of problems, under reasonable assumptions regarding the electrical characteristics of the computer. That significant speedups arise in each case shows that even some problems whose computations are ordinarily assumed to be communication-bound benefit from I-cache.

The simulation results show that I-cache makes good use of chip area in the PE chip. Therefore, scalable data-parallel applications for which it is paramount to make good use of chip area demand I-cache. The observed I-cache speedups, along with the modest estimated chip-area cost of I-cache, suggest that an I-cached SIMD computer would exhibit the highest throughput of any programmable multiprocessor for scalable data-parallel problems.

Chapter 2

The SIMD Instruction Cache Idea

In SIMD computers, maximum PE instruction execution rate is higher than global instruction broadcast rate. The ratio between these rates is denoted ρ_b. The alternatives available for achieving maximum throughput when $\rho_b > 1$ include SIMD instruction cache, or *I-cache.* I-cache in this context is new. Its function, and therefore its design, are subtly different from those of the more familiar caches that are used in microprocessors.

By presenting some of the main hardware and control choices for I-cache, this chapter sketches the I-cache design space. This chapter concludes with an example of how I-cache is used and how it impacts the throughput of a SIMD computation.

2.1 Instruction Delivery in SIMD Computers

The PE of a SIMD computer contains only the minimum circuitry needed to operate its data storage and function units under control of the single instruction stream which it shares with the other PEs. Conceptually, a SIMD PE may be thought of as a MIMD PE whose program-control components that store, fetch, and decode instructions have been removed.

On one hand, the SIMD PE's simplicity means that it occupies minimum chip area. Minimum chip area per PE allows the greatest number of PEs to be packed into a given total chip area. The densest packing of PEs should translate into maximum performance per hardware cost for scalable data-parallel problems.

On the other hand, the SIMD PE's simplicity introduces a new limitation in the rate at which instructions are executed. The simplified PE does not perform its own program control. Instead, the system controller provides a new machine code instruction on each cycle of the system clock. The limitation arises when the rate at which instructions are provided from afar is lower than the maximum rate of PE calculation.

The SIMD PE's simplicity allows many PEs to be packaged within a PE chip of modest size using current VLSI implementation technique. The system controller in a SIMD computer performs the program-control function once for all PEs, providing a fresh instruction to the PEs on each cycle of the system clock. This arrangement is sketched in Figure 2.1.

PE calculation takes place entirely within the confines of the PE chip. By contrast, the electrical pathway carrying instructions from the system controller must cross between chips at least once before it arrives at any PE.

The set of wires through which instructions are delivered to the PE chips from the system controller is the *global instruction broadcast network.* Let R denote the highest rate at which the PEs within the PE chip can execute instructions. Let B denote the rate at which instructions are delivered to the PE chips through the global instruction broadcast network. Then $\rho_b = \frac{R}{B}$ is the ratio

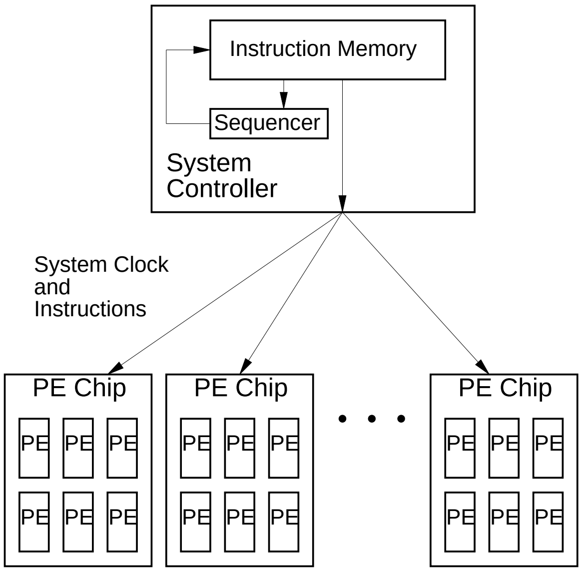


Figure 2.1: The system controller provides a system clock and broadcasts a new instruction on each clock cycle.

of these two rates. That is, ρ_b denotes the factor by which the highest rate of PE operation exceeds the rate of global instruction broadcast.

In existing SIMD computers, ρ_b=1 because R=B. This parity between R and B arises in existing SIMD computers not because global instruction broadcast networks are carefully engineered (so that B is high) but rather because PEs are artificially slow (making R low). As a typical example, consider CM-2, a well-known SIMD computer whose PE contains a bit-serial function unit [22]. The system clock rate in CM-2 is about 8MHz. For comparison, chips of complexity greater than that of the CM-2 PE have been fabricated using similar VLSI process technology that run at clock rates as high as 100MHz [86]. This disparity is a factor of 12. Other SIMD computers whose PE chips operate surprisingly slowly include VASTOR (2MHz) [87], CLIP7A (5MHz) [32], AIS-5000 (7MHz) [68], AAP2 (10MHz) [53], CAAPP (10MHz) [73], MP-1 (14MHz) [62], and Blitzen (20MHz) [36].

One ostensible reason for making R artificially low might be to reduce the cost of a PE chip. For example, it was possible to use relatively inexpensive external memory chips with the slow PE chips in CM-2 at no performance penalty. Unfortunately, an economic rationale for making R artificially low is inconsistent with the design objectives of SIMD computer architecture. Making best use of chip area is paramount for a relatively small subset of all computational problems, so PE chips for SIMD computers are produced in relatively small quantities. Economies of scale make SIMD PE chips relatively costly per part. Given that the monetary cost of a PE chip is high, saving cost by under-designing the PE chip or by using inexpensive external memories is not a good reason for R to be as low as B. If monetary cost is the prevalent concern, then SIMD computer architecture is not likely to be an attractive choice. However, where maximum throughput is desired for the available total chip area, or where minimum total chip area is desired in achieving a given throughput target, SIMD computer architecture is a compelling alternative, so long as R is not unnecessarily low.

Making the PE chip as fast as possible, such that R is maximum for the PE architecture and for the VLSI implementation technique, presents design challenges for system integration. While ρ_b need not be as large as the factors around 10 suggested by the operation rates and VLSI implementation techniques of existing SIMD computers, it may be the case that even using high-speed interconnect techniques to globally broadcast instructions, ρ_b is yet greater than 1, perhaps as high as 2 or 3. Fast instruction broadcast requires careful engineering of the global instruction broadcast network, and it reduces the flexibility of the system with respect to scaling and geometric re-arrangement of the PEs. Therefore, higher values of ρ_b are associated with SIMD computers designed for scalability at lower system re-integration costs. A further potential drawback of fast broadcast of instructions is that it demands the system controller, which performs a potentially complicated control function, to operate at high speed. Finally, the pin-limited nature of PE chips makes time-sharing of instruction receiver pins attractive, if time-sharing can take place without compromising throughput. Compromise in respect of any of these factors leads to large values of ρ_b.

2.2 Overcoming Slow Instruction Delivery

If the PEs are to perform calculations at the highest possible rates, it may be possible to bring B into parity with R through careful electrical engineering of the global instruction broadcast network. However, if it is impractical to do this, then ρ_b is larger than 1. This possibility leads the computer designer to wonder: What are the architectural alternatives to overcoming the instruction execution rate limitation that arises when ρ_b is significantly greater than 1? Is it possible to take advantage of a value of R that is greater than B, or must R be made artificially low?

One option is to make the PE chips microprogrammed. With microprogramming, globally broadcast instructions encode sequences of single-cycle PE operations. A microcontroller inside the PE chip sequences each globally broadcast instruction's microprogram for the PEs within the chip.

The global broadcast of microcoded instructions has been applied in limited ways in existing SIMD computers. For example, multiply and divide operations in SLAP are controlled using a pair of globally broadcast instructions, one to initiate and one to terminate an arithmetic sequence [27](p.373). The PE instructions carrying out the arithmetic sequence are provided from within the PE chip. This design choice was made for SLAP to free up the PE chip's instruction pins so that inter-chip communication operations could be controlled concurrently with multiplication or division. The PE chip in SLAP is clocked at the rate of global instruction broadcast.

CM-2 incorporates microprogramming in a limited way. Driven by a high-level language program, the CM-2 front end generates complex instructions that are decomposed into single-clock-cycle operations for the PEs by a microcoded sequencer. This design choice was made for CM-2 to simplify the computer's high-level programming interface. The CM-2 sequencer is not integrated in the PE chip, so it could not help alleviate the consequences of a large value of ρ_b.

An alternative to microprogramming the PEs is SIMD instruction cache. SIMD instruction cache, or *I-cache*, exploits temporal locality in the broadcast instruction stream. The I-cache is a buffer within the PE chip that stores instruction sequences that are identified as being repeated. Repetitions of a stored sequence are subsequently delivered to the PEs at the highest rate of PE operation. Each PE does not necessarily need its own I-cache; one I-cache is able to provide instructions to the collection of PEs located in a PE chip.

I-cache and microcoding both involve beefing up the program control provided within the PE chip. Comparing these two alternatives, it is apparent that I-cache suffers the relative disadvantage that instruction sequences must first be stored at the slow broadcast rate B before they can be retrieved at the high chip rate R. An inherent advantage of I-cache where chip area is limited is that only the instruction sequences needed for a given computation are stored in an I-cache. By contrast, a set of microprograms committed to ROM in the PE chip may be large and, at best, difficult to modify.

2.3 A New Use of the Term “Instruction Cache”

A SIMD instruction cache differs in a number of important ways from the direct-mapped instruction caches typically used in microprocessors. This section highlights the main differences.

The principal difference between conventional instruction cache and SIMD I-cache is that SIMD I-cache requires explicit control. The presence of ordinary instruction cache is not apparent in programs and involves no change to the instruction set architecture of the computer. A SIMD I-cache, by contrast, is explicitly managed in programs through the use of a small number of cache-control instructions added to the set of globally broadcastable instructions. A SIMD I-cache operates under the programmer's control in a pre-determined manner, whereas an ordinary instruction cache exploits temporal locality in an instruction stream opportunistically without the programmer's intervention.

Instructions are read from an ordinary cache on an individual basis to satisfy cache hits, and instructions are stored in an ordinary cache as fixed-size blocks (or lines) to exploit expected spatial locality in an instruction stream. A SIMD I-cache is not read for individual instructions, nor are instructions written into it in fixed-size lines. Rather, a SIMD I-cache stores instruction *sequences* of varying lengths. The length of a SIMD I-cache block matches the length of the corresponding repeated sequence of instructions. For example, the body of an inner loop is managed as a single block in a SIMD I-cache.

SIMD I-cache is managed differently from ordinary instruction cache. Whereas with an ordinary cache comes hardware to determine dynamically whether each successive instruction reference may be satisfied from cache, the SIMD I-caches discussed herein use no such hardware. Rather, these SIMD I-caches are managed *statically*, by the programmer, perhaps in conjunction with a compiler. This characteristic is not inherent, but it is appropriate for demonstrating the concept of SIMD I-cache with programs whose flow-graph structures are easily statically analyzed. From the point of view of how the cache memory is managed, static management of SIMD I-cache resembles the well-known compile-time problem of register allocation.

A microprocessor contains a program-control component that, among other things, generates instruction memory references. Those instruction memory references are sped up through the use of ordinary instruction cache. By contrast, a SIMD PE chip does *not* generate instruction memory references, because it instead passively receives a sequence of globally broadcast instructions. Therefore, instructions are placed explicitly in a SIMD I-cache, under the direction of globally broadcast instructions. Subsequently, the PE chip's local controller is directed explicitly to retrieve a stored sequence of instructions from I-cache for execution by the PEs.

The difference between SIMD I-cache and ordinary instruction cache can be summed up by observing that while some accesses to an ordinary instruction cache miss, all accesses to a SIMD I-cache hit. If a needed instruction is not present in I-cache, then it is globally broadcast as it would have been in a SIMD computer without I-cache. I-cache does not change the lock-step nature of instruction execution among the PEs of a SIMD computer. All PEs still receive a common sequence of instructions, but a single I-cache in the PE chip makes it possible to provide repeat instructions at the highest rate to the collection of PEs within the chip.

It is now evident that there are many differences between a SIMD I-cache and the architectural component ordinarily connoted by the term "cache". Despite these differences, the term is used here to refer to PE chip instruction buffers because in its most general sense, "cache" means a relatively small, fast repository used to speed up computation. The reader need only bear in mind that instead of circumventing redundant slow accesses to some larger repository of instructions, SIMD I-cache is used to eliminate redundant instruction broadcasts through a slow network.

2.4 I-Cache Design Parameters

Just what does a SIMD instruction cache look like? This section outlines the I-cache design space by identifying four major physical design choices.

1. Number of cache blocks: An I-cache may contain a single block at a time, or it may contain multiple blocks.

   A single-block I-cache is simplest with respect to delimiting blocks in cache memory and also with respect to managing the available memory, because there is a unique starting address for any cache block. For a multi-block I-cache, the globally broadcast instruction activating a cache block must specify the starting address of the block in cache.

   Containing more than one block at a time is advantageous for computations wherein a number of repeated instruction sequences alternate over the course of the program's execution. With a single-block I-cache, the alternating sequences must be re-stored in I-cache prior to each use, because the alternating sequences displace one another. Such re-storing of instruction sequences in cache is a form of thrashing that reduces I-cache effectiveness.

2. Number of iterations per cache block activation: An I-cache controller may sequence single passes through a cache block, or it may be capable of sequencing multiple passes in response to a global broadcast instruction activating the cache block.

   A single-pass I-cache is simplest with respect to managing iterations. A multi-pass I-cache variant requires an equivalent of an iteration counter. A multi-pass I-cache variant might, for example, be one in which loop-control instructions may be placed in cache.

   A multi-pass I-cache is advantageous for instruction sequences whose duration is not an integer multiple of ρ_b. After each pass through such a sequence, a single-pass I-cache variant awaits the next globally broadcast instruction to arrive. A multi-pass I-cache variant, by contrast, does not wait after completing one iteration; rather, it begins sequencing the next iteration immediately.

3. Number of cache memory ports: Cache memory may have a single port, or it may have more ports.

   Single-port memories are electrically simpler to design than multi-port memories, and their cells lay out more compactly than do multi-port memory cells.

   Using a two-port cache memory in conjunction with a multi-block I-cache variant, it is possible to pre-store one cache block while another cache block is active. Using a second cache memory port in this way is a form of prefetching that minimizes the time spent idly by the PEs waiting for the next block to be placed in cache.

4. Nesting of cache blocks: A multi-block I-cache variant may or may not allow cache blocks to activate one another.

   An I-cache without nesting is simpler, because no cache block execution stack is needed.

   Allowing cache block activation instructions to be placed in cache memory means that arbitrarily complex loop structures may be cached entirely. Once an entire program is stored in cache, the PE chips need not wait for globally broadcast instructions over the course of the computation.

   A PE chip containing an I-cache variant of this complexity may be viewed as a SIMD computer in its own right, because the program stored in cache which controls the collection of PEs in the PE chip may execute independently of the other PE chips in the computer. SIMD computers with multiple program-control units are called Multi-SIMD (or *MSIMD*) computers, as exemplified in GPA [10].

This dissertation evaluates in detail two of the simplest I-cache variants, F_0 and F_2. F_0 is the simplest I-cache variant, containing one block at a time that is executed in single passes. F_0 is a "one-block, one-shot" I-cache variant. Of course, F_0 has only one port and the question of cache block nesting is moot. F_2 is almost identical to F_0, with the addition of the ability to execute multiple iterations of a cache block from a single globally broadcast activation instruction. F_2 is a "one-block, multi-shot" I-cache variant.

2.5 Management of I-Cache

All components of a SIMD computer are centrally controlled (by the system controller), and I-cache is no different. Although each PE chip has its own I-cache, the states of all of them are identical. The I-cache replicated in each PE chip is logically redundant, but it serves to remove redundantly repeated instructions from the global broadcast instruction stream. With I-cache, those repeated instructions may instead be delivered to the PEs within the PE chip from the fast on-chip repository also within the PE chip.

I-cache is managed by the system controller using globally broadcast cache-control instructions. The local controller within the PE chip is directed when to begin storing instructions in cache, when to stop storing instructions there. Subsequently, the system controller instructs the local controller to begin executing a cache block by providing the parameters needed to activate the cache block.

Management of SIMD instruction cache is performed either *statically* by a programmer or compiler, or *dynamically* by a cache management algorithm running on the system controller. Static cache management occurs prior to execution of a program. Static management is accomplished by modifying the program controlling the computation that is loaded into the system controller before the outset of the computation. Static management is sometimes useful also for ordinary instruction caches; a compiler that statically re-arranges instructions in memory to increase the efficiency of uniprocessor caches is reported in [57]. The static management of I-cache is reminiscent of the *overlaying* used in programs to manage the small main memories of early computer systems including the DEC PDP-11 and the IBM 360-370 [11](p.15). Dynamic I-cache management amounts to a case of the well-known memory management problem that arises, for example, in implementing virtual memory [72](Sec.5.2).

Whether I-cache is managed statically or dynamically, and however complex the I-cache design variant, there is a single set of cache-management sub-problems that are solved in all cases. These sub-problems are:

1. identifying the cachable instruction sequences,
2. determining which sequences are stored in cache,
3. determining where in cache to store cache blocks,
4. scheduling cache blocks appropriately,
5. directing the storing of the scheduled cache blocks in cache prior to their use,
6. and activating stored cache blocks at the appropriate points in the computation.

These sub-problems are considered in more detail later (in Section 4.5). Although the I-caches evaluated herein are managed statically, this has been done for simplicity in the system controller. The evaluations are performed for sample problems for which the best use of I-cache is straightforward. For programs whose flow graphs are difficult to analyze well statically, the system controller should contain a unit that performs dynamic I-cache management. Such an I-cache management unit would maintain a model of which instructions are stored where in I-cache. Before globally broadcasting a cachable instruction sequence, the I-cache management unit would be consulted to determine if the sequence were already in cache. If so, the sequence would be executed from cache.

If not, the decision would be made at that point whether to place the sequence in cache, and where. The examples of static I-cache management given in this dissertation are indicative of how those decisions are made.

2.6 An Example of I-Cache Use

This section presents a simple example of how I-cache is used. For an industrial-strength example for an assembly language, the reader should consult Appendix C.

Consider the skeleton for program **simplest**, consisting only of a simple loop:

```
program simplest;
   for j = 1 to J do
      A
   end;
end simplest;
```

The symbol **A** in program **simplest** denotes the sequence of instructions that is the loop body. Assuming that **A** is a cachable instruction sequence, the problems of determining which sequence to store in cache, when to store it, and when to activate it have obvious solutions.

Recall that an F_0 I-cache is the simplest I-cache variant, capable of storing only one cache block at a time and executing only single passes through the stored block. The following skeleton for program **simplest_cache** illustrates how program **simplest** is modified to use an an F_0 I-cache:

```
program simplest_cache;
   store sequence A in cache
   for j = 1 to J do
      activate cached sequence A
   end;
end simplest_cache;
```

Ideally, each pass through cached sequence **A** in **simplest_cache** is ρ_b times faster than each corresponding iteration of the inner loop in **simplest**. However, storing **A** in cache is an extra pass through that sequence of instructions in **simplest_cache** that has no counterpart in **simplest**.

2.7 I-Cache Speedup

In general, the benefit of using I-cache for a program can be measured directly by comparing the execution time of the program on a baseline generic SIMD computer against that of a version modified to run on a SIMD computer with I-cache. The ratio of these two times gives the *speedup* due to I-cache for the subject program.

A convenient unit to measure the time taken to run a SIMD computation is numbers of cycles of the system clock. For example, the time to run program **simplest** above is given as

$$\text{time for } \textbf{simplest} = J * |\mathbf{A}| \text{ cycles}$$

where $|\mathbf{A}|$ denotes the number of system clock cycles taken for instruction sequence **A**.

Assuming that instruction sequence **A** is cachable and that it runs faster from cache by the factor ρ_b, then the time for the modified program to run with I-cache is given as

$$\text{time for } \textbf{simplest\_cache} = |\mathbf{A}| + J * \frac{|\mathbf{A}|}{\rho_b} \text{ cycles}$$

The I-cache speedup is given as the ratio of these two execution times:

$$\begin{aligned}\text{speedup for } \mathtt{simplest} &= \frac{J * |\mathbf{A}|}{|\mathbf{A}| + J * \frac{|\mathbf{A}|}{\rho_{\mathrm{b}}}} \\ &= \frac{J}{1 + \frac{J}{\rho_{\mathrm{b}}}} \qquad (2.1) \\ &= \frac{J * \rho_{\mathrm{b}}}{J + \rho_{\mathrm{b}}} \qquad (2.2)\end{aligned}$$

Equation 2.2 suggests that the speedup for program **`simplest`** approaches ρ_{b} as J approaches ∞. At the other extreme, if sequence **A** is executed just once (so that J=1), then the speedup is less than 1. While I-cache makes it possible to speed up a computation when ρ_{b} is large, if used inappropriately then I-cache can actually slow down a computation. This possibility arises because the time needed to store an instruction sequence in a simple I-cache is not negligible.

Equation 2.1 points up an analogy between I-cache speedup and speedup due to parallelism. Recall that a P-PE multiprocessor achieves up to P times the throughput of its uniprocessor counterpart, although in practice multiprocessor speedup is usually considerably less than that limit [78]. In general, C of the N instructions executed in the uniprocessor computation are inherently sequential, such that they cannot be run faster through parallel execution. The multiprocessor speedup limit is given in Equation 2.3:

$$\text{speedup due to parallelism} \leq \frac{N}{C + \frac{N-C}{P}} \qquad (2.3)$$

Amdahl's law is the observation that multiprocessor speedup cannot exceed $\frac{N}{C}$, irrespective of the number of PEs in the multiprocessor. For example, if 90% of the instructions executed on a uniprocessor execute P times faster on a P-PE multiprocessor while the rest of the instructions execute no faster on the multiprocessor (such that $\frac{N}{C}$=9), then the speedup from parallelism cannot exceed 10. In this example, for P=1000, the actual speedup is just over 9.9.

The analogy with Amdahl's law for I-cache speedup is that speedup cannot exceed J, irrespective of the value of ρ_{b}. In an I-cached SIMD computation, some instructions must be globally broadcast, for example to store them in I-cache as in program **`simplest_cache`** above. Just as the actual speedup from multiprocessing is usually less than the number of PEs P, the actual I-cache speedup is usually less than the limit of ρ_{b}, even for ideally suitable problems. This limiting effect of J is evident in the small speedups for **`simplest`** for low values of J in Figure 2.2.

This analogy with Amdahl's law shows the dangers in assuming naively that the benefit from I-cache is proportional to ρ_{b}. In fact, the interactions among I-cache design, system design, and program properties are complex. The difficulty in ascertaining *a priori* the impact of I-cache on the throughput and chip area of a given SIMD computation necessitates the empirical evaluation of I-cache variants that is the focus of this dissertation.

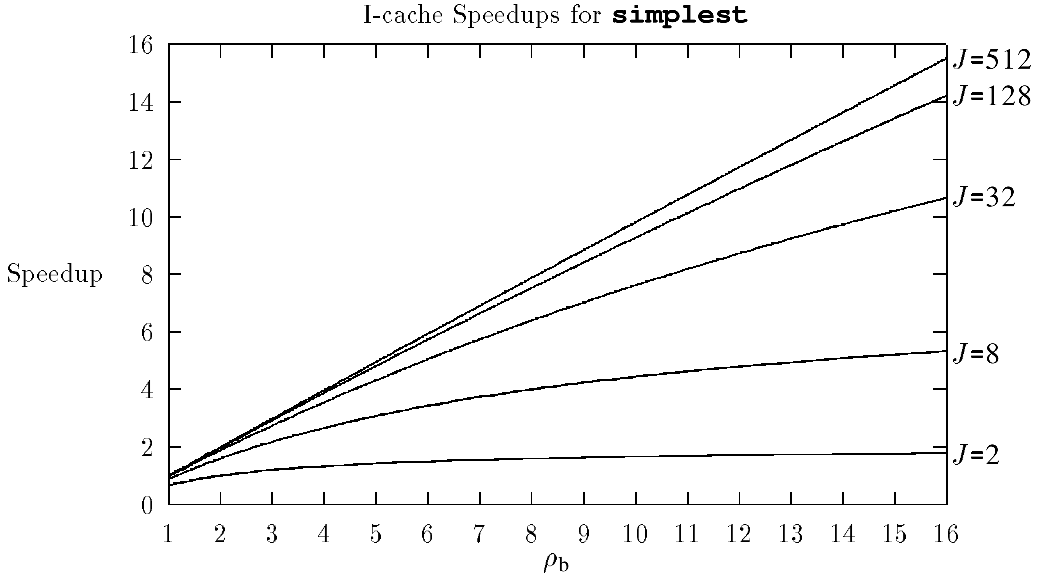


Figure 2.2: Loop iteration count J limits I-cache speedup for program **`simplest`**

Chapter 3

SIMD Computer Implementation

I-cache is an architectural element added to the PE chips of a SIMD computer to increase the rate at which repeated instructions are provided to the PEs. So I-cache changes the physical structure of the computer and the time required to perform a computation. Because it is explicitly managed by broadcast instructions, I-cache also changes the logical structure of a SIMD computation. To be able to assess the impact of these many changes, analysis of I-cache requires a model of the SIMD computer that encompasses both the physical structure of the machine and the logical structure of the computations it performs.

The SIMD computers produced over the years have been designed under continually changing technological constraints with a diversity of specific application targets. The great variety in PE architectures, sizes, interconnection topologies, and engineering cost relationships frustrates the goal of describing all SIMD computers with a single, universal model.

Nonetheless, it is possible to identify the salient elements that differentiate a SIMD computer from any other kind of multiprocessor. Furthermore, it is possible to combine those elements in a model that is parameterized to capture a broad range of implementation alternatives.

This chapter describes the model of generic SIMD computation underlying the analysis of I-cache. The model highlights the PE, whose replication in large numbers at least cost is the main objective of SIMD computer architecture. An example illustrates how the SIMD computer's components carry out computation.

The SIMD computer's subsystems that move information between chips (multi-chip subsystems, or *MCS*s) are represented in the model in the same way as PE function units (or *FUs*). This uniformity of representation facilitates writing assembly language programs that describe SIMD computations as sequences of FU and MCS operations without regard to the detailed implementation of these sometimes complicated components.

The SIMD computer executes machine code programs which specify the clock cycle by clock cycle activity. Assembly language programs are drawn on a sequential model wherein each instruction's execution completes before the next instruction's execution begins. Machine code programs reflect the physical characteristics of the computer, wherein the length of time required to complete an instruction depends on the operation it specifies, and wherein mutually independent operations are performed concurrently, resources permitting.

The most important aspects of this model of SIMD computation reflect the characteristics of VLSI-based implementation. The PE chips contain many PEs whose FUs can operate at the high rates attainable within chips. The MCS, for example that through which the PEs inter-communicate and that access memory external to the PE chip, contain wires that run between chips. Inter-chip wires typically present relatively large capacitances and long distances, and circuits containing them are typically slower than intra-chip circuits. Therefore, MCSs typically operate slower than PE FUs. The global instruction broadcast network is the MCS whose operation rate relative to that of the PE

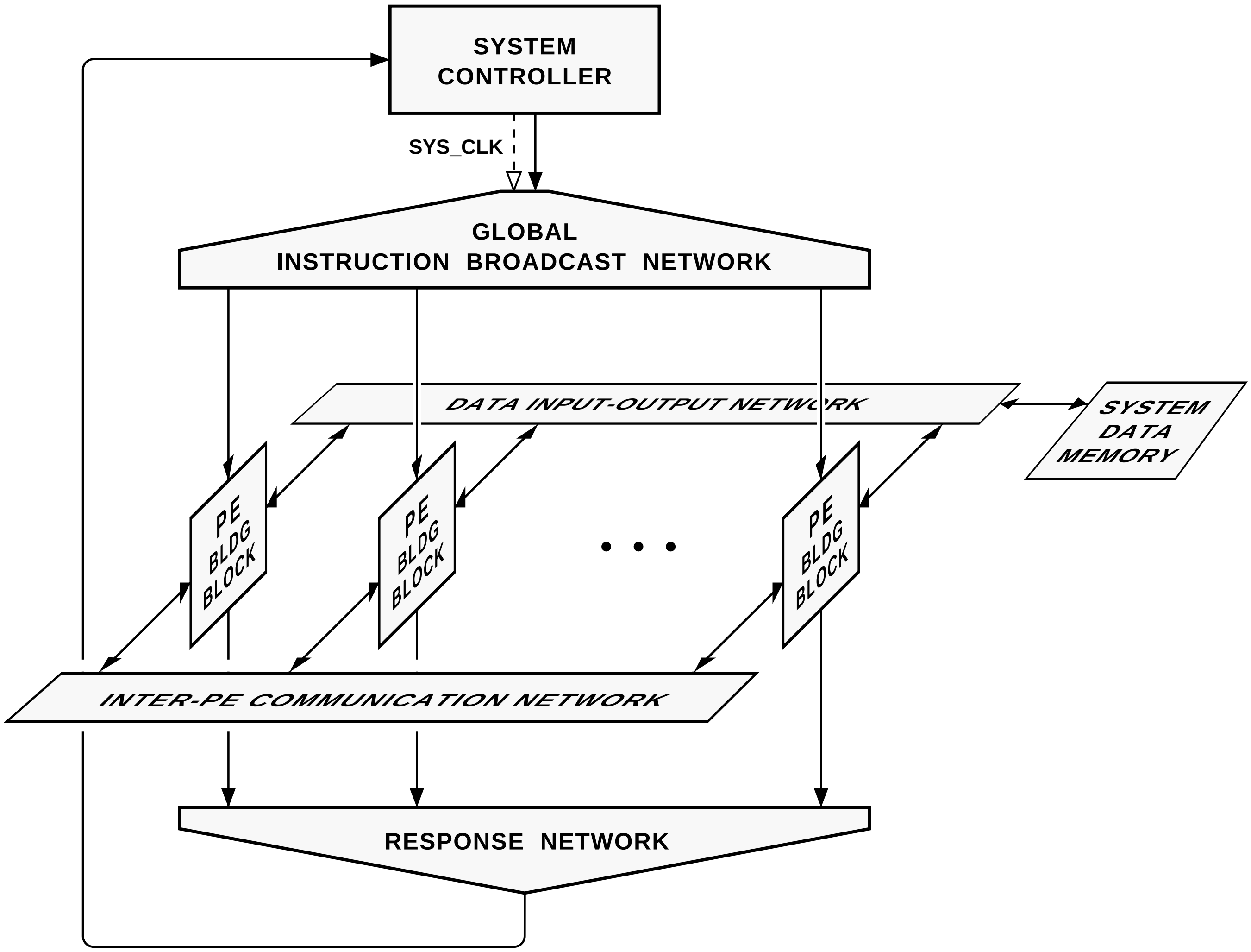


Figure 3.1: SIMD Computer

is crucial to the value of I-cache. Also important are the operation rates of the other MCSs; in the model, these rates are parametric relative to the PE clock rate.

The model allows I-cache to be described as a detailed change to the structure of a SIMD computation. The model also provides a basis for evaluating the throughput impact of I-cache.

3.1 Generic SIMD Computer

Figure 3.1 depicts a SIMD computer. The system controller generates a system clock that regulates all elements. The system controller also sequences instructions and broadcasts them via a *global instruction broadcast network* to an array of *PE building blocks*. Other elements of the computer, including an *inter-PE communication subsystem*, a *data I/O subsystem*, and a *response subsystem*, each comprises one or more chips connecting through inter-chip wires to at least one PE building block, and is each therefore a multi-chip subsystem, or *MCS*. The topology of the inter-PE communication network is a principal discriminator among SIMD computers; existing SIMD computers contain inter-PE communication network topologies ranging from linear [27] to grid [4, 30, 36] to multi-stage permutation [6, 8] to hypercube [42].

The system controller consists of a sequencer and a mechanism for evaluating loop-index-dependent expressions and providing the resulting literal values to the PEs. The system controller provides

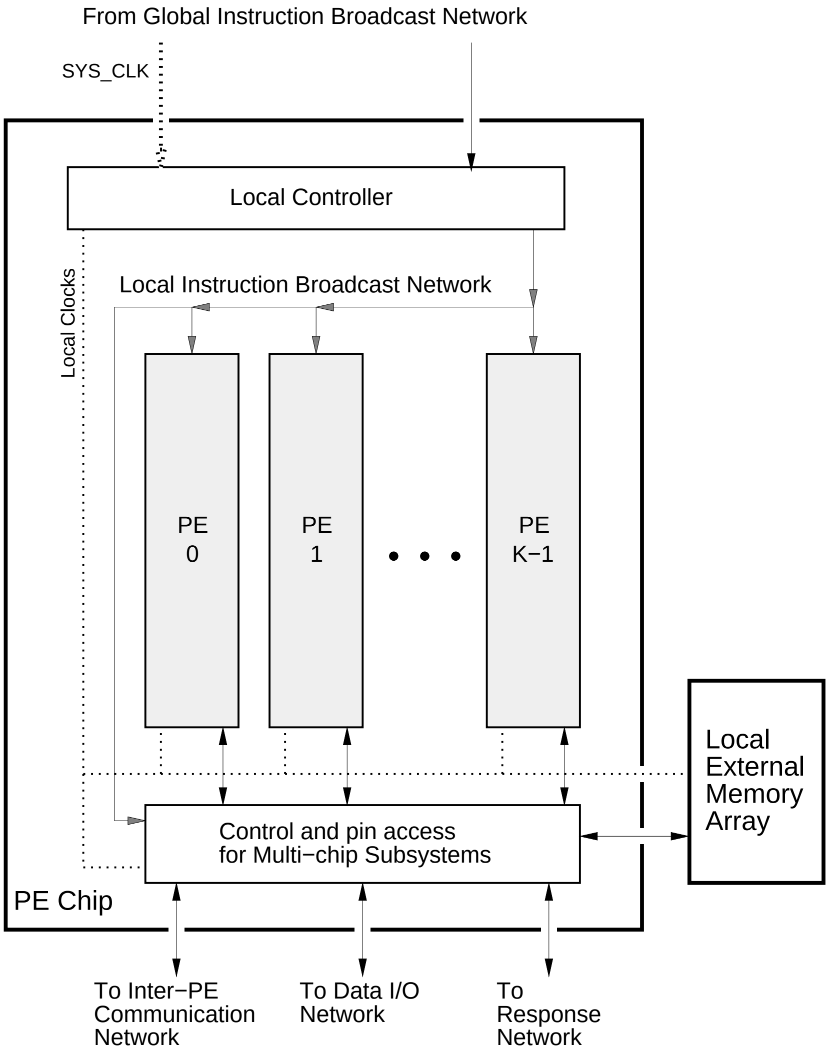


Figure 3.2: SIMD Computer Building Block

the program-control functions required in a SIMD computer. The system controller is here assumed to use a small set of single-cycle operations. An actual system controller may be optimized for specific computations being performed on a given computer, as for example is the case in SLAP [28]. The program-control functions performed by the system controller may be complicated. The potentially crucial topic of SIMD system controller design is beyond the scope of this work.

Figure 3.2 depicts a building block for a generic SIMD computer. The building block contains a PE chip connected via inter-chip wires to memory chips. The memory chips are included in the *local external memory subsystem*, an MCS realized within the building block to accommodate PE data requirements exceeding the memory capacity within the PE chip. The PE chip contains a *local controller*, a number of identical PEs, control and pin access for the MCSs.

Within the PE chip, the local controller provides cycle-by-cycle instructions to the PEs via a *local instruction broadcast network*. The local controller also generates the clocks regulating the PE chip's constituents; in a generic SIMD computer, there is only one such clock, standardized to the system clock. Figure 3.3 depicts a local controller for a generic SIMD computer.

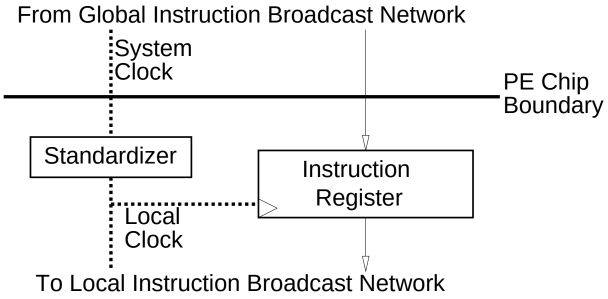


Figure 3.3: Generic SIMD Local Controller

3.2 Processing Element

The SIMD computer's PEs carry the brunt of the computational load. The PE is specialized for performing calculations, with the program-control functions being relegated to the system controller.

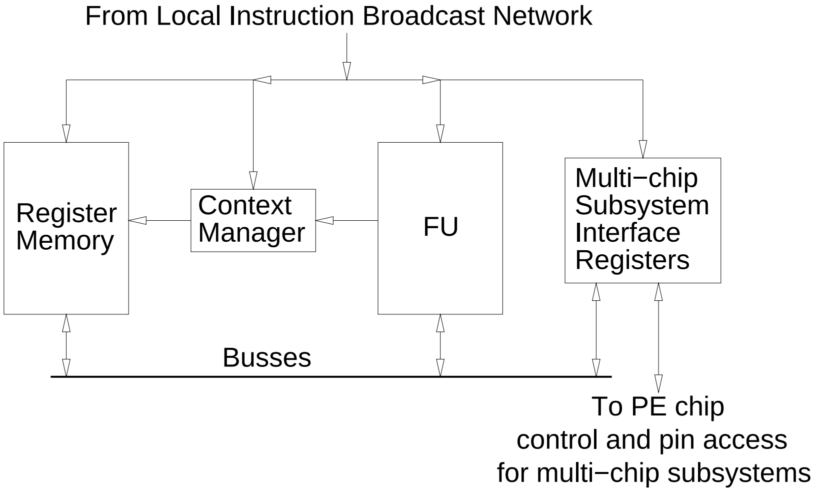


Figure 3.4: SIMD Processing Element

Figure 3.4 contains a sketch of the PE. The calculation component is essentially the same as the calculation component of a uniprocessor, consisting of a function unit (or *FU*) and register memory. Figure 3.4 indicates that the PE contains interfaces to the various MCSs, so that it may access external memory, communicate with other PEs, obtain input data sets, provide output data sets, and signal data-dependent conditions to the system controller.

The PE also contains a *context manager*, which performs a limited program-control function. The context manager allows a SIMD computer to execute *data-dependent* programs, wherein the sequence of executed instructions depends on values of intermediate results. Such data dependence arises, for example, from conditional **IF−THEN−ELSE** constructs in the program executed by the PEs.

The context manager maintains a one-bit control flag as directed by context management instructions contained in the globally broadcast instruction stream. These context management instructions conditionally set and clear the value of the control flag based on intermediate data values. When the control flag is clear, the PE is said to be "awake" in the current context, and it executes instructions as they are broadcast. However, when this control flag is set, the PE is said to be "asleep" in the current context. When the PE is asleep, all writes to PE state, including registers and external memory, are inhibited. Instructions broadcast when the PE is asleep do not change the state of the PE, and thus have no effect.

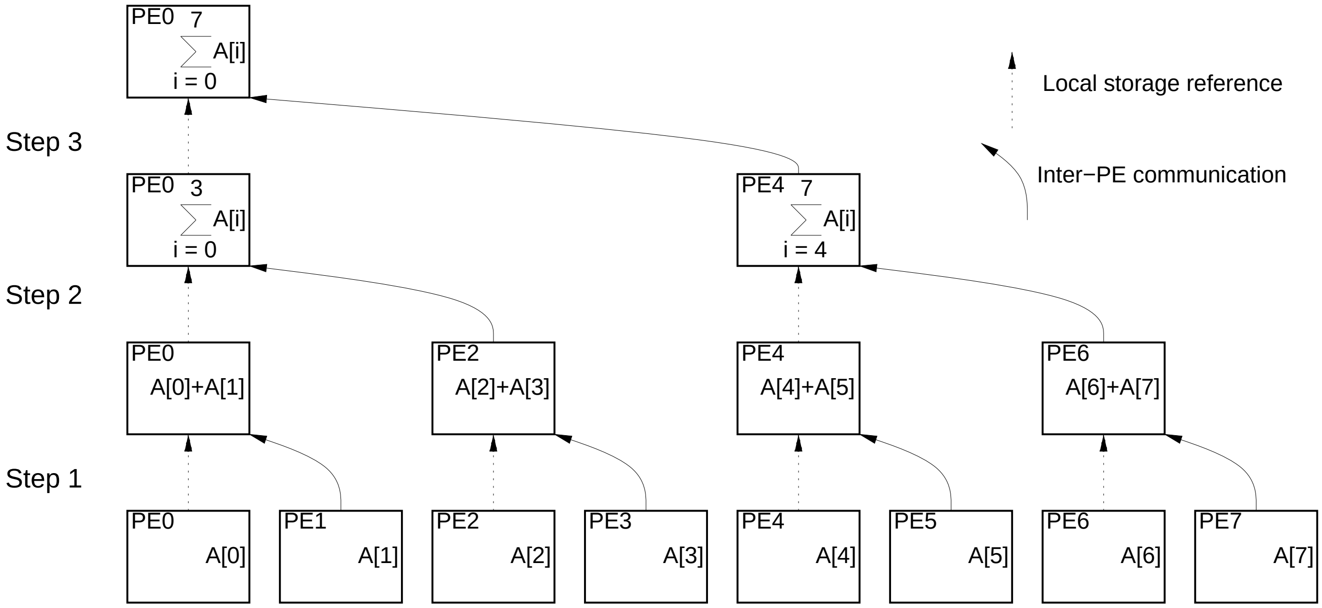


Figure 3.5: An example of tree-summation with P=8. The PEs communicate in a tree pattern to calculate the sum $\sum_{i=0}^{7} A[i]$. Time progresses upwards in the figure.

Conceptually, the context manager maintains a stack of contexts within the PE, pushing the value of the control flag whenever a new context is determined. The context manager may be implemented as an up/down counter with a small amount of associated control logic [27]. A context manager occupies far less chip area than do a uniprocessor's program storage and program sequencing elements.

3.3 An Example of SIMD Computation: Tree-Summation

The following example illustrates what the PE does over the course of a SIMD computation. In the computation described here, the PEs form the sum of an array of integers. This example is typical of PE activity in a SIMD computation, because applying an associative operator to elements of an array occurs often in data-parallel computations. The program-control activity of the system controller, I/O, and the details of communication are not emphasized here, in favor of focussing on the PE.

In this example, the PEs are inter-connected by a routed inter-PE communication network. To perform communication, each PE specifies the index of a target PE and a value to be sent to that PE. The flexible inter-PE communication allows the summation to be performed in $\log_2 P$ steps on P PEs, where there are P integers to be added.

Figure 3.5 illustrates the successive steps of the tree-summation for the case where P=8. Initially, the array is stored one element per PE. Figure 3.5 shows that the pattern of communicating PEs forms a binary tree. At each leaf of the tree is a PE storing one element of the array. During each step of the computation, values move one step closer to the root of the tree. Each active PE sends its value to its parent, and the parent adds together the values it receives. The parent becomes a child on the next step. Figure 3.5 suggests that one of the two children sending a value to a parent resides in the same PE as the parent. A PE becomes inactive following a step when its parent in the tree resides in a different PE. After $\log_2 P$ steps, the root of the tree contains the sum of the P elements.

The algorithm sketch in Figure 3.6 shows the sequence of operations performed by the PE. The computation consists of a loop wherein each PE maintains a value, **sum**. **sum** is initialized in each PE to contain its assigned array element. (Where the PE index is denoted π, **sum** in PE π is initialized

```
sum = $A$[$\pi$];
targmask = -1;                 /* initialize target mask to all 1's */
for (i=0 ; i<$\log_2 P$; i++)
{
   targmask = targmask << 1;   /* shift target mask left one position */
   targ = $\pi$ && targmask;
   if (targ == $\pi$)
      targ = -1;               /* do not send to self */
   rxval = route(sum,targ);    /* send sum to PE whose index is targ, where
                                  it is stored in rxval.  */
   if (targ != -1)
      sleep;                   /* Once its sum has been sent to another PE,
                                  this PE becomes inactive. */
   sum = rxval + sum;          /* Active PE accumulates received value. */
}
wakeup PE;
```

Figure 3.6: The operational structure of the tree-summation loop. π denotes the PE index ($\pi \in \{0 \ldots P-1\}$). A[] is the array whose elements are added together, and A[π] resides in the memory of PE π. `sum`, `targmask`, `targ`, and `rxval` are local PE variables. The final sum ends up in PE 0.

to contain array element A[π].) In the loop body, the PE determines the index of its current tree parent and sends `sum` to the parent. If the PE is not itself a parent at this step, then it is asleep for the remainder of the tree-summation. Otherwise, the PE remains awake and replaces `sum` with the sum of its childrens' values. The algorithm sketched in Figure 3.6 is executed in lock-step on the array of PEs.

The loop control variable i is maintained in the system controller, not in the PEs, because the system controller performs all of the program-control functions. The PE operations shown in Figure 3.6 are carried out using the components shown in Figure 3.4. The local PE variables used in the loop body are kept in register memory. The first operation retrieves A[π] from local external memory and places the value in `sum`. Local external memory is an MCS shown in Figure 3.2. If the variable `targmask` is maintained in the PE, then the shift operation is performed in the FU. However, the value of `targmask` is not data dependent and it has a common value in all PEs. Therefore, `targmask` could be maintained once for all PEs by the system controller. In either case, the loop body begins with FU operations to calculate `targ`, the index of the PE to whom the current value of `sum` will be sent. `sum` is then transmitted through the inter-PE communication network (if `targ` $\neq \pi$). The inter-PE communication network is an MCS shown in Figure 3.1. After sending its sum to PE `targ`, a PE is deactivated for the remainder of the computation. PEs that remain active are the tree parents at the current step. An active PE receives a value from the inter-PE communication network in the register memory variable `rxval`. At the end of the loop body, the FU adds `rxval` and `sum`, placing the result back into register memory.

By the end of the last iteration of the loop, all PEs except PE 0 are asleep. The instruction following the loop body awakens the PEs, removing the contexts pushed during the loop's execution.

3.4 Representations of Multi-Chip Subsystems

Each MCS may comprise control and pin time-sharing sub-circuits within the PE chip, inter-chip wires, and components contained in chips other than PE chips. The PE's interface to an MCS

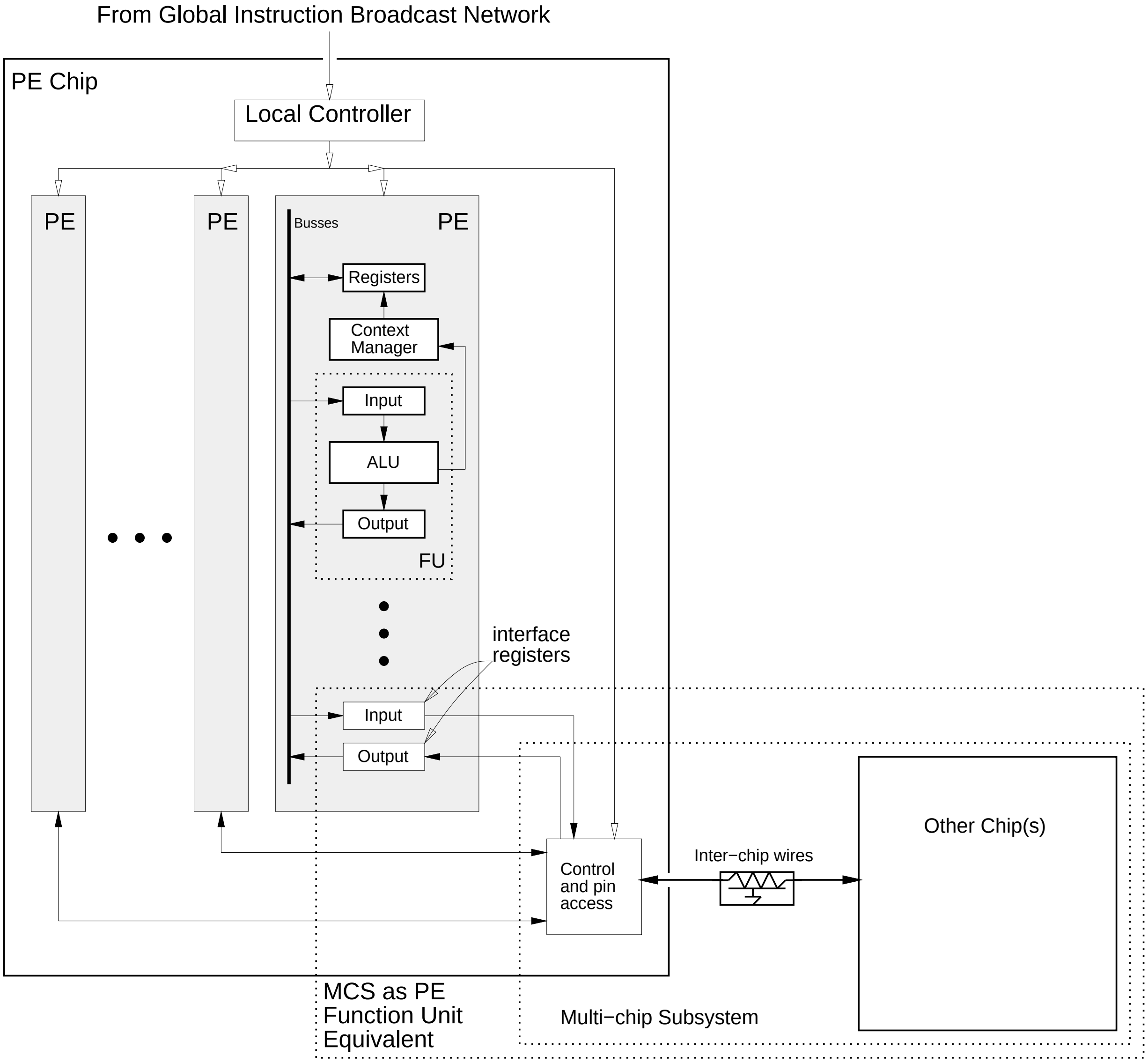


Figure 3.7: Generic Abstraction of a Multi-Chip Subsystem as a Function Unit Equivalent

consists of input and output registers containing data sent out and received through the subsystem. The numbers of input and output registers in the PE for each MCS depend on the MCS function; either (but not both) of these numbers is 0 in some cases.

The interface registers associated with each MCS make each MCS behave like the FU, from the point of view of activity on the PE busses: “operands” are stored into input registers from busses, a designated “operation” is performed, and at the completion of that operation a “result” may be driven onto a bus from the output register. Figure 3.7 depicts this analogy between MCS and FU. Whereas the FU performs arithmetic or logical calculation using an ALU contained in the PE chip, an MCS performs local external memory access, inter-PE communication, system data memory access, transmission of a logical value to the system controller, or delivery of a broadcast literal, using circuits that include inter-chip wires.

The abstraction of MCSs as FU equivalents simplifies programming, in that it allows an assembly language program to specify MCS activity as register-to-register PE operations, without regard to the detailed implementation of the MCS.

Variable	π	`sum`	`targmask`	`targ`	`rxval`
Register	R0	R2	R3	R4	R1

Table 3.1: Register Assignments for Tree-Summation Loop Body

Table 3.1 shows a register assignment for the variables used in the tree-summation loop. Figure 3.8 shows an assembly language program for the tree-summation loop using the register assignments shown in Table 3.1. A **`LOAD`** operation is used to fetch $A[\pi]$ from local external memory, while a **`ROUTE`** operation specifies routed inter-PE communication. MCS operations resemble FU operations in the code in Figure 3.8. The code in Figure 3.8 also demonstrates the context management operations used to control conditional execution of instructions. (Assembly language syntax is detailed in Appendix B.)

3.5 Constraints Arising from VLSI Implementation

At least since the advent of microprocessors, the characteristics of VLSI implementation technique have constrained computer throughput. The minimum time required for a VLSI circuit to change the voltage on a capacitor through a wire grows with both the driven capacitance and the length of the wire. A characteristic of VLSI circuits is that capacitances tend to be greater and wire lengths tend to be longer between chips than within chips. Therefore, a key property of VLSI-based systems is that intra-chip circuits tend to be faster than multi-chip circuits.

A VLSI manufacturing process determines the maximum number of transistors that can fit into a chip, although wires that inter-connect transistors typically cause the actual number of transistors realized in a chip to be well below the maximum. The maximum number of transistors is a function of the process resolution (characterized by the length unit λ [58](p.48)) and of the physical dimensions of a chip that is manufacturable with acceptable yield. Commercial factors tend to cause λ to decrease and the physical size to increase over time [23]. The constituents of the PE chip of a SIMD computer compete for the limited resources available within the chip. As suggested in Figures 3.2 and 3.4, these resources are shared among the FU, context manager, and registers of each PE, MCS local control and interfaces, and the local controller.

Chips have limited numbers of external connections. If the input and output pads to which inter-chip wires are attached are arrayed around the periphery of a chip of edge length N, then there are $O(N)$ such pads. This number grows more slowly with N than does the maximum number of transistors per chip, which is $O(\frac{N}{\lambda})^2$. As VLSI implementation technique improves, N grows while λ decreases. Therefore, the ratio of transistors to pads arranged around the periphery grows as $O(\frac{N}{\lambda^2})$. The placement of I/O structures around the periphery is not necessary for VLSI implementation of the PE chip; however, it is characteristic of inexpensive present-day chips.

A large proportion of the chips in a SIMD computer are PE chips. As indicated in Figure 3.2, each PE chip is accompanied by a set of memory chips. Other chips are sometimes required for inter-PE communication, for system data memory access, for problem-specific I/O, or for transmitting status information to the system controller. The continual use of a global instruction broadcast subsystem distinguishes a SIMD computer from other multiprocessors. The global instruction broadcast network fans out from the system controller to all PE chips in the system. The total capacitive load presented by the PE chips' global instruction receiver pins is large, and the worst-case distance of these pins from the driver is likely to be as large as that for any signal driven in the SIMD computer.

```
program tree_sum_loop(logP,&A_pi):
! The tree-summation loop.  Parameters:
! logP is log(P), P the number of array elements and PEs
! &A_pi is the address in local external memory of Aπ
        LDX IR0 `0' ; R1 = LITERAL(`&A_pi')
        ; R2 = LOAD(R1)                ! Initialize  sum
        ; R3 = PASS(`-1')              ! Initialize  targ
        CJSR FORC LOOP `logP - 1' ;  ! logP iterations at label LOOP
        ; [CLR]                        ! Awaken all the PEs
        HALT ;
LOOP:
        ; R3 = LSHIFT(R3,`1')
        ; R4 = AND(R0,R3)
        ; LC_PUSH_EQ(R4,R0)         ! Sleep unless  targ == π
        ; R4 = PASS(`-1')            !  targ = -1 ⇒ send to noone
        ; [POP] R1 = ROUTE(R2,R4)  ! perform inter-PE communication
        ; LC_PUSH_NE(R4,`-1')       ! Remain awake only if a parent this step
        ; R2 = ADD(R1,R2)
        LTST ICT0 LOOP ;             ! iterate until loop counter reaches 0
```

Figure 3.8: Assembly language for the tree-summation loop. Each line specifies one assembly language instruction. To the left of the semicolon is a system controller instruction, to the right is a PE instruction. Exclamation point begins a comment that continues to end of line. The PE instructions of the form **`LC_PUSH_`** create a new context in which the PE is conditionally asleep, depending on the value of the specified condition in the PE. The part of the PE instruction in brackets also manipulates local context, reverting to the previous context (in the case of **`[POP]`**) or unconditionally waking up the PE (in the case of **`[CLR]`**). The system controller instruction **`CJSR`** begins executing the loop body, while the system controller instruction **`LTST`** performs the loop completion test.

3.6 Operation Stepcounts

Limited chip area means that there might not be as much chip area available as desired in the PE chip for FUs and register files. Also, limited pins means that there may not be as many pins available as desired per MCS. A common compromise introduced by such resource constraints is to time-multiplex the available resources. FU or MCS operations that are time-multiplexed take multiple clock cycles to complete.

One way that FU time-multiplexing arises is when the FU has a width (in bits) that is less than the width of the operands. For example, the MP-1 PE's 4-bit FU adds 32-bit integers in 8 steps [62]. Another way that FU time-multiplexing arises is when the circuit complexity of the FU is less than that required for a given operation. For example, there may not be sufficient chip area for a combinational multiplier, and multiplication may be carried out as a sequence of additions. The SLAP PE provides an example, wherein its 16-bit FU (with a built-in Booth's bit-pair recoder) multiplies 16-bit integers in 8 steps [27].

Depending on the degree of time-sharing of PE chip pins and on network design, an MCS may use multi-step sequences to perform its data transfers. For example, local external memory access through a shared port requires a number of steps proportional to the number of PEs in the chip. As another example, the SLAP I/O subsystem delivers a new datum to each PE in a number of steps proportional to the total number of PEs [25].

The need for multiple steps to carry out an operation is the principal difference between assembly language instructions and machine code instructions. An assembly language instruction specifies an FU or MCS operation. Assembly language semantics are sequential, in which each instruction's operation has completed before the next instruction begins executing. Machine code semantics, by contrast, allow that multiple machine clock cycles are sometimes needed to perform a given operation.

To make it possible to represent SIMD computers that vary in these ways, a *stepcount* parameter is associated with every assembly language FU and MCS operation. FU operation stepcounts characterize FU bit-width and circuit complexity relative to the requirements of application data, while MCS operation stepcounts characterize PE chip pin sharing and inter-chip network complexity. The stepcount of an operation gives the number of machine clock cycles in the underlying SIMD computer required to perform it. As a simplification, the model prohibits pipelined operations.

For some operations, the actual number of steps depends on the specific data involved. This dependence occurs, for example, in logical word rotation using a distance-1 barrel shifter. This dependence also occurs in the general case of routed inter-PE communication. Assigning fixed numbers of clock cycles to operations is sometimes an imperfect approximation that compromises the validity of results. The sensitivities of results to these approximations are identified and compensated where appropriate in the analysis of measured results.

As an example of how stepcounts characterize a SIMD computer, the following is a list of stepcounts for some operations of a computer operating on 32-bit integers with 16-bit PEs packed 4 to a PE chip (as in SLAP [26]) and inter-connected through a three-stage permutation network (as in GF11 [6]):

	System Controller Instruction				PE Module Instruction						
#	SC_Op'n	f1	f2	f3	C	Dest	Operation	SrcA	SrcB	L_B_Value	r_d_c
0	LDX	0	0			REG.1	LITERAL.0			17	
1							LOAD_TX.0		LIT_OUT		
2							PASS.1	-1			
3						REG.3	PASS.0 [N]				
4						REG.2	LOAD_RX.0				
5	CJSR	FORC	16	9							
6					CLR						
7											
8											
9											
10											
11											
12											
13											
14											
15	HALT										
16							LSHIFT.1	REG.3	1		
17						REG.3	LSHIFT.0 [N]				
18							AND.1	REG.0	FU_OUT		
19						REG.4	AND.0 [N]				
20							LC_PUSH_EQ.1	FU_OUT	REG.0		
21							LC_PUSH_EQ.0 [N]				
22							PASS.1	-1			
23						REG.4	PASS.0 [N]				
24											
25							ROUTE_TX.0	REG.2	REG.4		
26											
27					POP	REG.1	ROUTE_RX.0				
28							LC_PUSH_NE.1	REG.4	-1		
29							LC_PUSH_NE.0 [N]				
30							ADD.1	REG.1	REG.2		
31						REG.2	ADD.0 [N]				
32	LTST	ICT0	16								

Figure 3.9: Machine code for the tree summation loop. Here, P=1024, the array element $A[\pi]$ happens to be at address 17 in PE memory (so &A_pi =17), operands are 32 bits wide, PE FUs are 16 bits wide, there are 4 PEs per chip, and inter-PE communication uses a 3-stage permutation network.

Operation Name	Meaning	Stepcount
LITERAL	Broadcast literal	2
AND	Logical and	2
ADD	Addition	2
LSHIFT	Left shift	2
LC_PUSH_EQ	Push new context: PE is active if operands are equal	2
LC_PUSH_NE	Push new context: PE is active if operands aren't equal	2
PASS	Logical identity	2
LOAD	Local external memory load	4
ROUTE	Routed inter-PE communication	3

Using the stepcounts in the above table and assuming that P=1024 and that array element A_π lies at address 17 in PE memory, the assembly language program in Figure 3.8 results in the machine code program shown in Figure 3.9. (Machine code is detailed in Appendix A.)

3.7 PE Chip Model

For scalable data-parallel problems, throughput is proportional to the number of PEs. To maximize the throughput for scalable data-parallel problems, SIMD computer architecture aims to maximize the number of PEs realized in a given total chip area. To that end, the PE of a SIMD computer is specialized, consisting mostly of FU and registers (as sketched in Figure 3.4). This specialization allows most of the chip area occupied by the PE to contain the components needed to perform calculations. PE chip *payload* is a chip-area measure that expresses the "amount of PE" contained in a PE chip. This section presents a generic model of a PE chip, develops a formula for payload, and applies the formula to a number of examples. Payload is a useful way to compare PE chips made using different VLSI implementation techniques. As shown in Chapter 4, payload provides a uniform quantitative basis for estimating the cost of I-cache.

It is difficult to find a uniform payload metric, because there are many alternative implementations of a PE chip containing PEs of a given architecture. VLSI implementation technique encompasses alternatives in logic designs, in circuit designs, in geometric layout, and in chip fabrication process. Diverse logic structures for MOS chips are presented systematically in [83]. As an example of the range of circuit design techniques, circuits with active storage elements (flip-flops) are usually easier to design than their counterparts with passive storage elements (capacitances), although the former tend to be larger. As an example of the range of layout techniques, automated layout of circuits is typically easier than the manual alternative, although the latter method tends to yield smaller and faster circuits. VLSI fabrication processes vary in geometric resolution, in design rules, in the physical sizes of switching devices, and in the electrical characteristics of the switching devices.

The diversity of VLSI implementation techniques defies uniform payload comparisons among PE chips. However, assuming similar design techniques have been used, it is possible to compare the payloads of chips based on their physical characteristics alone. The parameters H and W represent the physical dimensions of the PE chip in mm. Typical values for H and W for microprocessors today range from about 10mm to more than 16mm. The parameter λ represents the geometric resolution of the VLSI fabrication process in μm [58](p.48). Typical values of λ for processes used to make microprocessors today range from as low as 0.3μm to as high as 1.0μm.

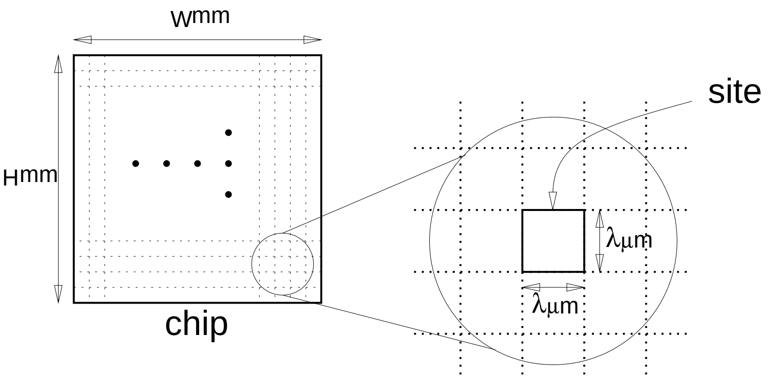


Figure 3.10: Grid of Sites on a Chip

To approximate its area, a chip can be thought to contain a grid whose resolution is $\lambda \times \lambda$, as sketched in Figure 3.10. Each grid square is a *site* on which can be placed integrated circuit elements, including switching devices or wires. A site is $\lambda\mu$m on a side and occupies chip area $\lambda^2 \times 10^{-12}$m. The total number of sites in a computer system is sometimes called its "grain size" [70](p.1252). The total number of sites on a chip is shown in Equation 3.1.

$$\begin{aligned}\frac{\text{\# sites}}{\text{chip}} &= \frac{H\text{mm} * W\text{mm}}{(\lambda\mu\text{m})^2} \\ &= \frac{HW}{\lambda^2} \times 10^6\end{aligned} \tag{3.1}$$

The total number of sites on a chip is determined solely by VLSI fabrication process characteristics. This number ranges over about an order of magnitude for current chips, from about 10^8 to just beyond 10^9.

The site is a unit of chip area that scales with H, W, and λ. For a given number of sites per switching device, the number of devices scales with the number-of-sites measure of variations in VLSI fabrication process analogously to the scaling of circuit geometries with the linear resolution parameter λ.

Although the "amount" of PE realized in the available chip area depends on the entire VLSI implementation technique, PE chip payload is proportional to the number of sites available for PEs. Π denotes payload, and Π is defined in Equation 3.2:

$$\Pi \equiv \text{\# of PE chip sites used for PEs} \tag{3.2}$$

The chip area occupied by a PE depends both on the PE architecture and on the VLSI implementation technique. PE architecture encompasses such details as datapath width, register count, MCS interface design, and FU circuit complexity[1].

Existing SIMD PE chips are typically organized as linear arrays of bit slices. Figure 3.11 illustrates such a PE chip organization. On-chip linear arrays are used in the MP-1 PE chip (pictured in [56]), in its precursor designed at DEC (pictured in [33]), and in the SLAP PE chip (pictured in [27]). The PE area in the Blitzen PE chip is tiled with a two-dimensional array of ALU-memory pairs, as shown in [36]. The Blitzen PE chip is equivalent to a linear array of bit slices, wherein each bit slice consists of FU and registers alternating in the vertical dimension. The four examples mentioned here are representative of the small number of SIMD PE chips for which microphotographs of working chips have been published.[2]

Figure 3.11 suggests that I/O structures occupy a ring around the perimeter of the PE chip. This placement of I/O structures is not necessary for VLSI implementation of the PE chip, although it is characteristic of inexpensive chips fabricated using present-day VLSI implementation technique. The sites within this perimeter are not available for PEs. Figure 3.11 reflects that assumption that the thickness of the I/O ring is 500μm, as happens to be the case for each of the three PE chips examined in detail in this Chapter. Given these assumptions, Equation 3.3 defines I, the *interior* chip area of the PE chip that is available for PEs:

$$\begin{aligned}I &\equiv \frac{\text{interior chip area}}{\text{PE chip}} \\ &= \frac{(H-1)\,(W-1)}{\lambda^2} \times 10^6 \ \text{sites}\end{aligned} \tag{3.3}$$

---

[1] The *circuit complexity* of the FU refers loosely to the amount of arithmetic work the FU performs in a single operation. For example, a combinational multiplier array performs multiplication in a single clock cycles' operation, whereas an adder with a built-in multiply step performs multiplication over a sequence of clock cycles. A combinational multiplier has greater complexity than a simple adder, which typically implies that the multiplier would occupy greater chip area and operate at a lower maximum rate.

[2] Floor plans for other SIMD PE chips, including those shown in [87] and in [68], suggest on-chip linear array organization. CLIP7A, an early VLSI-based SIMD computer, contains just 1 PE within its relatively small PE chip [32].

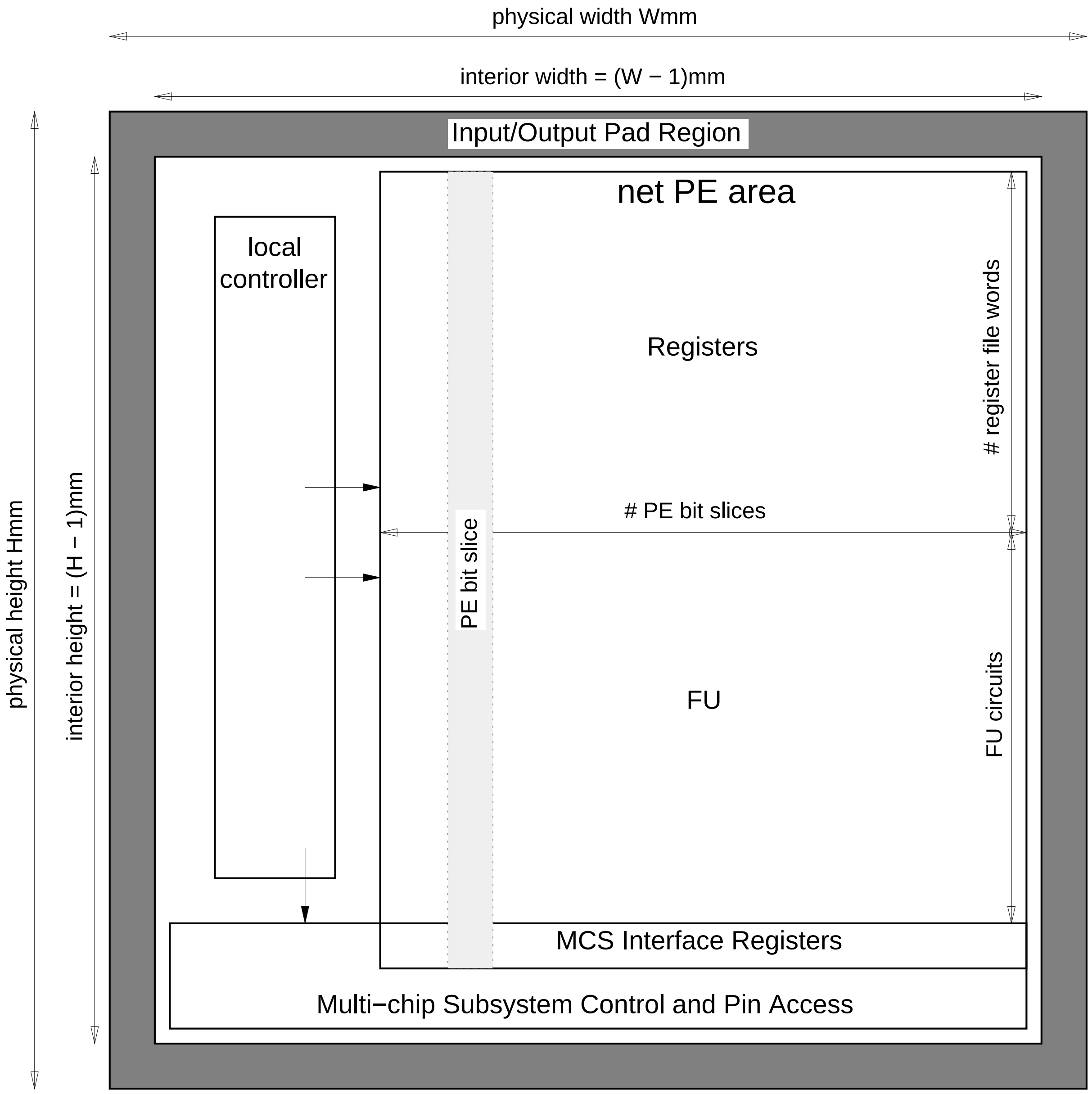


Figure 3.11: Generic SIMD PE Chip Floor Plan

Ideally, the interior of the PE chip would be used entirely for payload. In practical SIMD computers, some proportion of the interior chip area is occupied by the local controller and by MCS control and pin-access circuits, as suggested in Figure 3.11. A denotes the interior chip area that is occupied by the local controller and by MCSs, while Π denotes the remaining interior chip area. A and Π are related to I as per Equation 3.4:

$$\begin{aligned} I &= \text{non-PE interior chip area} + \text{payload chip area} \\ &= A + \Pi \ \text{ sites} \end{aligned} \tag{3.4}$$

Whereas I depends only on VLSI implementation technique, A depends also on local controller design and on MCS interface requirements. Therefore, the payload Π depends severally on local controller design, on MCS interface requirements, and on VLSI implementation technique.

Substituting for I from Equation 3.3 into Equation 3.4 yields the the PE chip payload formula shown in Equation 3.5:

$$\begin{aligned} \Pi &\equiv \text{\# of PE chip sites used for PEs} \\ &= \text{interior area} - \text{non-PE interior area} \\ &= \frac{(H-1)\,(W-1)}{\lambda^2} - A \ \text{ sites} \end{aligned} \tag{3.5}$$

Equation 3.5 gives a definition for PE chip payload that is independent of PE architecture. Equation 3.5 suggests that payload depends only on the following characteristics of the PE chip:

- VLSI implementation technique,
- MCS interface requirements,
- and local controller design.

Note that only the last of these characteristics changes when I-cache is added to the PE chip.

Equation 3.5 gives the payload Π as a function of H, W, λ, and non-PE interior area A. It is possible to calculate the payload of a PE chip given a microphotograph of the chip, if the VLSI fabrication process parameters are known. Published microphotographs of PE chips reveal floorplans that conform roughly to that sketched in Figure 3.11, with the exceptions that the MCS interfaces occupy an annular ring just inside the I/O ring, and the local controller occupies a vertical strip down the center of the chip instead of being to one side of the PE area as shown in Figure 3.11. The dimensions of the local controller and the MCS interfaces can be measured in the microphotograph using a ruler. Adding together these measured dimensions and then dividing by the total interior area yields an estimate of the fraction of the interior area used for MCS interfaces and for the local controller. Multiplying this fraction by the interior area (I, defined in Equation 3.3) yields non-PE interior area A. Finally, substituting this value for A and the known physical parameters into Equation 3.5 yields an value for PE chip payload Π.

Table 3.2 shows a summary of the physical parameters and payload for four PE chips: SLAP, MP-1, Blitzen, and ALAPH. The first three of these chips have been described in the literature, while the fourth is hypothetical, based on current VLSI process technology.

The SLAP PE chip contains 4 16-bit PEs [27], organized as one row of 64 bit slices. Inspection of the SLAP chip (shown in the photo in [27]) indicates that the local controller and the MCS interfaces together occupy about 36% of the interior area.

	SLAP	MP-1	Blitzen	ALAPH
H (mm)	9.2	9.5	11.0	13.9
W (mm)	7.9	11.6	11.7	16.8
λ (μm)	1.0	0.8	0.5	0.375
interior area I ($\times 10^6$ sites)	57	141	428	1400
non-PE area fraction $\frac{A}{I}$ (%)	36	17	17	
non-PE area A ($\times 10^6$ sites)	20	23	72	100
payload Π ($\times 10^6$ sites)	36	118	356	1300
PE FU width (bits)	16	4	1	
fraction of PE bit-slice occupied by registers (%)	20	34	60	

Table 3.2: Physical Parameters and Payload Estimates for Four SIMD PE Chips

The MP-1 PE chip contains 32 4-bit PEs [62], organized as one row of 128 bit slices. Inspection of the MP-1 chip (shown in the large color photo in [56]) indicates that the local controller and the MCS interfaces together occupy about 17% of the interior area.

The Blitzen PE chip contains 128 1-bit PEs [36], organized as 8 rows of 16 PEs per row. Inspection of the Blitzen chip (shown in the large photo in [37]) indicates that the local controller and the MCS interfaces together occupy about 17% of the interior area.

The ALAPH PE chip is a hypothetical one whose parameters are taken from the VLSI process used for the Alpha microprocessor [21]. The estimate for ALAPH's non-PE area A of 100×10^6 sites is conservative given the values for the existing chips. ALAPH would contain two or more 32-bit PEs, each with at least 2000 registers.

Table 3.2 shows the proportion of the PE bit slice occupied by registers in the three existing PE chips. It is interesting to note that the proportion of chip area allocated to registers increases as the FU bit-width decreases.

Another interesting feature of Table 3.2 is that the ratio $\frac{A}{I}$ happens to be 17% in both the Blitzen and MP-1 PE chips, whereas SLAP's non-PE area represents twice that fraction. There are two likely sources of this disparity:

1. Multi-chip subsystem interface complexity.

   Local external memory access in Blitzen and in MP-1 has the characteristic that the system controller provides 1 address through the global instruction broadcast network that is used for all PEs' addresses. By contrast, the SLAP PEs each supply their own addresses to local external memory. For this reason, the local external memory interface within the SLAP PE chip is occupies more chip area than its counterparts in the other two PE chips.

2. Local controller complexity.

   Whereas the Blitzen and MP-1 local controllers occupy negligible area in their respective PE chips, the SLAP instruction decoder occupies almost 20% of the interior. This difference is due to the different functions performed by the local controllers in these chips.

   In all three of these PE chips, machine code instructions are executed in a pipelined manner. Typically, there are three execution phases, consisting successively of:

(a) register operand fetch,

(b) FU calculation, and

(c) register result write.

Successive pipeline stages execute on successive PE clock cycles. A machine code instruction requires three clock cycles to complete, and on any given clock cycle, three successive machine code instructions are in execution. The instruction provided to the PEs within the PE chip on each clock cycle controls a single clock cycle's activity, and therefore has contributions from each of the three instructions currently in execution.

In SLAP, the system controller broadcasts machine code instructions. The SLAP PE chip's local controller decomposes each arriving instruction into its partial contributions on each of three successive PE clock cycles to the single-cycle control word provided to the PEs. In Blitzen and in MP-1, by contrast, the system controller broadcasts single-cycle PE instructions. Instead of broadcasting the machine code instructions themselves, the system controller assembles each globally broadcast instruction from the relevant parts of the three machine code instructions currently in execution.

To perform its pipeline control function, the SLAP local controller contains pipeline staging registers and decode logic that are not present in the Blitzen and MP-1 local controllers. The design choice in SLAP for the local controller to perform pipeline control decoding represents a departure from strict SIMD computer architecture. As for other program-control functions, this decoding is more efficiently performed once for all PEs in the system controller, rather than redundantly replicatedly within each PE chip. The relatively large instruction decoder in SLAP highlights the large chip-area cost paid for non-trivial instruction decoding within the PE chip.

3.8 How Large is ρ_b?

ρ_b expresses the ratio of maximum PE clock rate to global instruction broadcast rate. The speedup due to I-cache depends on the value of ρ_b, which is in turn determined by characteristics of chips and their interconnections. There is a range of possible values of ρ_b. In some existing SIMD computers, ρ_b appears to be greater than 10. Detailed estimates suggest that ρ_b would be at least 2 in practical SIMD computers made with foreseeable implementation technique. High-rate instruction broadcast incurs significant PE chip pin and board wiring costs. Minimizing these costs leads to lower instruction broadcast rates, so that a new SIMD computer whose resources are concentrated in the PE chips instead of in the instruction broadcast network might exhibit ρ_b even higher than observed in existing SIMD computers.

PE clock rate depends solely on the characteristics of the PE chip, whereas global instruction broadcast rate depends on the size of the computer and on the electrical characteristics of the broadcast network. For a given PE architecture, the PE clock rate is determined within a fairly narrow range by the VLSI implementation technique. By contrast, the sizes of existing and foreseeable SIMD computers vary widely. Variations in computer size and in system-level implementation techniques give corresponding variations in the broadcast rate. For example, a very small SIMD computer might fit within a single chip, wherein instructions are delivered to the PEs at the high on-chip rate. At the other size extreme, a very large SIMD computer would fill a large chassis, wherein the rate of instruction broadcast might be considerably lower than the PE clock rate. While it is possible in principle to construct a network that broadcasts instructions at any reasonable PE clock rate for a SIMD computer of any size, so doing requires precise matching of wire lengths and of electrical component characteristics. For some SIMD computers, such precision is prohibitively expensive.

Maximum PE clock rate is determined by VLSI implementation technique and by PE architecture. Current single-chip systems, including microprocessors, provide a basis for estimating the PE clock rate achievable with readily foreseeable VLSI implementation technique. For example, a 32-b PE made using recent CMOS VLSI implementation technique should operate at rates well beyond the rate of 200MHz achieved for a 64-b microprocessor [21], while a 32-b PE made using recent BiCMOS VLSI implementation technique should at least equal the 300MHz achieved for a 32-b microprocessor [47]. Simplicity of function typically shortens critical paths and allows higher clock rates. For example, operation rates of one-bit arithmetic components have been measured in excess of 250MHz using 3μm CMOS [89], while a 333MHz 32-b adder has been implemented in 0.5μm BiCMOS [35]. These examples suggest that the clock rate for a new PE would lie between 200MHz and 300MHz, or possibly even higher if the function unit operates on narrow words.

Having based PE clock rates on those of microprocessors, it makes sense to extrapolate from the disparity between on-chip and off-chip clock rates observed for modern microprocessors to estimate ρ_{b} directly. Most modern microprocessors operate internally at least two times faster than their external memory interfaces [21, 88, 5]. A primary reason for this disparity between on-chip and inter-chip operation rates is that off-chip wires are driven as lumped capacitances: The capacitances of inter-chip wires are typically about an order of magnitude greater than the capacitances associated with on-chip signals, and the time needed for a MOS circuit to drive a lumped capacitance grows logarithmically with the capacitance [58](p.14).

The global instruction broadcast network in a SIMD computer is typically electrically larger and far more geometrically complicated than a microprocessor's external memory interface. Were the PE chips made using the microprocessors' VLSI implementation technique and the global instruction broadcast network driven as a lumped capacitance, then ρ_{b} would certainly be much larger than 2. For example, in a SIMD computer occupying a single 50cm × 50cm printed circuit board (PCB), each bit of broadcast instruction might present a lumped capacitance of around 2.5nF. Even using a very low-resistance driver of R_{on}=6Ω, the instruction bit rise time would be 33ns. Allowing 3 times the rise time for the bit interval, this corresponds to a maximum broadcast rate of 10MHz. Against a PE clock rate of 300MHz, this yields a ρ_{b} of 30.

In a generic SIMD computer, the PE clock rate equals the system clock rate, which is determined by the global instruction broadcast rate. One might expect the electrical design of a SIMD computer's global instruction broadcast network to be do better than driving the instruction bits as lumped capacitances, so as to provide instructions to the PEs at their highest execution rate. It is therefore surprising that there is in fact a significant disparity between maximum possible PE operation rates and system clock rates in existing SIMD computers. The global instruction broadcast rates of recent SIMD computers range from as high as 20MHz to as low as 8MHz, as indicated in Table 3.3. The clock rates indicated in Table 3.3 are significantly lower than the operation rates of circuits of similar logical complexity to the PE function units implemented using similar VLSI implementation techniques. A comparison of the operation rates for the chips listed in Table 3.4 against those of the computers listed in Table 3.3 suggests that PEs in existing SIMD computers execute instructions at rates about 10 times lower than they otherwise could. In other words, ρ_{b} appears to be around 10 in existing SIMD computers.

High values of ρ_{b} for existing SIMD computers suggest that board- and chassis-level engineering factors prevent high-rate global instruction broadcast. An alternative to driving each instruction bit as a single lumped capacitance is to organize the global instruction broadcast network as a clocked fanout tree whose nodes drive smaller lumped capacitances. In such a network, a high-rate clock regulates the advance of instructions from the system controller through a tree of clocked registers to the PE chips. The instruction fans out by a modest factor at each successive level of the tree. Assuming that the system clock is distributed with sufficiently low skew, instructions progress at the rate of inter-chip signaling. Of course, inter-chip communication in this network incurs module and

System		Clock-rate	Technology	Year	Single-cycle PE Op	PE FU Width
GF11	[6]	20MHz	mixed	1985	floating-point multiply	32b
CM-2	[16]	8MHz	2.0μm CMOS	1987	$\frac{1}{3}$ add	1b
SLAP	[28]	8MHz	2.0μm CMOS	1988	2-b multiply step	16b
Blitzen	[36]	20MHz	1.0μm CMOS	1988	add	1b
MP-1	[62]	14MHz	1.6μm CMOS	1990	add	4b
CM-200	[16, 17]	10MHz	1.5μm CMOS	1991	$\frac{1}{3}$ add	1b

Table 3.3: SIMD Computer Speeds

Chip		Clock-rate	Technology	Year	Single-cycle Op	Operand Width
Divider	[86]	100MHz	2.0μm CMOS	1987	1-b divide step	54b
Yuan	[89]	250MHz	3.0μm CMOS	1989	count	4b
DataWave	[67]	125MHz	0.8μm CMOS	1990	2-b multiply step	12b
Yuan	[90]	700MHz	2.0μm CMOS	1991	count	8b
Alpha	[21]	200MHz	0.375μm CMOS	1992	4-b multiply step	64b
Hitachi	[63]	250MHz	0.3μm BiCMOS	1992	add	32b

Table 3.4: Chip Speeds

board crossings, and the need for all levels in the distribution tree to be synchronized likely means that $\rho_b > 2$. The fanout tree achieves fast instruction broadcast at the cost of high latency, which price is paid as PE idle time at each global data-dependent branch in a program. The greatest drawback of a clocked fanout tree is that it contains a large number of chips and wires, which resources could otherwise be used for PE chips themselves.

PCB traces typically have low resistance and may be modeled as distributed inductors and capacitors, such that when run over a ground plane they may be used as transmission lines. The design of a high-speed broadcast network that contains fewer chips than a clocked fanout tree would exploit the electrical properties of transmission lines. The time for a voltage step to propagate through a terminated transmission line is given as the time of flight delay, t_f:

$$t_f = \frac{l}{v} \tag{3.6}$$

where l is the length of the line and v is the propagation velocity. v is typically on the order of the speed of light, so transmission lines minimize signaling delays between chips.

More important with respect to high-rate instruction broadcast is the fact that a unit voltage step does not deteriorate as it propagates through a uniform lossless transmission line. The rate at which signals are delivered through a transmission line is determined not by the propagation delay for a single voltage step, but rather by the time interval needed for a receiver to be able to distinguish between successive voltage steps. Where the time of flight (t_f) through a transmission line is significantly greater than the driver's rise time (t_r), a transmission line makes high-rate signaling possible by allowing multiple voltage steps to be in transit through the line at any given time.

The global instruction broadcast network is driven at one point (at the system controller) and is received by every PE chip. Unlike typical one-to-one inter-chip signal paths or many-to-many busses (where many is small), the broadcast network is designed for one-to-many communication where many is large. Among well-known electrical engineering problems, the design of the broadcast network is closest to the design of a clock network.

The SIMD computer's PEs operate in lock-step, often exchanging information. To minimize the time required for inter-PE communication, and to avoid the myriad design issues related to synchronizing independent signals, assume that the clocks of the PEs are kept in-phase. One way to keep the PEs in-phase is to distribute the system clock with minimum skew. Low-skew clock distribution has been widely studied. Ordinarily, low-skew clock distribution demands careful matching of signal path lengths in the distribution network [1](Chap.8). Some clock distribution techniques exploit the inherent regularity of a clock signal. One such technique is to distribute the clock as a standing wave, the phase of which is constant within regions bounded in diameter by half the wavelength of the standing wave [13].

The physical device characteristic mis-matches that give rise to skew in a clock distribution network also cause skew through the global broadcast instruction network. The time allowed for each broadcast instruction bit must allow for variations in the time for the bit to arrive at each of the PE chips. Although this skew problem resembles that arising for the system clock, it is more difficult to solve for two reasons: First, a clock signal is simply repetitive, whereas the broadcast instruction is not. Second, a clock is typically only one signal, whereas an instruction contains many bits. This multiplicity exacerbates the skew problem for broadcast instructions by increasing the number of signals whose arrival times need to be matched.

Figure 3.12 illustrates the design of a broadcast network using transmission lines. If the transmission lines in Figure 3.12 are ideal, properly terminated, and of equal lengths, then the rate of instruction broadcast is determined by the rise time t_r of the driver. (A driver is indicated at point B in the Figure 3.12.) The time allowed per bit might be 3 times t_r, to ensure meeting the set-up and hold constraints on the PE chip latch that receives the instruction bit. (A latch is indicated at point D in Figure 3.12.) Using a high-speed ECL driver, such as MC100E111 [59] with t_r=400ps, the bit interval would be 1.2ns, for a broadcast rate of 833MHz.

The network sketched in Figure 3.12 would therefore provide instructions to the PEs as fast as the PEs can execute them. However, that network is not practical, because it contains a transmission line carrying each bit of the instruction directly from the system controller to each PE chip. In a SIMD computer with a 32-b instruction that is delivered to each of 100 PE chips, the network sketched in Figure 3.12 requires driving thousands of transmission lines from the system controller. The number of driver chips needed, and the density of wiring near the system controller, make such a large number of direct lines prohibitive.

A more realistic scenario is constructed by considering a specific hypothetical SIMD computer with the following characteristics:

- The broadcast instruction is 32 bits wide.
- The computer contains 4800 PEs.
- The PEs are packed 4 to a chip, so there are 1200 PE chips.
- Each PE chip is mounted along with local external memory on a 3cm × 6cm multi-chip module (MCM).
- The MCM and its board-level wiring take up a 4cm × 8cm region on a PCB containing PEs (the PE board), so the PE board contains 6 rows with 10 PE chips per row.
- The computer fits in a single rack of 50cm × 50cm PCBs.
- 20 PE boards make up the entire computer, along with a PCB containing the system controller.

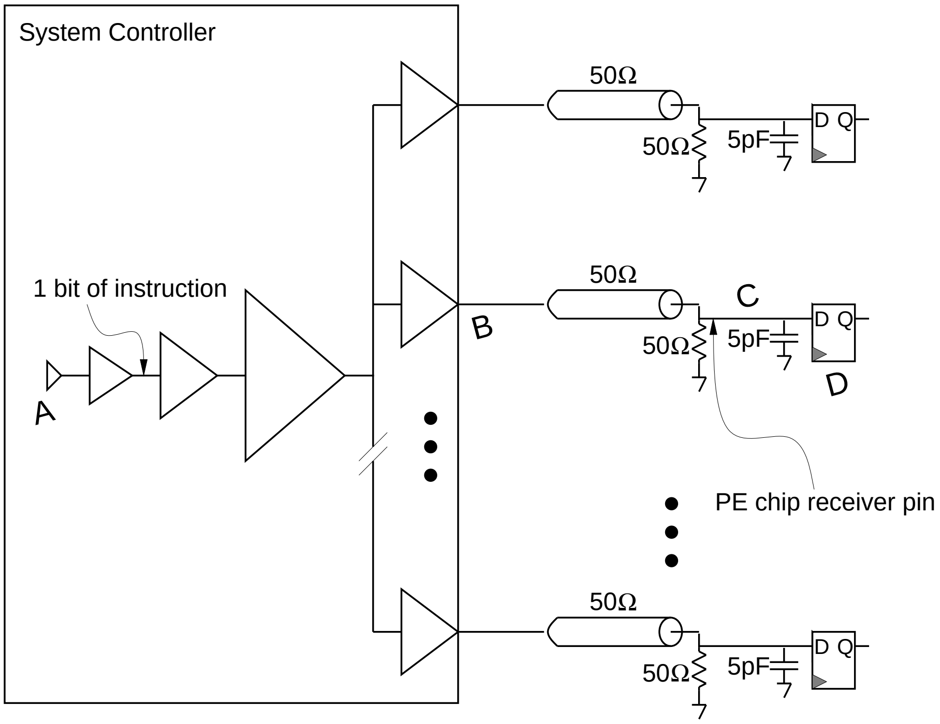


Figure 3.12: One way to implement a very fast global instruction broadcast network uses a set of transmission lines leading directly from the system controller to each PE chip.

- The 21 boards fit in a rack that is 65cm wide, with the system controller occupying the middle slot.

Figure 3.13 shows the rack layout. 32 bits driven to each of 20 PE boards means that 640 transmission lines are driven from the system controller board. This is still a very large number of wires to route from the system controller, but it may not be prohibitively large.

Figure 3.14 shows the layout of the PE board, and Figure 3.14 shows how the broadcast instruction might be delivered along one row of the PE board.

A broadcast network is shown in Figure 3.16, wherein each instruction bit is driven only once to each PE board.

The system controller generates instructions which are fanned out and buffered on the system controller board. The instructions are delivered to the PE boards via transmission lines. The column of fanout buffers down the middle of the PE board receives the broadcast instruction, fans it out, and drives 12 tapped transmission lines half the length of the board, 2 such lines for each of the 6 rows shown in Figure 3.14.

As shown in Figure 3.15, a buffer on the PE board drives a transmission line "bus" that is tapped by number of PE chips. This arrangement conserves the number of driver chips in the network, and it further reduces the overall wiring complexity as compared to that of the network in Figure 3.12.

Allowing $3t_r$ per bit for set-up and hold times at the instruction latches in the PE chips, the minimum interval of the broadcast instruction is given as

$$\text{minimum bit interval} = 3t_r + \text{worst-case total skew} \tag{3.7}$$

The contributions to skew are as follows:

$$\text{skew components} = \text{driver delay variations at system controller}$$

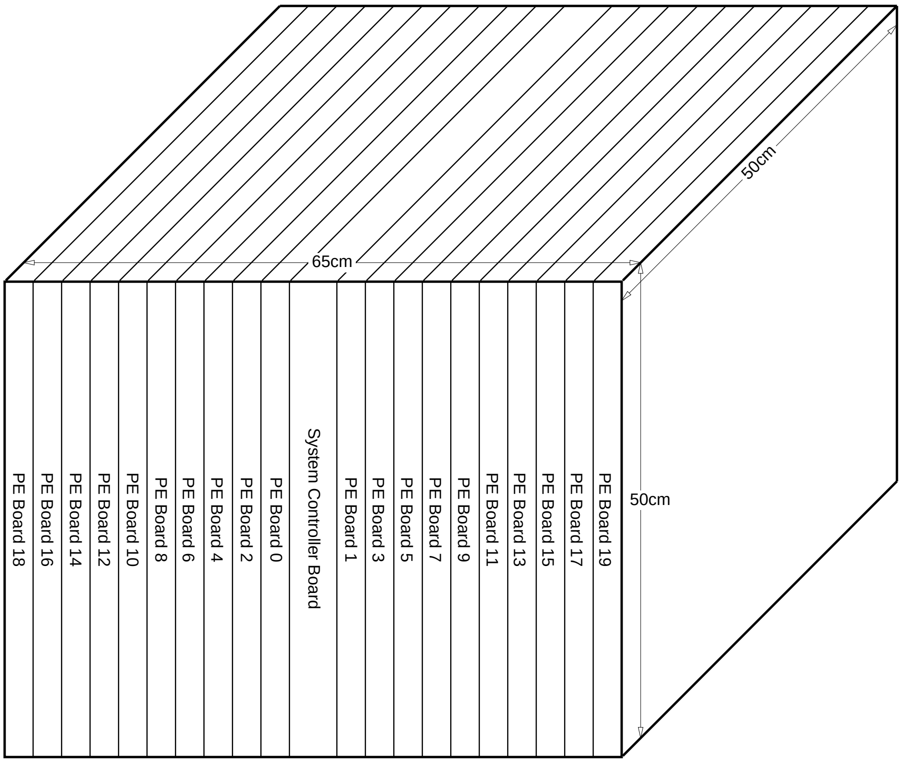


Figure 3.13: The layout of a rack containing a 4800-PE SIMD computer. The system controller is in the center of the rack, from where it broadcasts instructions to the PEs.

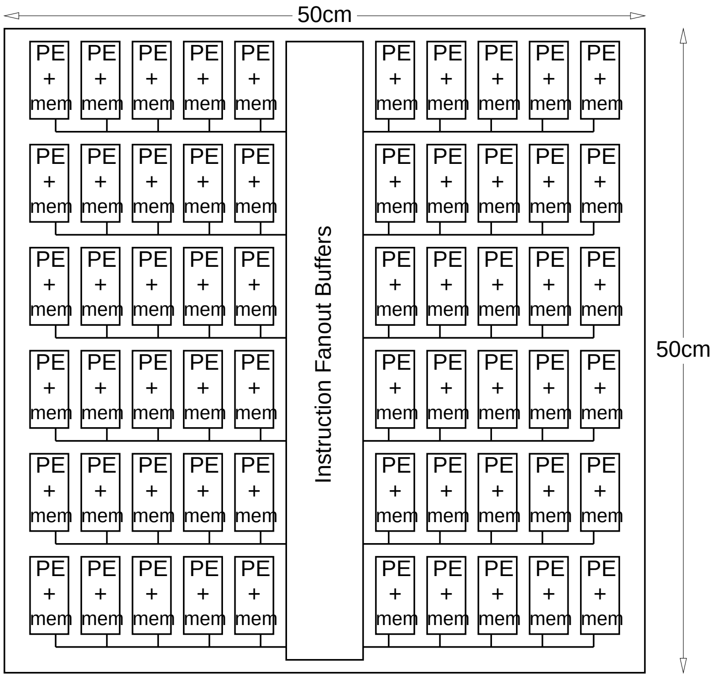


Figure 3.14: Layout of the PCB containing the PEs. The PE board has a set of low-skew fanout buffers in the center that drive the broadcast instructions to the PE chips on the board.

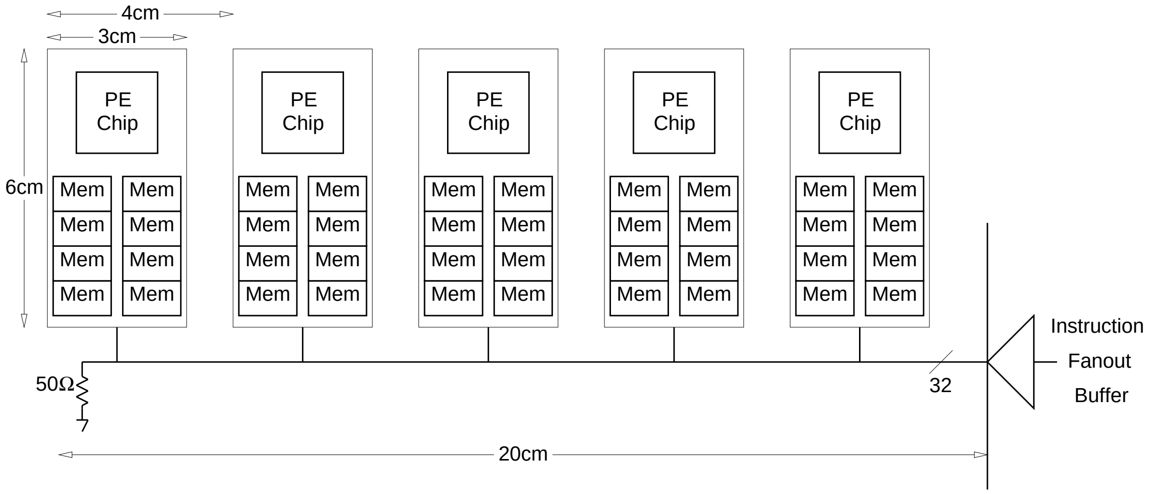


Figure 3.15: Detail of one row of the PE board. The instruction is driven through a tapped transmission line along each row of the board. The PE chips along each row tap a shared line.

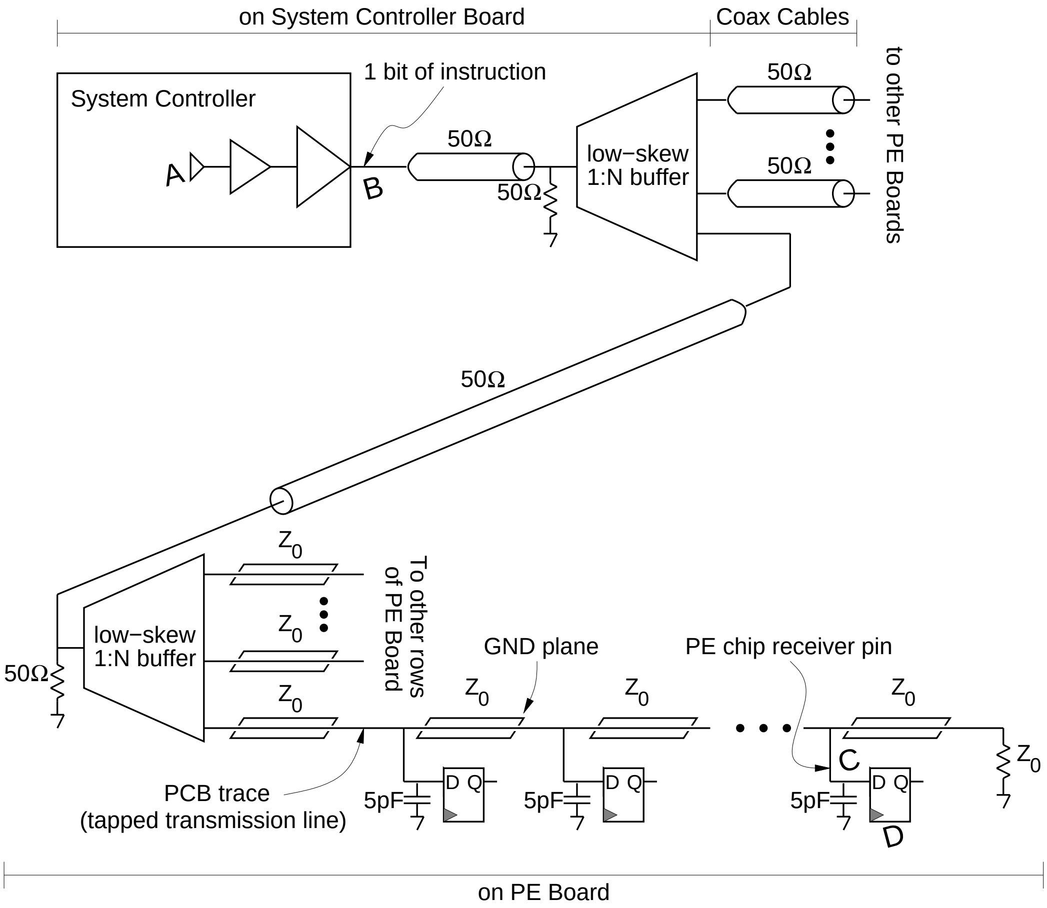


Figure 3.16: A more practical implementation of a fast global instruction broadcast network. Instructions generated on the system controller's board are distributed to the PE boards through coaxial cables where they are buffered and driven through tapped transmission lines.

$$
\begin{aligned}
&+ \text{coax cable length variations}\\
&+ \text{driver delay variations at PE board}\\
&+ \text{PCB trace length variations on PE board}\\
&+ \text{bit arrival time variations on PE board}\\
&+ \text{dispersion}\\
&+ \text{rise time variations at PE chip receiver stubs}
\end{aligned}
\qquad (3.8)
$$

The following paragraphs estimate the skew components:

The best readily available driver is the MC100E111 low-skew fanout buffer [59]. Although the rise time (t_r) of the MC100E111 is just 400ps, single-ended delays through the drivers on different chips vary by up to 400ps.

There will be some variation in the lengths of the coax cables leading from the system controller to the PE chips. The longest coax cable would need to be at least 82cm (half the system controller board length (25cm) plus half the length of the rack (32cm) plus half the PE board length (25cm)). Assuming that the coax runs vary in length by as much as 5%, or 4.1cm, the worst-case skew arises from a path length difference of 8.2cm. Signals propagate at about 20cm/ns in coax, so a length difference of 8cm corresponds to a skew of 440ps through the coax cables.

The low-skew fanout buffer on the PE board contributes another 400ps skew.

The instruction bit is received at one point on the PE board. From the point of reception, the instruction bit is fanned out and buffered within the middle column of the PE board shown in Figure 3.14. Ultimately, the instruction bit is driven horizontally from the center of the board through the tapped transmission line shown in Figure 3.15. The lengths of the PE board traces that route from the reception point through the buffer to the horizontal drive point vary by up to 25cm, half the length of a 50cm PCB. The propagation velocity through a PCB trace is about 15cm per ns [1](p.241). At that velocity, a 25cm variation in trace length yields a skew contribution of 1700ps.

The use of busses on the PE board, instead of point-to-point drivers supplying each PE chip directly with its own copy of the instruction, conserves driver pins as well as wires in the broadcast network. An unfortunate consequence of this economy is that the time for the broadcast bit to arrive at every PE chip tapping one such bus differs by as much as the time of flight along the bus. The bus is 25cm in length. The PE chip receiver pin at each tap adds to the capacitance of the transmission line. The capacitance of a PCB is typically about 1pF per cm. Adding 5pF per pin spaced at 4cm, the PE chips increase that capacitance by a factor of 2.2. The propagation velocity c through a transmission line is given as

$$v = \frac{1}{\sqrt{LC}}$$

where L and C are the total inductance and capacitance of the line. Multiplying the capacitance by 2.2 decreases v by a factor of 1.5, from 15cm per ns to 10cm per ns. Substituting l=25cm and v=10cm per ns into Equation 3.6 gives t_f along the tapped transmission line of 2500ps.

Unfortunately, PCB traces do not make ideal transmission lines. The trace widths are not perfectly uniform, dielectric thickness varies, and the traces have finite series resistance. These deviations from ideality give rise to dispersion, whereby the propagation velocities of signals are frequency dependent. A step input contains a range of spectral components, so the step input in fact degrades as it propagates through a PCB trace. Dispersion measures the frequency dependence of propagation delay through an imperfect transmission line. Dispersion tends to be greater at higher frequencies. Because of its path dependence, dispersion is a skew component that is added to the time per bit. Dispersion might contribute a skew of up to 50% of the signal rise time, or 200ps in this case.

The signal is not terminated inside the PE chips, so a single-ended instruction bit brought onto the PE chip represents a capacitive stub. (A PE chip pin is indicated at point C in Figure 3.16.) The stub capacitances may vary by as much as 2.5pF due to variations in tap geometries for the various bits of one instruction. Rise time of a lumped capacitance is $2.2RC$, so a difference of up to 2.5pF through 50Ω gives a time difference of up to 225ps.

Substituting the various skew estimates for the terms in Equation 3.8 yields the following totals:

$$\begin{aligned}\text{worst-case total skew} &= 400\text{ps} \\ &\quad + 440\text{ps} \\ &\quad + 400\text{ps} \\ &\quad + 1700\text{ps} \\ &\quad + 2500\text{ps} \\ &\quad + 200\text{ps} \\ &\quad + 225\text{ps} \\ &= 5865\text{ps} \qquad (3.9)\end{aligned}$$

Substituting t_r=400ps and the value from Equation 3.9 into Equation 3.7 yields the following estimate for the instruction broadcast interval:

$$\begin{aligned}\text{minimum bit interval} &= 1200\text{ps} + 5865\text{ps} \\ &\approx 7.0\text{ns} \qquad (3.10)\end{aligned}$$

At 7.0ns per bit, instructions are broadcast at about 142MHz. For a PE clock rate near 300MHz, this gives a ρ_b of about 2.

The broadcast rate estimates given above neglect reflections in the broadcast network. Parasitic impedances, such as are formed at corner turns in PCB traces or by imperfect connectors, cause reflections that erode signal quality and lower the maximum distribution rate.

Beyond the physical limitations discussed above, there are other practical considerations that tend to keep the global broadcast rate low.

One consideration is the chip area occupied by the broadcast network. Economy of chip area is the principal engineering motivation for SIMD computer architecture. Driver chips used in a fast global instruction broadcast network displace PE chips from the PE boards, thereby eroding the chip-area economy of SIMD computer architecture.

The speed of the resulting network is in any case limited in the rate at which instructions are provided by the system controller (indicated at point A in Figures 3.12 and 3.16). The system controller sequences instructions from memory, perhaps modifying loop-index-dependent literal fields before driving the instruction to the broadcast network. Broadcasting instructions at a very high rate requires the system controller to be able to supply them at that high rate. High-rate instruction distribution requires a fast system controller whose cost may be high. SIMD computer architecture leverages the optimized design of a PE chip through replication, but there is only one system controller. A high-cost system controller may not be desirable. If an inexpensive system controller were used, instructions might then be broadcast at a modest rate. For example, if the system controller were a commercial microprocessor, instructions might be broadcast over the microprocessor's I/O bus. Low-cost implementations of the system controller and global instruction broadcast network tend to yield large values of ρ_b.

Chips typically have limited numbers of signal pins. It is advantageous to allocate a least number of PE chip pins to receiving instructions, such that a greatest number are available for the other inter-chip communication requirements of the PEs. One way to conserve PE chip pins is to time-share the global instruction broadcast network receiver pins. Time-sharing the global instruction broadcast receiver pins increases ρ_b by a factor equal to the degree of time-sharing. For example, bit-serial distribution of a 32-bit instruction word increases ρ_b by a factor of 32, while freeing 31 PE chip pins for use in other multi-chip subsystems. Bit-serial instruction broadcast has the added benefit of reducing the complexity of the skew problem for broadcast instructions.

Another consideration is power consumption. Terminated transmission lines and high-current bipolar drivers typically dissipate large amounts of power. In application contexts with small power budgets, fast global instruction broadcast using transmission lines as illustrated above would not be feasible.

It might be nice to be able to upgrade an existing SIMD computer by making its PE chips faster using newer VLSI implementation technique. If the new PE chips have the same pinouts as the original ones, and if power supplies and cooling are adequate, then such an upgrade would occur simply by replacing the PE chips. As the boards would be unchanged, the resulting increase in ρ_b is proportional to the increase in PE clock rate. Even if ρ_b was about 1 in the original computer, it would be higher in the upgraded one.

Finally, the SIMD computer architecture might be scalable, such that a single PE chip design is intended to be used in a range of computers whose sizes match the varying size requirements of a range of problems. A fast global instruction broadcast network requires solving the board-level electrical design problems for each instance. Simpler, slower global instruction broadcast networks are not so constrained in the size of the network or in the geometry of the wires, and therefore lend themselves more readily to scaling of the computer.

Note that the use of transmission lines to distribute instructions introduces a form of pipelining in global instruction broadcast, because a number of instructions are in transit in the network at any one time. When a program specifies a branch whose outcome depends on intermediate results calculated in the PEs, the latency of the branch as measured in the number of instruction times is equal to the number of instructions in the broadcast pipeline. After broadcasting a branch instruction, the global instruction broadcast network fills up with (probably wasted) branch delay slot instructions. The number of instructions in transit through the broadcast network is given by the ratio of the total delay of the network to the bit interval. If the path length of an instruction bit is about 200cm, then the time to propagate through the network at 15cm per ns would be about 13ns, or more than 8 instruction times at 600MHz.

A similar global branch latency arises using I-cache. At a global branch, the execution of cached instructions effectively ceases, awaiting the arrival of the first instruction following the branch. If the target instruction sequence is in cache, execution resumes at the high PE clock rate. Otherwise, the needed instructions must be delivered to the PE chips at the low rate. Fast instruction broadcast is superior in this regard, because the cache store overhead is never incurred, and only the branch delay slot instructions are potentially wasted.

High-rate global instruction broadcast is attainable, although the highest broadcast rate is at least a small factor slower than the highest PE clock rate. Furthermore, fast instruction broadcast network components compete with PE chips for board real estate. I-cache is an architectural enhancement which, if effective and practicable, allows lower-cost system controller and board-level designs. There is a broad range of reasonable estimates for ρ_b. High estimates for the PE clock rate are extrapolated from existing microprocessors, whereas high estimates for global instruction broadcast rate are speculative and have yet to be demonstrated. Based on these estimates, it appears that ρ_b is as low as 2 and as high as 16. The mid-point of this range, ρ_b=8, is a conservative estimate for ρ_b in existing SIMD computers.

3.9 Clock Intervals and ρ-Sets

Instruction broadcast rate is lower than PE clock rate because the global instruction broadcast network wires are much longer and more heavily loaded than the typical capacitances driven inside the PE chip. This observation applies not only to global instruction broadcast, but to every MCS, because every MCS contains inter-chip wires and inter-chip wires typically present greater driving loads than intra-chip wires. The top operation rate for each MCS is determined by the VLSI implementation technique and by the geometries of inter-chip wires. Inter-chip wire geometries are in turn are determined by the MCS network topologies and by board-level wiring constraints. It is reasonable to assume that the top operation rate for each MCS lies somewhere between the PE clock rate and the instruction broadcast rate.

The PEs in an I-cached SIMD computer are clocked at their highest rates irrespective of the rate of global instruction broadcast. Similarly, the presence of I-cache means that instruction availability no longer prevents each MCS from operating at its top rate. Therefore, the model that is the basis for evaluation of I-cache should allow for the possibility that the top clock rates of the various MCSs differ.

Let t_{PE} denote the interval of the fastest clock within the PE chip. t_{PE} is determined by PE architecture and by VLSI implementation technique. For example, t_{PE} exceeds the time required to drive the PE busses and the time required for the FU to produce a single-step result. PE chips in SIMD computers tend to exhibit low circuit complexity, as for example do MP-1 [62], Blitzen [36], and SLAP [27]. t_{PE} tends to be low, significantly lower than the interval of the system clock that regulates global instruction broadcast. Large wire delays between chips in an MCS mean that its minimum clock interval is larger than t_{PE}.

Figure 3.17 shows a sketch of the simple delay model used to relate the intervals of the clocks regulating the various subsystems. The parameters of the model sketched in Figure 3.17 have the following interpretations:

- t_{X} represents the minimum propagation time through a component of MCS X resident on a chip remote from the PE chip. For example, t_{l} represents local external memory access time, while t_{c} represents packet forwarding time for router-based inter-PE communication. t_{X} is determined by functional complexity and by VLSI implementation technique.

- W_{X} is the delay of the wires connecting the PE chip to a remote integrated component of MCS X. W_{X} represents the delay of on-chip wire drivers lumped together with electrical propagation delays along the wires themselves. W_{X} depends on wire geometry, on capacitive loading, and on the on-resistance of the transistors driving the MCS's wires.

- ρ_{X} represents the factor by which the minimum interval of the clock regulating MCS X is greater than t_{PE}.

In the simple model sketched in Figure 3.17, wires are driven as lumped capacitances, with the notable exception of the global instruction broadcast network (discussed in Section 3.8). For simplicity, all MCS operation rates are modeled in a uniform manner here.

Assuming that t_{X} and t_{PE} are roughly equal, and that signaling through inter-chip wires overlaps with the operation of the remote circuits, Equation 3.11 gives ρ_{X}:

$$\rho_{\text{X}} \approx \frac{W_{\text{X}}}{t_{\text{PE}}} \tag{3.11}$$

Note that if t_{PE} is taken to be 1, then ρ_{X} represents the minimum interval of a clock regulating MCS X. A lower bound for ρ_{X} is obtained by comparing a lower bound W_{X} against an upper bound for t_{PE}.

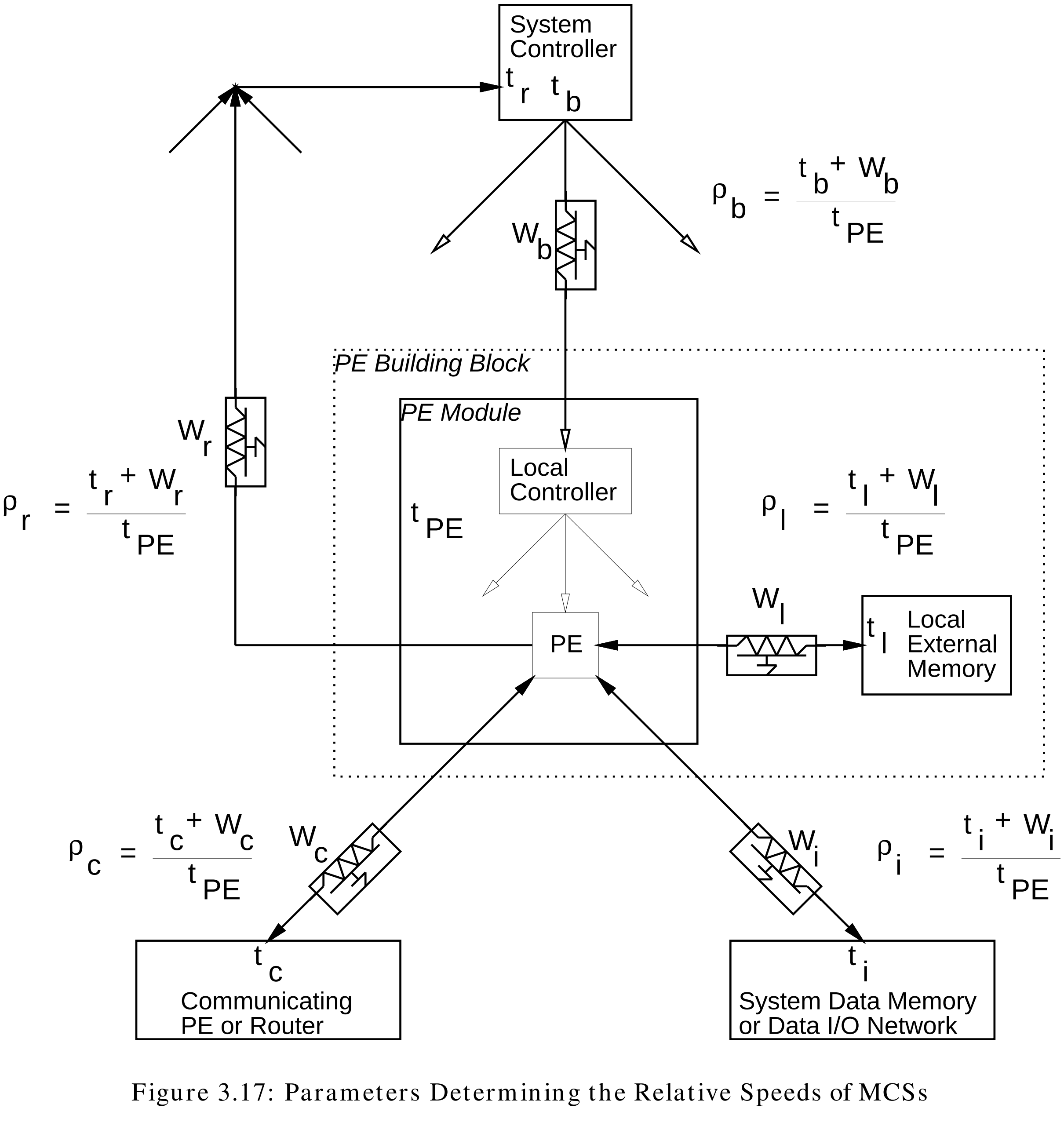


Figure 3.17: Parameters Determining the Relative Speeds of MCSs

An MOS circuit can drive a capacitive load in minimum time using an *exponential horn*, a series of inverters of geometrically increasing size, such that the delay is uniform (and minimum) at each successive stage. The ideal step-up ratio has been estimated to be e [58](p.13), while other estimates range as high as 10 [61](p.161). Where the step-up ratio is r and the (scale-independent) parasitic delay of an inverter is p, then the delay of one stage in the horn is $r + p$. The ideal step-up ratio r is determined by the VLSI process and is independent of the number of stages [77].

Assuming the driven signal originates from a minimum inverter, the number of stages is $\log_r \frac{C_L}{C_g}$, where C_g is the minimum inverter's gate capacitance. The minimum time t_{C_L} to drive a capacitive load C_L is then given by Equation 3.12:

$$\begin{aligned} t_{C_L} &= (r + p) \log_r \frac{C_L}{C_g} \\ &= (r + p) \frac{\ln \frac{C_L}{C_g}}{\ln r} \end{aligned} \tag{3.12}$$

Let C_X denote the capacitive load driven between chips in MCS X and C_{on} denote the worst-case intra-chip load capacitance determining t_{PE}. Then ρ_{X} is given by Equation 3.13:

$$\begin{aligned} \rho_{\text{X}} &\approx \frac{(r + p) \frac{\ln \frac{C_X}{C_g}}{\ln r}}{(r + p) \frac{\ln}{\frac{C_{\text{on}}}{C_g}} \ln r} \\ &= \frac{\ln \frac{C_X}{C_g}}{\ln \frac{C_{\text{on}}}{C_g}} \end{aligned} \tag{3.13}$$

Typical capacitances for printed-circuit-board-based (or *PCB-based*) technology, are as follows:

$$\begin{aligned} C_{\text{g}} &\approx 10\text{fF} \\ C_{\text{on}} &\leq 3\text{pF} \\ C_{\text{off}} &\geq 15\text{pF} \end{aligned}$$

Substituting these values into Equation 3.13 yields the rough lower bound in Equation 3.14:

$$\rho_{\text{X}} > 1.3 \tag{3.14}$$

When inter-chip wires are long or where inter-chip wiring networks are heavily loaded with multiple taps, ρ_{X} exceeds the lower bound given in Equation 3.14. However, the top clock rates for MCSs other than instruction broadcast are likely to be nearer to the PE clock rate than the system clock rate. Table 3.5 summarizes the delay model terms for each MCS.

I-cache speedup depends on ρ_{b}, but also on the relative clock rates of the

ρ_{b}, the factor by which the instruction broadcast interval exceeds t_{PE}, is a critical determiner of I-cache speedup. The benefit of instruction caching depends also on the other ρ_{X} values, which determine the times for operations using the various MCSs.

A collection of values for the MCS clock intervals comprises a *ρ-set*. A ρ-set is a set of five numbers of the form

$$\rho\text{-set}=\{\rho_{\text{b}}, \rho_{\text{r}}, \rho_{\text{i}}, \rho_{\text{c}}, \rho_{\text{l}}\}$$

Multi-chip Subsystem	Important inter-chip wires	Wire delay	Factor by which minimum clock interval exceeds t_{PE}
Global instr bdcast	Global broadcast network	W_{b}	ρ_{b}
Response	Response network	W_{r}	ρ_{r}
Input/output	System data I/O network	W_{i}	ρ_{i}
Inter-PE comm	Inter-PE comm network	W_{c}	ρ_{c}
Local external memory	Intra-building-block connection	W_{l}	ρ_{l}

Table 3.5: Summary of delay model terms. $\rho_{\mathrm{X}} \approx \frac{W_{\mathrm{X}}}{t_{\mathrm{PE}}}$.

Due to the limited size of a manufacturable chip, a high-PE-count SIMD computer incorporates more than one PE chip. In fact, a high-PE-count SIMD computer likely encompasses an integration hierarchy, containing MCMs, PCBs, racks, chassis, and so on. Significant delays are incurred for signaling across chip boundaries, due to the relatively large energies required for inter-chip signaling. Similar, although less marked, penalties accrue crossing other boundaries in an integration hierarchy. W_{X} depends in part on the level of integration hierarchy boundaries crossed by the wires in MCS X. The following enumeration considers the likely values of the various W_{X}:

1. The global instruction broadcast network connects a single source (the system controller) to all PE chips in the computer. The global instruction broadcast network wires are long, geometrically complex, and electrically heavily loaded, so W_{b} is large. As pointed out in Section 3.8, for this reason the broadcast network wires may be driven as transmission lines.

2. The response network is system-wide in extent, aggregating fan-in from each PE. W_{r} is therefore likely to be large.

3. The system I/O network structure can vary over a wide range. Depending on the system, this network may contain long wires and W_{i} may be large. In a specialized SIMD computer, for example one used for CCD sensor-embedded image processing, the I/O network may incorporate relatively short wires and so W_{i} may be low.

4. To the extent that the wires in the inter-PE communication network are long, W_{c} is large. Networks of high topological dimension necessarily contain long wires [79].

   Regular meshes contain point-to-point wires. At least one wire in a mesh inter-PE network must cross a boundary at the coarsest level of integration in the system's hierarchy. By folding such a network (as described in [19](p.154)), the wires become as short as possible for wires having to cross integration hierarchy boundaries. In this case, W_{c} approaches the minimum delay exhibited by a wire crossing integration hierarchy boundaries in neighbor inter-PE communication networks. Fast inter-PE communication on regular grids is illustrated for multiprocessors in Mosaic [71].

5. The local external memory array is packaged alongside the PE chip within a PE building block. The building block is a physically compact system component. If the building block is implemented using the fastest available inter-chip technology, then local external memory wire delay W_{l} approaches the minimum possible for inter-chip wiring.

As VLSI implementation technique continues to improve, inter-chip wire delays do not decrease as fast as intra-chip circuit speeds increase. $\frac{W_\mathrm{X}}{t_\mathrm{PE}}$ tends to increase over time, so the values in a ρ-set tend to increase over time.

A generic SIMD computer uses a single clock to regulate all subsystems. When $W_\mathrm{b} > t_\mathrm{PE}$, PEs in generic SIMD computers are under-utilized. Similarly, when $W_\mathrm{b} > W_\mathrm{X}$ for some MCS X in a generic SIMD computer, that MCS is under-utilized.

3.10 Alternatives for Maximum-rate Instruction Delivery

When the highest PE clock rate exceeds the rate of global instruction broadcast, instruction delivery becomes a bottleneck in generic SIMD computations. What options are available to the architect to overcome this limitation?

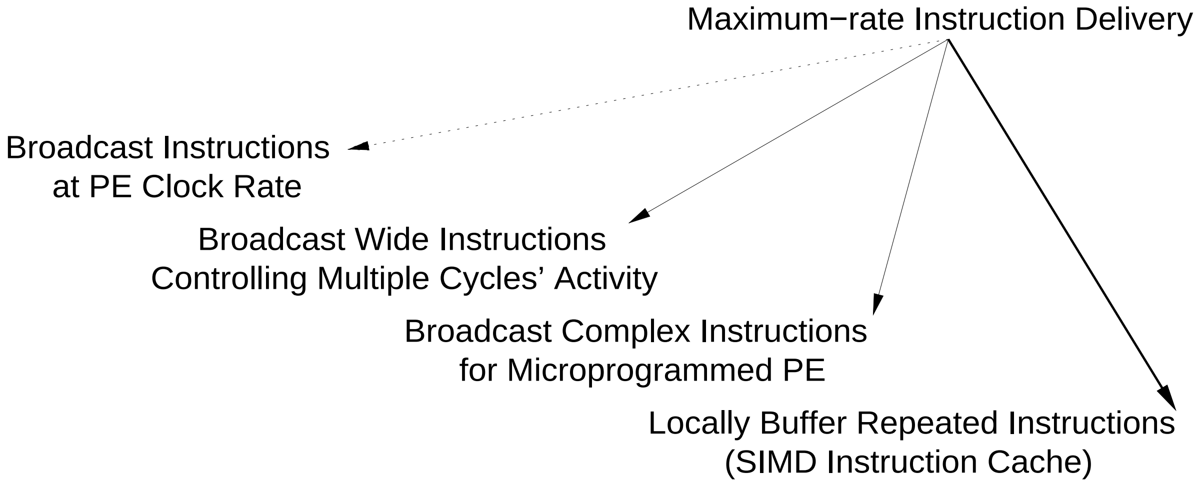


Figure 3.18: Alternatives for Maximum-rate Instruction Delivery

Figure 3.18 illustrates a range of possible options for delivering instructions to the PEs at the maximum rate. The dotted line pointing to the fast global instruction broadcast option indicates that the option is not always available. The remaining options include PE microprogramming, wide-instruction broadcast, and I-cache.

3.10.1 PE Microprogramming

The inherent chip-area advantage of SIMD computers arises from consolidating replicated program control that is redundant in some cases. The SIMD computer designer typically attempts to minimize instruction decoding logic within the PE chip, because this logic is redundantly replicated with each PE chip. Some decoding is sometimes unavoidable, for example to conserve the number of PE chip pins used for receiving broadcast instructions.

One way to keep the PE chip supplied with instructions for each of the multiple PE chip cycles following receipt of each broadcast instruction would be to globally broadcast "complex" instructions to the PEs. The PE chip local controller in this case would contain a microprogram sequencer. Each globally broadcast instruction would dispatch a multi-step microprogram that is sequenced within the PE chip.

Many aspects of PE microprogramming are architecturally undesirable. The main drawback of PE microprogramming applies also in microprocessors, namely that microprogramming commits considerable chip area to a mechanism capable of sequencing a fixed complex instruction set, each member of which may or may not be appropriate for the task at hand [65, 38]. Chip area used for program control that could otherwise be performed off-line is better used for FU and registers where maximum throughput is the design objective. Chip area is especially precious in high-PE-count computers, wherein large numbers of replications raise the stakes for efficient use of PE chip area. An additional unfortunate consequence of PE microprogramming is a degree of inflexibility that further restricts the class of problems for which the SIMD computer is appropriate.

3.10.2 Wide-Instruction Broadcast

Another way to provide a new PE instruction on each PE clock cycle would be to globally broadcast groups of instructions in parallel. The global instruction broadcast network would deliver many instructions to the PE chip in parallel on each system clock cycle. The PE chip's local controller would select a sequence of individual PE instructions from each globally broadcast group.

Broadcasting multiple instructions per clock cycle is tantamount to broadcasting one wide instruction. Such an approach to instruction delivery is infeasible because of the demand it places on PE chip pins. There are not likely to be enough pins on an affordable manufacturable PE chip to allow even 2 or 3 instructions to be received at once, let alone as many as 8 or more.

3.10.3 SIMD Instruction Cache

A SIMD instruction cache is an explicitly managed instruction buffer within the PE chip. I-cache added to the PE chip local controller comprises instruction memory and means of accessing it. Repeated sequences of instructions, such as those appearing in loop bodies, are stored on-chip in the I-cache for subsequent execution at the relatively high rate attainable within the confines of the PE chip.

Compared to the alternative of PE microprogramming, I-cache is a flexible means of defining “complex” instructions as sequences of primitive instructions sequenced at the highest possible rate.

I-cache appears to be either more practical or more affordable than the alternative techniques for overcoming the instruction delivery rate bottleneck that arises when $\rho_{\mathrm{b}} > 1$.

Chapter 4

I-Cached SIMD Computer Design

Figure 4.1 illustrates a local controller for the PE chip of an I-cached SIMD computer. Whereas the generic SIMD local controller (illustrated in Figure 3.3) is extremely simple, I-cache introduces some new design complexity.

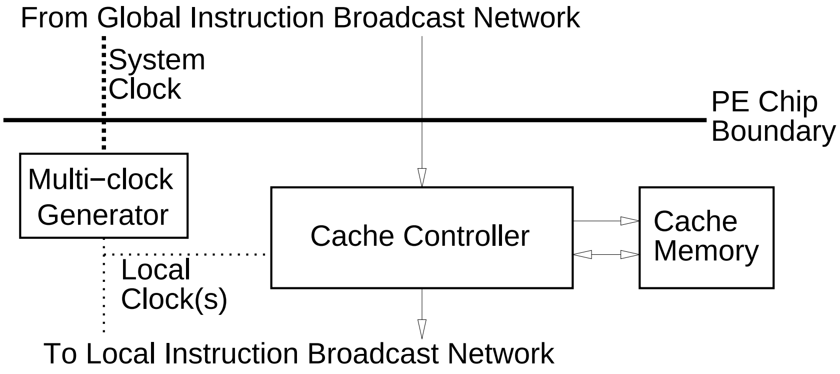


Figure 4.1: Local Controller with I-cache

The *multi-clock generator* shown in Figure 4.1 provides all of the required clocks within the PE chip. The fastest clock output by the multi-clock generator is the PE clock, which is ρ_b times faster than the system clock. The elements of a SIMD computer operate in lock-step, so all PE chip clocks are synchronized to the system clock.

Because there are multiple clock rates, and because the number of PE clock cycles required by an MCS to perform an operation depends in some cases on the particular operation performed, an MCS operation may conclude on an arbitrary one of the multiple PE clock cycles which occur between successive system clock cycles. Therefore, the format of globally broadcast instructions is augmented for an I-cached SIMD computer so that a globally broadcast instruction may specify the index of the PE clock cycle on which the MCS operation specified by that instruction completes.

The *cache controller* shown in Figure 4.1 outputs a new instruction for local broadcast within the PE chip on every cycle of the PE clock. When a new globally broadcast instruction arrives at the PE chip, the cache controller may provide a copy of that instruction within the chip. The cache controller also delays instructions terminating high-latency MCS operations until the PE clock cycle specified by the most recently globally broadcast instruction.

The cache controller also manages the *cache memory*. Under the direction of globally broadcast instructions, the cache controller stores repeated instruction sequences, or *cache blocks*, in cache memory and subsequently sequences them within the PE chip. With I-cache, *cache-control instructions* are added to the generic SIMD computer's repertoire of globally broadcastable instructions. A variety of I-cache designs are possible, each associated with a particular set of cache-control instructions.

I-cache speedup depends on a variety of factors including the set of functions implemented in the cache controller, the number of instructions that can be stored in cache memory, the programs being executed, and the means by which the I-cache is controlled in the broadcast instruction stream.

This chapter presents design considerations for the components shown in Figure 4.1. A family of I-cache variants is presented, and the chip area occupied by an I-cache is estimated. These examples quantify the displacement of payload from the PE chip ensuing from I-cache. Simplified example programs are used to illustrate the interactions of properties of programs and cache controller functions for members of the family of variants. Finally, issues in the programming problem of static I-cache management are presented.

4.1 I-cache Design Elements

Several detailed implementations of the local controller in an I-cached SIMD computer are possible. The design of the local controller determines its chip area. Also, the local controller design relative to the requirements of the program controlling a given computation determines what fraction of the factor of ρ_b maximum execution-rate increase is realized for the computation. This section enumerates the design elements, identifying those of greatest concern to the I-cached SIMD computer designer.

4.1.1 Multi-clock Generator

The maximum operation rate of each subsystem is determined by the VLSI implementation technique and by the topologies of the subsystem's inter-chip wires. If the computer is synchronous and each subsystem has a unique maximum operation rate, then regulating each subsystem at its maximum rate requires a unique clock per subsystem.

One way to distribute clocks at the various rates is to broadcast them globally. In a high-PE-count SIMD computer, global broadcast of high-rate clocks would be performed using transmission lines [1](Chap.8). The key design constraint for the resulting clock distribution network is the matching of path lengths and impedances, to minimize skew and reflectance. Such a clock distribution network forms a clock pipeline, wherein multiple clock events are propagating at any given time. Pipelined clocking is commonly used to regulate synchronous PE arrays [24](Chap.3). Pipelined clocking cannot provide a clock whose interval is less than the minimum inter-chip signaling interval. Another potential problem with pipelined clocking is the high chip-area cost of a clock distribution network containing large numbers of chips in a fanout tree.

An alternative means of obtaining the required clocks is to generate them within the PE chip. One widely practiced means of generating on-chip clocks uses a voltage-controlled oscillator (VCO) that is synchronized with other chips using a phase-locked loop (PLL) [12]. PLL-based clock generators are increasingly commonly in microprocessors, wherein the on-chip clock rate tends to be a multiple of the off-chip reference clock rate [5, 88]. PLLs have been described for use in synchronization among MIMD PEs [64](Sec.3.3.1). PLLs have also been used for clock skew elimination in SIMD computers [44, 16], although not for multiplying the system clock rate to obtain a higher PE clock rate.

The multi-clock generator is a mundane architectural element, because PLL-based clock generators have long been used in computers, as demonstrated in [46]. What is important about the multi-clock generator is the chip area that it occupies. Although VCOs tend not to scale with VLSI implementation technique as readily as does switching logic, they are practically useful for modern microprocessors [5].

On-chip generation of a high-rate PE clock that is phase-locked to the system clock does not eliminate the low-skew requirement on system clock distribution. However, it less difficult to distribute one clock signal with low skew than it is to distribute many.

The multi-clock generator design may introduce artificial constraints among the various clock rates, as does the design described in Appendix A. The multi-clock generator described in Appendix A provides a set of free-running clocks whose rates are constrained to be integer multiples of the PE clock rate and integer sub-multiples of the system clock rate. An alternative multi-clock generator might allow a clock to be stopped when the subsystem it regulates is idle, or similarly allow a clock's phase to be varied dynamically to minimize the delay in starting an operation on an otherwise idle subsystem. Inevitably, the design of a multi-clock generator encompasses meeting several interacting system timing constraints. [69] provides an excellent overview of these issues.

4.1.2 Cache Memory

The cache memory may have one or more ports. With a two-ported cache memory, one port is used to provide instructions already stored while the other port is used concurrently to pre-store instructions that will be needed subsequently.

Assuming that the best available memory cell with the required number of ports is used, the only cache memory parameters of interest to the I-cached SIMD computer designer are instruction word width (in bits) and cache size (in instructions), which together determine the number of cache memory cells and thus the chip area occupied by the memory.

4.1.3 Cache Management

The decisions as to which cache blocks to place where in the cache, as well as when to put them there and when to activate them, are all explicit in the global instruction broadcast stream. For simple programs, good choices can be made statically, in advance of running the computation. Programs whose loop structure is complex may require these decisions to be made in the system controller during the course of a computation, in which case the system controller requires a potentially complex cache-management mechanism. Section 4.5 discusses the management problem in detail, illustrating good solutions for simple I-cache variants.

4.1.4 Cache-control Protocol

The globally broadcast instructions in an I-cached SIMD computation include cache-control instructions in addition to the usual PE instructions. The cache-control instructions follow a *cache-control protocol* to store cache blocks and to activate them. Each I-cache variant specifies a cache-control protocol.

4.1.5 Cache Controller

As indicated in Figure 4.1, the cache controller is interposed between the global instruction broadcast network and the local instruction broadcast network within the PE chip. The cache controller selects the source of the instruction driven onto the local broadcast network on every PE chip clock cycle. The cache controller also manages the control inputs to the cache memory, and so contains a program counter providing a cache memory address.

The cache controller design and the concomitant cache-control protocol are the principal discriminator among I-cache variants.

There are many possible ways to denote the cache locations occupied by a cache block. For example, a cache block may be delimited by markers placed in the cache memory; alternatively, a cache block may be delimited by a parameter supplied upon its activation. Loops may or may not be unrolled when cached, subject to the details of a given cache design. A cache block does not necessarily correspond to an entire loop or subroutine body appearing in a program; some I-cache variants profitably cache subsequences of program bodies. A particular I-cache variant may allow multiple entries or multiple exits for a given cache block to facilitate a compact representation.

A globally broadcast cache-control instruction alerts the local controller to begin storing a cache block at a specified cache address. This instruction may also specify the length of that cache block, or the end may be indicated by a cache-control instruction transmitted at the end of the cache block.

A cache block is activated with a call specifying the parameters required for its execution, possibly including initial and final cache addresses and iteration count. Some I-cache variants provide mechanisms allowing cache blocks to activate one another, or to *nest*, in cache with varying degrees of generality.

4.2 A Family of Single-Port I-cache Variants

A cache design is characterized by answers to each of the following five questions:

- Does cache memory have more than one port?
- What is the maximum number of instructions that can be stored in cache memory?
- Can more than one block be stored in cache memory at a given time?
- Can the cache controller independently iterate cache blocks?
- Do cache blocks nest? In other words, can cache blocks activate one another?

The I-cache parameters imply the existence of several distinct classes of cache design. There are fewer than 16 classes, because some of the parameters are not mutually independent. For example, it is not possible for cache blocks to nest in a cache that can contain just one block.

The last three questions in the list apply only to the design of the cache controller. To illustrate the range of possible cache designs, consider the *F-family* of single-port I-cache variants.

An F-family cache memory has one port, so concurrently pre-storing cache blocks is not possible with F-family caches. The ends of cache blocks are delimited with sentinels, so that the length of a cache block is not specified in the block's activation.

A member of the family is designated F_i. The six family members vary in their cache controller characteristics, as enumerated below:

F_0 is a "one-block, one-shot" cache. F_0 is the simplest F-family cache. F_0 is capable of containing only a single cache block at any given time and of executing single passes through a cache block. A cache-control instruction activating an F_0 cache block supplies no parameters, because there is only one possible starting address, the ending address is delimited explicitly in cache memory, and the iteration count is always 1.

F_1 is a "multi-block, one-shot" cache. F_1 is similar to F_0 in that it executes only single passes through cache blocks. However, F_1 is not as simple as F_0, because F_1 is capable of containing more than one cache block at once. The questions relating to where to place each cache block are germane for an F_1 cache, giving rise to the myriad of replacement algorithm issues that have been studied in the contexts of ordinary caches and of virtual memory management. A cache-control instruction activating an F_1 cache block supplies a single parameter specifying the starting address of the cache block.

F_2 is a "one-block, multi-shot" cache. F_2 is similar to F_0 in that it contains only a single cache block. However, F_2 is not so simple as F_0, because F_2 is capable of sequencing through a cache block multiple times in response to a single activation. A cache-control instruction activating an F_2 cache block supplies a single parameter specifying a number of iterations of the cache block. F_2 does not require the ability to place cache-control instructions in cache.

F_3 is a "multi-block, multi-shot" cache. F_3 can contain multiple cache blocks, any of which can be iterated. A cache-control instruction activating an F_3 cache block supplies two parameters, the first specifying the starting address and the second specifying a number of iterations of the cache block.

F_5 is a "multi-nestable-block, one-shot"cache. F_5 is similar to F_1, in that it contains multiple blocks that are executed singly. However, F_5 has the additional characteristic that cache blocks may activate single iterations of one another. An F_5 cache requires that cache-control instructions may be placed in cache memory.

F_7 is the most complex member of the F-family. An F_7 cache may contain multiple blocks that may activate one another for multiple iterations. An F_7 local controller contains a scaled-down replica of the system controller's program-control components. Since an entire program could be stored in an F_7 cache, an PE chip with an F_7 cache becomes a mini-SIMD computer in its own right. If the individual local controllers were allowed to progress through different paths through the program in cache, an F_7-enhanced SIMD computer becomes a multi-SIMD (or *MSIMD*) computer [10].

Detailed designs for F_0 and F_2 I-cache variants are given in Appendix A.

4.3 I-Cache Chip-Area Estimates

In order to for I-cache to yield significant speedup for real problems, it must not displace too much of the payload from the PE chip. γ expresses the fraction of chip area taken up by I-cache inside the PE chip, so γ expresses the PE chip payload reduction due to I-cache. This section relates γ to the PE chip payload estimates derived in Section 3.7. The lowest values of γ are made possible by the most advanced VLSI implementation technique. chip-area estimates for F_0 and F_2 caches, two of the simplest members of the F-family, indicate that F_0 and F_2 caches are feasible for current VLSI implementation technique.

The local controller and MCS interfaces compete with PEs for interior area I even in a generic SIMD computer's PE chip. I-cache of the local controller further reduces the payload, assuming that physical dimensions H and W and resolution parameter λ are fixed. The extent to which the payload of the PE chip is reduced by I-cache depends on the chip area occupied by the multi-clock generator, the cache controller, and the cache memory. The cache controller's chip area depends on on the set of functions performed by the I-cache variant, while the cache memory's chip area depends on the width (in bits) of an instruction word and the maximum number of instructions that the memory contains. Reducing the payload in a PE chip means reducing the number of PE bit slices in the chip and/or reducing one or more of the PE architecture parameters (including datapath width, FU circuit complexity, and number of registers).

Π denotes the payload of a PE chip. Estimates for three existing PE chips and one hypothetical PE chip are summarized in Table 3.2. Let Π_g denote the payload of a generic SIMD PE chip, and let Π_c denote the payload of the chip when I-cache is added to its local controller. I-cache forces the local controller to expand by some chip area δ. Let A_g represent the chip area of the interior of that generic PE chip occupied by the local controller and by MCS interfaces, and let A_c be the non-PE chip area in the chip with I-cache. Then A_c is given in Equation 4.1:

$$A_c = A_g + \delta \tag{4.1}$$

The payloads of the PE chip without and with I-cache are given in Equations 4.2 and 4.3:

$$\begin{aligned} \Pi_g &= I - A_g && (4.2) \\ \Pi_c &= I - A_c && (4.3) \\ &= I - A_g - \delta && (4.4) \\ &= \Pi_g - \delta && (4.5) \end{aligned}$$

The increased chip area occupied by local controller as a fraction of the generic PE chip payload is the marginal chip area occupied by I-cache. γ represents this marginal decrease in payload, as shown in Equation 4.6:

$$\gamma = \frac{\delta}{\Pi_g} \tag{4.6}$$

The ratio of payload with I-cache to payload without I-cache is given in Equation 4.7:

$$\begin{aligned} \frac{\Pi_c}{\Pi_g} &= \frac{\Pi_g - \delta}{\Pi_g} && (4.7) \\ &= 1 - \gamma && (4.8) \end{aligned}$$

Because $\delta > 0$, $\gamma > 0$ and Π_c is strictly less than Π_g. It remains to consider the specific sizes of the various local controller components needed for I-cache.

Existing on-chip clock generators serve as guides in estimating the chip area of the multi-clock generator. A clock generator has been fabricated entirely within CMOS chips [88]. The oscillator dominating the clock generator's chip area runs at up to 220 MHz and is phase-locked to a lower-rate external clock. The entire clock generator occupies .31mm^2 in a λ=0.4μ process, or roughly 1.9×10^6 sites. Although the analog components inhibit scaling clock generator chip area as readily as register or FU chip area, the area of this existing clock generator is useful for approximating the area of the multi-clock generator needed for I-cache. Phase-locking is not the only way to generate fast clocks; other means to achieve high-rate timing references include the use of synchronous delay lines, described in [5] as occupying chip area of the same order as the PLL described in [88]. The most general multi-clock generator derives all required clocks as sub-multiples of from one high-rate reference. Allowing a factor of two in chip area for the circuits added to the PLL to derive multiple subsystem clocks as sub-multiples of the PE clock, a multi-clock generator would occupy less than 4×10^6 sites.

An F_0 cache controller contains a state register with next-state logic, 3 counter-registers, two instruction latches, a 4-input instruction selector, and a small number of other small logic blocks. An F_2 cache controller (shown in Figure A.11 to be a superset of an F_0 cache controller) contains an additional counter-register and an associated logic block. The small logic requirement for these simple I-cache variants is very likely to be dominated by the cache memory itself in the use of chip area for I-cache.

A typical register memory cell may be used as a conservative estimate for the chip area of the cache memory cell. A typical CMOS register memory cell occupies $40\lambda \times 40\lambda$ area, or 1600 sites. Note that single-port memories, as used with an F-family cache, occupy less chip area than their

	SLAP	MP-1	Blitzen	ALAPH
payload without I-cache (Π_g) ($\times 10^6$ sites)	36	118	356	1300
100-word cache: $\delta \approx 10 \times 10^6$ sites				
payload with I-cache (Π_c) ($\times 10^6$ sites)	26	108	346	1290
payload decrease (γ) (%)	28	8	3	1
1000-word cache: $\delta \approx 55 \times 10^6$ sites				
payload with I-cache (Π_c) ($\times 10^6$ sites)	0	63	301	1245
payload decrease (γ) (%)	100	47	15	4

Table 4.1: Summary of Chip-Area Estimates for Simple I-Caches in Four SIMD PE Chips

multi-port counterparts such as may be used in the PE register files. 32 of these cells, as needed for one 32-bit instruction of cache memory, occupies 51,200 sites. A conservative estimate would allow further chip area equivalent to 20 words of memory for sense amps and bit-line drivers, or about 1×10^6 sites.

The chip area occupied by one of these simple I-cache variants is obtained by adding the areas for the multi-clock generator area and the memory cell array with drivers. A resulting formula for the area occupied by an F-family I-cache variant, where N is the number of instructions in the cache, is given in Equation 4.9:

$$\delta \approx (5 + \frac{N}{20}) \times 10^6 \text{sites} \tag{4.9}$$

Figure 4.2 shows values of γ against N for the 4 PE chips whose chip area has been measured above. Figure 4.2 shows that for newer VLSI process technologies with values of λ below 0.5μm, up to 1K words of cache memory correspond to values of γ less than .15.

Figure 4.3 plots values of γ for each of the PE chips at a cache size of 100 instructions. Two curves are super-imposed on the set of points, the lower one ignoring the data point for the SLAP chip.

VLSI technology continues to improve over time so as to make possible larger manufacturable chips with decreasing geometric resolution [23]. The number of sites in a chip grows linearly with the physical area $H \times W$ and quadratically with the component density $\frac{1}{\lambda}$, as shown in Equation 3.1. The typical SIMD PE chip organization, wherein the available PE area is tiled with replicated bit slices, readily exploits increases in the size of the PE chip. As interior area I grows, the allowable number of PEs, number of registers per PE, and/or FU complexity increases. For example, re-implementation of a PE chip containing a PE of fixed FU complexity at higher I yields a PE chip containing an increased number of PEs, each with increased per-PE memory.

By contrast, the MCS interfaces and local controller occupy a number of sites that need not change as the VLSI technology improves, so that A_c remains roughly constant. For fixed A_c the VLSI technology scaling effects of increasing H and W while decreasing λ takes γ toward 0, so that value of $\frac{\Pi_c}{\Pi_g}$ given in Equation 4.7 tends toward 1. The scaling of VLSI fabrication process parameters cannot continue indefinitely, however, due to the existence of fundamental physical lower limits on the sizes of MOS devices and their interconnections [43, 48].

The estimates for γ suggest that VLSI implementation technique has only recently reached sufficient chip component densities so that a simple I-cache containing about 100 instructions occupies negligible chip area in a PE chip.

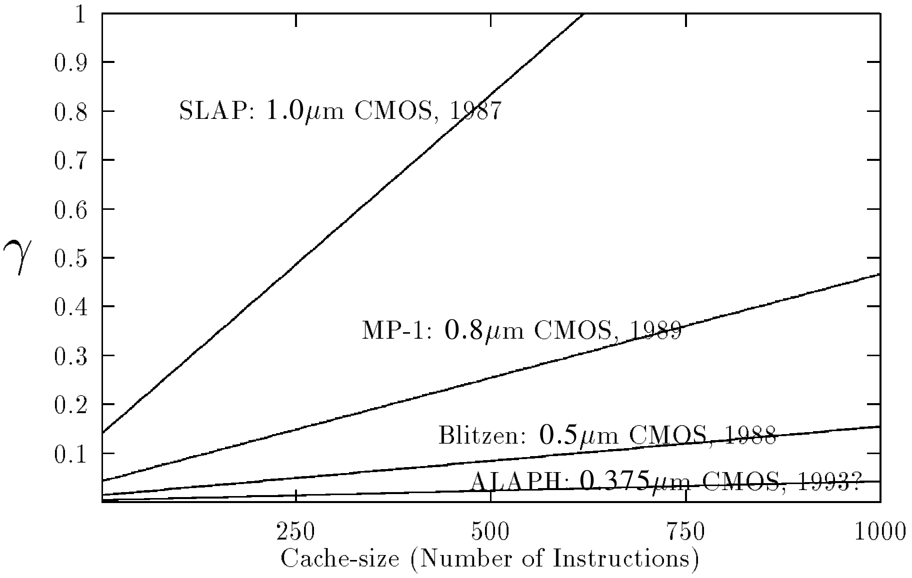


Figure 4.2: γ v. Cache Size for Each of Four PE Chips

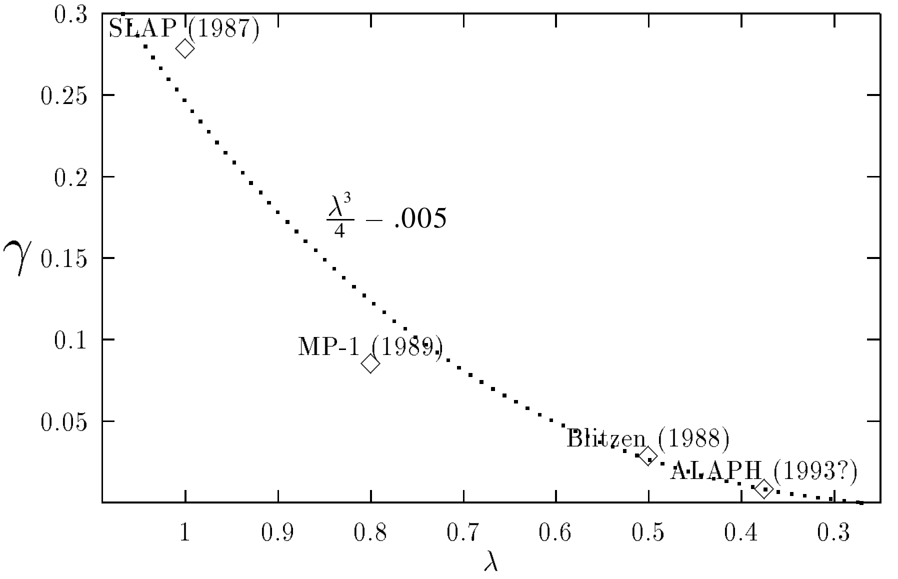


Figure 4.3: γ v. λ for each of the 4 PE Chips at N=100

4.4 Effects of Program Properties

SIMD computers are usually used to solve *data-parallel* problems. A data-parallel problem is divisible into a collection of sub-problems, each of which is associated with a subset of the problem-defining input data set [41]. One way to solve a data-parallel problem is to distribute the sub-problems to PEs such that the sub-problems are solved concurrently. This distribution induces a requirement for inter-PE communication that may limit the throughput of the computation. How much inter-PE communication is required is proportional to how much the sub-problems' data-subsets overlap. The degree of overlap varies among data-parallel problems. For a given data-parallel problem, distributing the sub-problems so as to minimize the amount of required inter-PE communication is important for achieving efficient computation [50, 51, 84].

High-level data-parallel programming languages often abstract the details of data set sizes and PE counts, allowing such values to be specified at compile time, or even at run time [18]. When there are far fewer PEs in the computer than there are sub-problems to solve, the computation consists of inner loops iterated many times. It is just this sort of computation, with many repeats of instruction sequences, for which I-cache should be most effective. On the other hand, if not many of the instructions are repeats, then I-cache cannot be very useful in speeding up the computation.

The observations that inter-PE communication frequency and loop iteration counts affect I-cache speedups raise the question, what are the properties of programs that should affect I-cache speedup? How are these properties assessed in estimating the I-cache speedup for an arbitrary program?

This section enumerates characteristics of programs and analyzes how each affects I-cache speedup.

4.4.1 Proportion of Repeat Instructions

The most important property of a program with respect to I-cache speedup is how many instructions are repeats. If no instructions are repeated, then I-cache is of no value, whereas if most instructions are repeats, then I-cache is of maximum value. Loop iteration counts express the degree of instruction repetition.

Consider the following program, **`simple`**:

```
program simple;
  B
   for j = 1 to J do
     A
   end;
end simple;
```

The symbols **A** and **B** in program **`simple`** denote sequences of instructions. Assuming that **A** is a cachable instruction sequence, the problems of determining which sequence to store in cache, when to store it, and when to activate it have obvious solutions.

Where the length of sequence **A** is denoted A and the length of sequence **B** is denoted B, the number of cycles of the system clock to run program **`simple`** is given as

$$\text{time for } \texttt{simple} = B + J * A \text{ cycles}$$

F_0 is the simplest member of the F-family of I-cache variants. F_0 is capable of storing only one cache block at a time and executing only single passes through the stored block. The following skeleton for program **`simple_cache`** illustrates how program **`simple`** is modified to use an an F_0 I-cache:

```
program simple_cache;
   B
   store sequence A in cache
   for j = 1 to J do
      activate cached sequence A
   end;
end simple_cache;
```

Ideally, each pass through cached sequence **A** in `simple_cache` is ρ_b times faster than each corresponding iteration of the inner loop in `simplest`. However, storing **A** in cache is an extra pass through that sequence of instructions in `simple_cache` that has no counterpart in `simple`.

Assuming that instruction sequence **A** is cachable and that it runs faster from cache by the factor ρ_b, then the time for the modified program to run with I-cache is given as

$$\text{time for } \texttt{simple\_cache} = B + A + J * \frac{A}{\rho_b} \text{ cycles} \tag{4.10}$$

The time for program `simple_cache` reflects the time to execute the instructions in sequence **B** in addition to the time to load **A** into the cache and subsequently execute it from there. The time for **B** represents the impact of un-cachable instructions on execution with I-cache, while the extra pass through **A** to store it into cache memory represents the run-time overhead of using I-cache. The I-cache speedup is given as the ratio of the two execution times:

$$\begin{aligned} \text{speedup for } \texttt{simple} &= \frac{B + J * A}{B + A + J * \frac{A}{\rho_b}} \\ &= \frac{\frac{B}{A} + J}{\frac{B}{A} + 1 + \frac{J}{\rho_b}} \end{aligned} \tag{4.11}$$

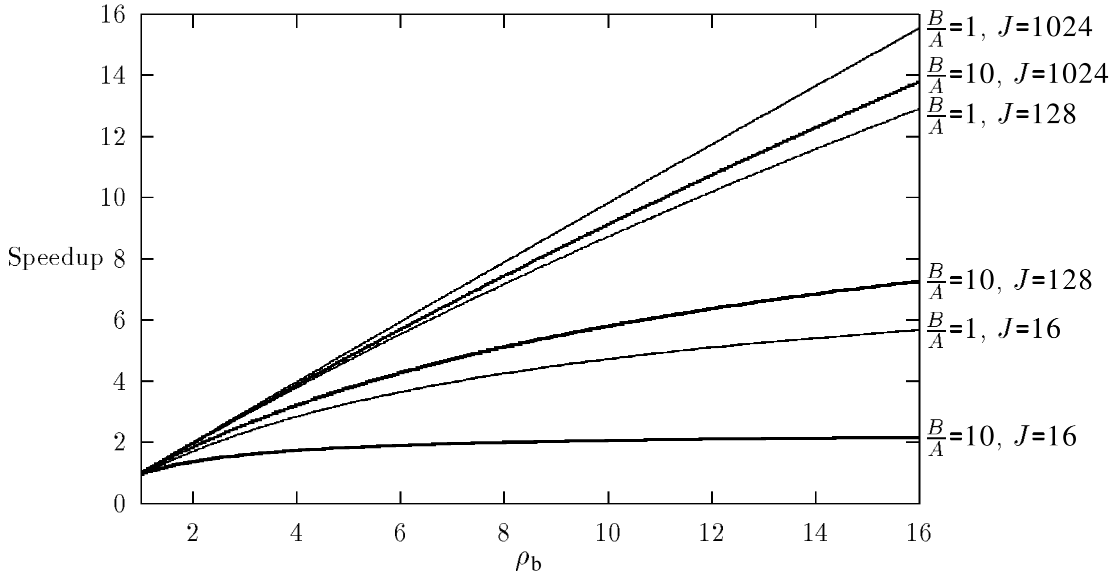


Figure 4.4: Ideal I-Cache Speedup for Program `simple`

Equation 4.11 suggests that the speedup for program `simple` approaches ρ_b as J approaches ∞, irrespective of the fraction $\frac{B}{A}$. However, if that fraction is large, so that the great majority of a program's instructions cannot be delivered from I-cache, then the speedup is significantly less than ρ_b. The impact of un-cachable instructions is greatest when J is low, as illustrated in Figure 4.4.

4.4.2 Quantization

Consider the speedup for **`simple`** when B=0, that is, when all of the program's instructions are cachable. Equation 4.11 becomes

$$\begin{array}{c} \text{speedup for } \mathbf{simple} \\ \text{when } B\text{=0} \end{array} = \frac{J * A}{A + J * \frac{A}{\rho_b}} = \frac{\rho_b * J}{\rho_b + J} \tag{4.12}$$

It is interesting to note that the length of sequence **A** cancels in Equation 4.12, so that the speedup is independent of the length of that sequence. Unfortunately, this simple expression is not entirely accurate. The time taken for a single pass through a cache block is *quantized* to an integer number of system clock cycles: when execution of a single iteration of a cache block completes in an F_0 I-cache, activity halts pending receipt of the next globally broadcast instruction. For an I-cache variant incapable of iterating cache blocks, this waiting time is spent after every pass through the block. Equation 4.13 gives the more accurate time to execute **`simple_cache`** with B=0 using an F_0 I-cache, and Equation 4.14 gives the corresponding speedup:

$$\begin{array}{c} \text{time for } \mathbf{simple\_cache} \\ \text{with quantization} \end{array} = B + A + J * \left\lceil \frac{A}{\rho_b} \right\rceil \text{ cycles} \tag{4.13}$$

$$\begin{array}{c} \text{speedup for } \mathbf{simple} \\ \text{when } B\text{=0} \\ \text{with quantization} \end{array} = \frac{J * A}{A + J * \left\lceil \frac{A}{\rho_b} \right\rceil} \tag{4.14}$$

Figure 4.5 shows how quantization affects I-cache speedup when B=0. The speedup impact of quantization is most pronounced for short loop bodies and for high iteration counts. Note that if A is a multiple of ρ_b, then quantization does not reduce I-cache speedup. The "steps" in Figure 4.5 occur at values of ρ_b where the quantized ratio $\left\lceil \frac{A}{\rho_b} \right\rceil$ decreases. It is apparent in Figure 4.5 that for fixed J, speedup does not increase simply with the length of the cache block as one might expect. Quantization causes the speedup curves to cross over one another anomalously at some values of ρ_b. Higher values of A tend to smooth out the effect of quantization.

4.4.3 Loop Structure

The objective of I-cache is to deliver repeated instructions at the highest rate from a repository within the PE chip. The actual speedup depends in part on the way in which instruction sequences occur in a program, and on how they are repeated. In many computations solving data-parallel problems, most of the time is spent executing instructions that reside in the bodies of inner loops. I-cache should work well for such programs. For one-block-at-a-time I-cache variants, including F_0 and F_2, loop bodies whose executions alternate in time displace each others' cache blocks. This alternation gives rise to a form of thrashing. Thrashing occurs also for multi-block I-cache variants when the capacity of cache memory is exceeded.

Having to re-store cache blocks reduces the I-cache speedup. To illustrate this effect, consider the program **`thrash`**:

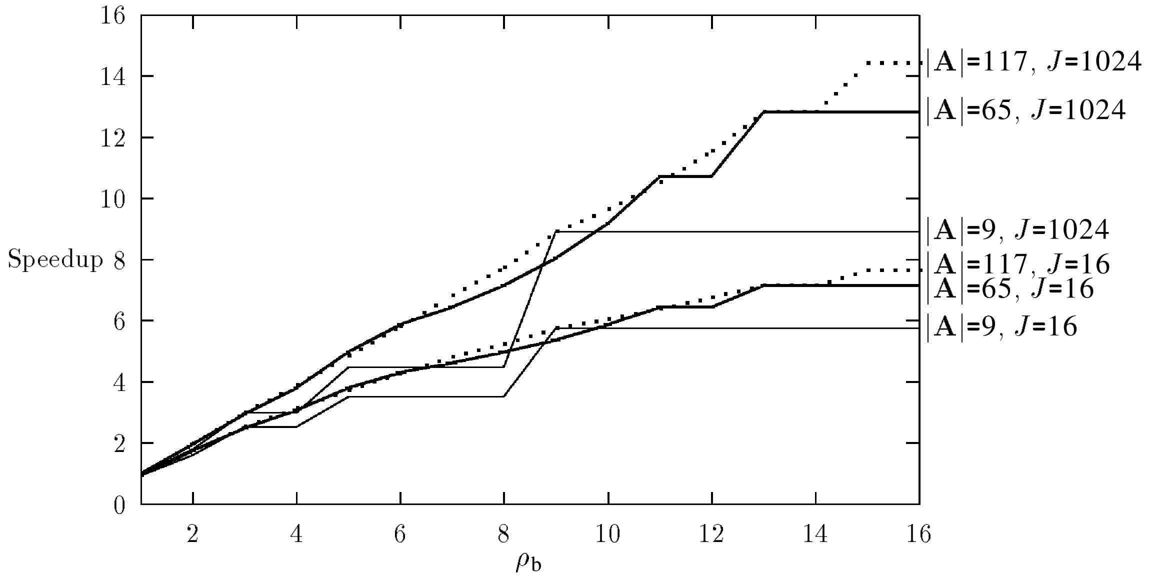


Figure 4.5: I-Cache Speedup for Program **simple** with Quantization, when B=0

```
program thrash;
   for i = 1 to I do
      for j := 1 to J do
         A
      end;
      for j := 1 to K do
         B
      end;
   end;
end thrash;
```

Where A and B are the lengths of instruction sequences **A** and **B**, respectively, then the runtime of **thrash** is given in Equation 4.15:

$$\text{runtime for } \mathbf{thrash} = I(JA + KB) \text{ cycles} \tag{4.15}$$

Clearly, if both instruction sequences could be accommodated in cache memory at once, as with a suitably large F$_1$ I-cache, then the instruction sequences would not compete, as in **thrash_F**$_1$:

```
program thrash_F$_1$;
   store sequence A in cache
   store sequence B in cache
   for i = 1 to I do
      for j := 1 to J do
         activate cached sequence A
      end;
      for j := 1 to K do
         activate cached sequence B
      end;
   end;
end thrash_F$_1$;
```

The runtime and speedup for **thrash_one** are given below:

$$\text{runtime for } \mathbf{thrash\_fone} = A + B + I(J\frac{A}{\rho_b} + K\frac{B}{\rho_b}) \text{ cycles}$$

$$\begin{array}{c}F_1 \text{ speedup}\\ \text{for } \textbf{thrash}\\ \text{ignoring quantization}\end{array} = \frac{I(JA + KB)}{A + B + I(J\frac{A}{\rho_b} + K\frac{B}{\rho_b})}$$

F_0 is a one-block I-cache, capable of containing only one cache block at a time. Therefore, the cache blocks in **thrash** replace one another in an F_0 I-cache, as shown in **thrash_**F_0:

```
program thrash_fzero;
   for i = 1 to I do
      store sequence A in cache
      for j := 1 to J do
         activate cached sequence A
      end;
      store sequence B in cache
      for j := 1 to K do
         activate cached sequence B
      end;
   end;
end thrash_fzero;
```

The runtime and speedup for **thrash_fzero** are given below:

$$\text{runtime for } \textbf{thrash\_fzero} = I(A + J\frac{A}{\rho_b} + B + K\frac{B}{\rho_b}) \text{ cycles}$$

$$\begin{array}{c}F_0 \text{ speedup}\\ \text{for } \textbf{thrash}\\ \text{ignoring quantization}\end{array} = \frac{I(JA + KB)}{I(A + J\frac{A}{\rho_b} + B + K\frac{B}{\rho_b})}$$

$$= \frac{(JA + KB)}{(A + J\frac{A}{\rho_b} + B + K\frac{B}{\rho_b})}$$

The thrashing of the cache blocks in **thrash_**F_0 increases the runtime by $(I - 1) * (A + B)$ cycles. Figure 4.6 illustrates this difference.

4.4.4 MCS-Intensiveness

During each PE clock cycle in an I-cached SIMD computation, some subsystem is busy, be it the PE FUs, one of the MCSs, the system controller, or some combination of these. When the computation cannot progress pending completion of an operation by a given subsystem, the computation is *subsystem-bound* at that point. For example, when inter-PE communication is in progress and there are no other operations to be performed that do not depend on the inter-PE communication result, then the computation at that point is inter-PE-communication-bound.

The likelihood of a computation becoming subsystem-bound by a given subsystem corresponds to the intensiveness of use of that subsystem in the program controlling the computation. One aspect of MCS-intensiveness is measured by the *calc-to-comm* ratio, the ratio of the number of calculation operations to the number of inter-PE communication operations occurring in a program. The calc-to-comm ratio expresses a program's intensiveness of inter-PE communication relative to FU calculation.

For example, consider the program **permuter** with the following structure:

```
program permuter;
   for j = 1 to J do
      Y: perform inter-PE communication operation
      A
   end;
end permuter;
```

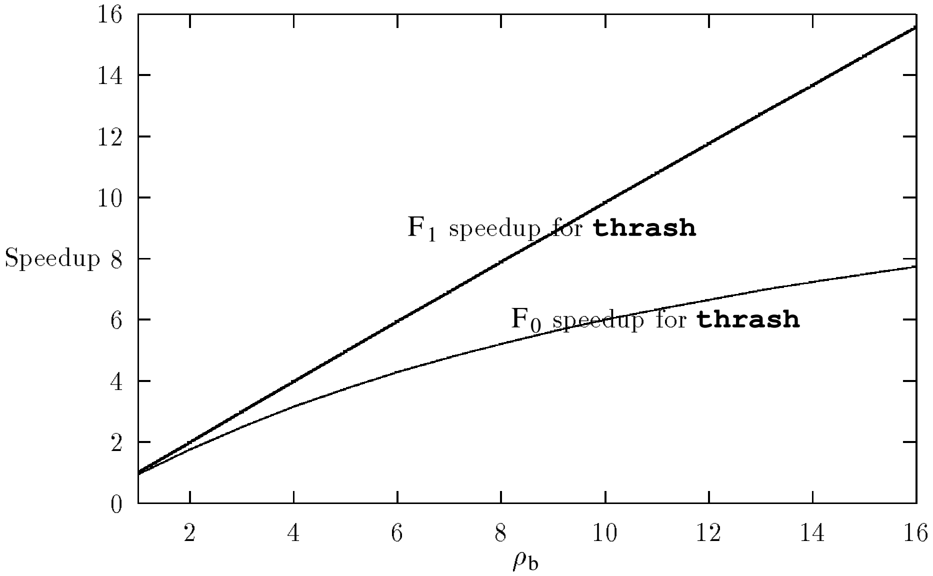


Figure 4.6: F_1 and F_0 Speedups for **thrash** (assuming A=B=40 and I=J=K=15, and ignoring quantization)

The inner loop of **permuter** contains an inter-PE communication operation (labeled Y) followed by an instruction sequence **A**. There are conditions under which I-cache cannot speed up a program like **permuter**. Specifically, for simple I-cache variants including F_0 and F_1 that are not capable of iterating stored instruction sequences, there is no I-cache speedup when the following two conditions obtain:

1. Instruction Y is flow-independent of the instructions in the sequence **A**. That is, Y may take place concurrently with any of the operations specified in sequence **A**.
2. The duration of instruction Y is at least as great as the time to globally broadcast all of the instructions in **A**.

The time to execute the loop body cannot be less than the time to execute instruction Y, which is determined by the duration of the inter-PE communication operation. Under these two conditions, delivering the instructions from **A** at a higher rate will not reduce the time needed to execute the loop body of **permuter**. Note, however, that these two conditions are highly restrictive. If there is even a single instruction in **A** that is flow-dependent on Y and thus cannot overlap Y, then I-cache may speedup the loop body. It is likely that there will be such a flow-dependence, because the result of the inter-PE communication specified in Y is likely to be used by another instruction in the loop body.

An additional condition must be met for an instruction sequence to exhibit no speedup for I-cache variants, including F_2, that are capable of iterating stored instruction sequences:

3. The duration of instruction Y is divisible by ρ_b.

Even if the first two conditions apply, if condition (3) does not apply, then an F_2 or higher F-family I-cache variant can begin executing the next iteration of the loop body without waiting for the ensuing globally broadcast instruction. For this reason, there may be some speedup unless condition (3) applies.

This discussion shows that I-cache speedup depends strongly on the particular operations performed in a program. I-cache speedup depends on the specific latencies of FU and MCS operations, the degree to which such operations may overlap (as constrained by flow-dependencies), and the ability of the local controller to iterate cache blocks without assistance from the system controller.

At the outset of a computation, before any instruction has been broadcast, the computation is global-instruction-broadcast-bound. Indeed, whenever no long-duration operation is outstanding and there is an operation whose operands are ready in PE registers and whose subsystem is idle, then the computation is instruction-delivery-bound. One way of viewing the function of I-cache is as minimizing global-instruction-broadcast-boundedness.

4.4.5 Relative Subsystem Clock Rates

The severity of subsystem-boundedness depends not only on the MCS-intensiveness of the program controlling the computation, but also on the electrical characteristics of the subsystems carrying out those operations. Subsystem electrical characteristics determine the time required for the subsystem to perform one step of an operation. For example, if local external memory is located physically close to the PE chip (shown together in Figure 3.2), then signaling through the local external memory subsystem may occur at the maximum inter-chip rate. The local external memory subsystem clock rate would be high in this case. As an example at the other extreme, router-to-router communication steps in an inter-PE communication network of high topological complexity might occur at as low a rate as that of global instruction broadcast. The inter-PE communication subsystem clock rate would be low in this case.

Recall (from Section 3.9) that a ρ-set characterizes the relative clock rates of the MCSs. A ρ-set of the form $\{N, N, N, N, N\}$, wherein the PE clock rate is N times faster than the clock rate of every MCS, characterizes a SIMD computer wherein all MCSs operate at the low rate of global instruction broadcast. For a given program, subsystem-boundedness is most likely when all MCS clock rates are low, because MCS operation durations are maximum. A ρ-set of the form {N,1,1,1,1} characterizes a computer wherein all MCSs other than the global instruction broadcast subsystem operate at the high PE clock rate. These two ρ-sets ratios represent the extremely slow and the extremely fast possibilities, respectively, for the clock rates of MCSs other than global instruction broadcast. For a given program, subsystem-boundedness is least likely when all MCS clock rates are low, because MCS operation durations are minimum.

4.4.6 Problem Size and PE Count

Problem size and machine size are two of the most important parameters of computation. Surprisingly, the speedup due to I-cache is largely independent of the data set size and the PE count.

To the extent that increased PE count decreases the operation rate of MCSs or increases the stepcount of MCS operations (for example, inter-PE communication, I/O, or response), increased PE count increases ρ-set values. Increased ρ-set values increases the time spent per MCS operation and thereby decreases I-cache speedup.

Similarly, increasing the problem size increases the amount of problem data stored by each PE. When the data storage requirement exceeds the PE's register file, some data is accessed in external memory at a lower rate than in the register file. In this way, increasing the problem size may decrease I-cache speedup.

As shown in Figure 4.4, I-cache speedup is extremely sensitive to iteration count. To a first approximation, iteration counts are proportional to the ratio of data set size to PE count. I-cache speedup is more sensitive to the ratio between data set size and PE count than it is to the absolute size of either of these parameters.

4.4.7 Data-dependence

Data-dependent instruction execution occurs in a program when the sequence of instructions is selected by values derived from the input data set. There are two types of data-dependence in data-parallel programs: local and global data-dependence. Local data-dependence occurs when the sequence of instructions to be executed by the PE depends on the value of intermediate data that is local to the PE. For example, an **IF-THEN** program construct specifies locally data-dependent instruction execution. Global data-dependence corresponds to conditional branching performed by the program-control portion of the system controller. Global data-dependence depends on aggregate information about the input data set. For example, a **WHILE** loop, executed until all PEs converge to some final state, is an example of global data-dependence.

Surprisingly, the consequences of these two types of data-dependence for I-cache speedup differ markedly. While global data-dependence tends to lower I-cache speedup, a high degree of local data-dependence actually favors large I-cache speedups.

As an example of local data-dependence, consider a simple program **local-cond**, in whose loop body the PE performs one of two sets of actions, depending on the value of the PE variable x:

```
program local-cond;
   for i = 1 to I do
       enable memory writes only where (x = 2)
     A
       enable memory writes only where (x != 2)
     B
   end;
end local-cond;
```

In **local-cond**, the PE should execute either sequence of instructions **A** or **B**, but not both, depending on the value of x in the PE. Because the PEs in a SIMD computer share a single instruction stream, both sequences **A** and **B** are broadcast to the PEs. Context management instructions direct the context manager (shown in Figure 3.4) to suppress writes to register memory within PEs to whose data the ensuing instruction sequence does not apply. Assuming that a context management operation is specified in a single instruction, the number of system clock cycles needed to execute **local-cond** in a generic SIMD computation is given below:

$$\text{runtime for } \mathbf{local\text{-}cond} = I(1 + A + 1 + B) \text{ cycles}$$

If **A** and **B** both contain cachable instructions, then the entire loop body can be placed in cache, resulting in the following program structure:

```
program local-cond_cache;
   store loop body in cache
   for i = 1 to I do
      activate cached loop body
   end;
end local-cond_cache;
```

The runtime and speedup for **local-cond_cache** are given below:

$$\text{runtime for } \mathbf{local\text{-}cond\_cache} = 1 + A + 1 + B + I\left(\frac{1 + A + 1 + B}{\rho_b}\right) \text{ cycles}$$

$$\begin{array}{c}\text{ideal I-cache speedup}\\ \text{for } \mathbf{local\text{-}cond\_cache}\end{array} = \frac{I(1 + A + 1 + B)}{1 + A + 1 + B + I(\frac{1+A+1+B}{\rho_b})}$$

Substituting the constant C=1 + A + 1 + B into this equation gives the speedup formula:

$$\begin{array}{c} \text{ideal I-cache speedup} \\ \text{for } \mathtt{local\text{-}cond\_cache} \end{array} = \frac{I * C}{C + I(\frac{C}{\rho_b})} = \frac{\rho_b * I}{\rho_b + I}$$

which is the simple speedup formula first introduced in Equation 2.2. Here, though, multiple instruction sequences are concatenated, separated by context management instructions, to form large cache blocks. Longer cache blocks are less susceptible to the quantization effect. To the extent that local data-dependence increases the lengths of instruction sequences appearing in loops, local data-dependence increases the I-cache speedup.

This effect is not surprising, for the following reason: SIMD computer architecture is motivated by a desire to remove redundant program-control components from the PE. Some amount of program control is appropriate for executing locally data-dependent programs, because the sequence of executed instructions depends on local conditions. I-cache can be seen as a way of putting a small amount of program-control logic back into the PE chip along with the PEs. It makes sense that the more independent program control a program requires, the better I-cache performs, within limits. The trick, of course, with respect to making the best use of available chip area, is to have just enough program control in the PE chip.

Global data-dependence has a very different impact on I-cache speedup. One reason is that global branches depend on conditions which require input from all of the PEs. The response network, usually used to query the state of the PEs, is typically a low-clock-rate MCS. Therefore, a program with frequent global branches is likely to be response-bound. Another reason that global data-dependence acts to lower I-cache speedup is that response instructions cannot be placed in cache. Whenever a response instruction is executed, global information regarding the ensuing instructions to be executed is required, which prevents the local controller from forging ahead inside the PE chip. Finally, if the global branch target instructions are not in cache, they need to be stored there through the slow global instruction broadcast network.

The difference between local and global data-dependence is summed up by observing the sequence of instructions delivered to the PEs. Although both kinds of data-dependence affect the instruction sequence executed by PEs, locally data-dependent instruction sequences are delivered to the PEs obliviously of how they are executed. In other words, the context management instructions determine what actually happens within the PE, but the sequence of instructions that is delivered to the PEs does not depend on the values of intermediate data. Not so global data-dependence. A globally data-dependent condition selects one of multiple candidate instruction sequences for delivery to the PEs. Local data-dependence tends to lengthen sequences of cachable instructions, increasing I-cache speedup, whereas global data-dependence tends to curtail cachable sequences, decreasing I-cache speedup.

4.5 SIMD Instruction Cache Management

An I-cached SIMD computation embodies two concurrent control threads: one running on the system controller, and another running replicatedly on the local controller in each PE chip. This control concurrency represents a departure from generic SIMD computation, wherein a single program running on the system controller specifies on every clock cycle both the activity within the system controller and the instruction delivered to the PEs.

The system controller maintains the program's principal control sequence. The thread running on the local controller is not always active; when the needed instructions are not present in cache, the local controller is said to be *locked* to the global instruction broadcast stream. Any activity that occurs within the PE chip while the local controller is locked occurs at the instruction broadcast rate. When the local controller is locked, either the PEs idle or they execute whatever PE instructions are broadcast from the system controller, depending on the cache design. In any event, the cache controller itself operates continually under control of globally broadcast instructions. When a cache block has been fully stored in the cache, the system controller is able to broadcast a *fork* activating that cache block. When the local controller begins executing that cache block, the cache controller sequences through the cache block at the PE clock rate. When cache block execution terminates, the local controller re-locks to the globally broadcast instruction stream.

A program counter in the cache controller advances at a rate different from that of one in the system controller. Since the relative rates of advance are fixed and known statically, it is possible for the system controller to maintain an accurate model of the state of the cache controller. This model is used in system controller cache management.

I-cache management is a programming problem of assigning repeated instructions to cache blocks, causing cache blocks to be stored, and causing them to be activated. Static management of ordinary direct-mapped instruction caches is used to minimize conflict misses among frequently executed instructions [57]. This section illustrates the sub-problems of I-cache management by showing examples of statically managing F-family I-cache variants. For static cache management, the sub-problems are solved at compile time. The illustrations presented here are representative also of dynamic I-cache management actions; although the program transformations are applied during the computation under dynamic I-cache management rather than beforehand, the modifications themselves are the same as for static I-cache management.

4.5.1 Step 1: Identify Cachable Instructions

An assembly language program specifies a sequence of operations that makes up a SIMD computation. That assembly language program is translated into a machine code program which is stored in system controller instruction memory prior to the computation. The assembly language instructions specify system controller operations as well as PE FU and MCS operations. The instructions that are globally broadcast during a generic SIMD computation, by contrast, are PE machine code instructions that do not specify program-control operations. For I-cached SIMD computation, a limited set of program-control instructions forming a cache-control protocol are added to the set of globally broadcastable instructions.

The goal of I-cache is to place all globally broadcast instructions that are repeated in cache and subsequently deliver them to the PEs from cache memory at a high rate. The machine code instructions corresponding to some assembly language program instructions cannot be placed in cache, because a given I-cache variant may lack the facilities to perform the associated program-control operations.

For example, instructions that alter global control flow are un-cachable with an F_0 I-cache variant. Only a restricted subset of program-control operations, those controlling fixed-iteration-count loop iteration, are cachable with an F_2 I-cache variant.

Some assembly language program instructions specify system controller indexer subsystem operations. These operations calculate loop-index-dependent values and are used to form literals for global broadcast to the PEs or to form system data memory addresses. The F-family contains simple I-cache variants that do not include indexer subsystems, so the machine code instructions corresponding to such assembly language instructions are un-cachable for F-family caches.

A basic block is a sequence of instructions containing no conditional branching and no branch targets [29](p.478); the instructions in a basic block are executed in a group irrespective of problem-instance data. For SIMD programs, PE context management instructions delimit basic blocks with respect to code re-ordering because they delimit the boundaries of conditionally executed instruction sequences. However, context management instructions do not affect cachability, because context management operations restrict the side-effects of instruction execution rather than the order in which subsequent instructions are executed.

The fact that some instructions other than conditional branches are un-cachable for simple I-cache variants means that caching is restricted to sub-sequences of basic blocks. An un-cachable instruction has the greatest negative impact for I-cache variants that are capable of iterating cached loops: an un-cachable instruction in a loop body prevents iterating the loop body in cache.

It is straightforward to identify cachable instructions from an assembly language program for a given I-cache variant. All of the resulting machine code instructions are cachable, with the exception of those corresponding to assembly language instructions that specify program-control functions that the I-cache variant is incapable of performing.

4.5.2 Step 2: Determine Which Sequences Become Cache Blocks

The objects that go into a SIMD instruction cache are sequences of instructions, rather than individual instructions. The reason is a consequence of the explicit management of I-cache: at least one extra cache-control instruction is required to place an instruction sequence in cache, and at least one cache-control instruction is required to activate a cache block. There is no benefit to caching an individual instruction (unless it appears alone in an iterated loop body, and then only for an F_2 or higher I-cache variant capable of iterating cache blocks). This restriction further differentiates SIMD instruction cache from typical instruction caches, wherein it is profitable to store individual instructions. (To exploit spatial locality in memory references, fixed-size groups of instructions are stored collectively as *lines* in ordinary caches [75](p.477). However, the use of multiple-instruction lines is orthogonal to the exploitation of temporal locality that is the fundamental motivation behind caching [85].)

A cachable instruction sequence that does not become a cache block is said to be *excluded* from cache. Not every instruction sequence that is a candidate for caching can be beneficially executed from cache. There are two possible reasons for excluding a candidate instruction sequence: placing the instruction sequence in cache may not speed up the execution of that sequence, or the instruction sequence may compete for cache space with other, more profitably cached alternative sequences.

- Exclusion due to ineffectiveness:

  Some instruction sequences take no less time when executed from cache than when globally broadcast. Such is the case, for example, for non-iterated single-instruction sequences. As another example, caching may not speed up an instruction sequence that is subsystem-bound. Because storing a cache block represents a time overhead proportional to the length of the sequence, caching an instruction sequence for which there is no cache speedup actually slows down the computation.

  A single instruction that is not the body of a one-instruction iterated cache block cannot be executed faster from cache, irrespective of ρ_b and of the number of times that particular instruction is used. Such a cache block is excluded from an F_0 cache. An F_2 cache is capable of iterating cache blocks, so single-instruction sequences are not necessarily excluded from an F_2 cache.

  The possibility of ineffectiveness makes it important to estimate statically the speedup from caching a given instruction sequence. Such estimation is discussed in Section B.4.5.

- Exclusion due to competition:

  Because F_0 and F_2 I-cache variants contain only one cache block at a time, all cachable instruction sequences compete for cache space with each other: whenever a block is cached in an F_0 or F_2 cache, it displaces the previously cached block.

Competition for cache space as occurs among instruction sequences whose executions alternate during the computation. Competition may lead to exclusion of some of the competing sequences. The determination of whether to exclude one or the other of a mutually conflicting pair of cachable sequences rests on the tradeoffs between the time to store each cache block versus the time saved by running that block from cache. If the time saved by caching the less-profitably cached instruction sequence is less than the time to re-store the more-profitably cached instruction sequence, then the less-profitably cached instruction sequence should be excluded.

4.5.3 Step 3: Determine Where in Cache to Place Blocks

There is no decision to be made here for F_0 or F_2 caches, because these caches contain only single cache blocks at a time. In the general case, this problem is equivalent to the storage management problem solved, for example, by segmented virtual memory replacement algorithms. Note that the correct solution depends collectively on the dynamic execution characteristics of a program's cache blocks and their sizes; this sub-problem is arbitrarily difficult for arbitrarily complex programs.

4.5.4 Step 4: Schedule Cache Blocks

With respect to scheduling the machine code instructions in a program controlling a SIMD computation, time is measured in numbers of instruction slots. The time interval of a global broadcast instruction is ρ_b times the time interval of a cached instruction. Therefore, operation latencies as measured in numbers of instruction slots are ρ_b times higher for cached instructions than for globally broadcast instructions.

Quantization affects the latencies of globally broadcast instructions but it does not affect the latencies of cached instructions, as measured in numbers of instruction slots. The following equations for the latency of an operation Y using MCS X whose stepcount is $S(Y)$ illustrate this point:

$$\begin{aligned}\text{latency of } Y &= \left\lceil \frac{S(Y) * \rho_X}{\rho_b} \right\rceil \text{ global broadcast instructions} \\ &= S(Y) * \rho_X \text{ cached instructions}\end{aligned}$$

The difference between the two latency measures reflects the fact that MCS X receives $\frac{\rho_X}{\rho_b}$ clock pulses during every global instruction broadcast interval, which corresponds to one clock pulse every ρ_X cached instructions.

The latency difference means that in general, the machine code instruction sequence corresponding to a given assembly language instruction sequence occupies more cached instruction slots than global broadcast instruction slots. The lengthening of instruction sequences for caching has a negative impact on I-cache speedup, because the time to store a cache block is proportional to the length of the cache block. The cache block store time becomes significant if it is not amortized over many iterations of the cache block's execution. Furthermore, lengthier cache blocks require larger cache memories, increasing the chip-area cost of a useful I-cache.

In any event, the scheduling dilation of cache blocks motivates the compression of sequences of ***NOOP***s that is common to all the F-family caches: in a manner reminiscent of the ***NOOP*** compression technique used to conserve instruction memory in VLIW computers [15](Sec.6.5.1), a sequence of ***NOOP***s added as place holders representing timing delays is represented compactly in an F-family cache. The simple encoding scheme associates a parameter with a cached ***NOOP*** that specifies a number of cycles for which to suspend incrementing the cache program counter before advancing to the next instruction in the cache block. Note that the need for such a compression scheme would be eliminated were the multi-clock generator augmented with control inputs that allow clocks regulating idle subsystems to be stopped and restarted at arbitrary phase. These observations suggest that although times as measured in instruction intervals are large for cached instructions than for globally broadcast instructions, this difference does not translate into cache blocks that are proportionally larger than their globally broadcast counterparts.

4.5.5 Step 5: Store Cache Blocks

A cache block is stored in cache before it can be activated. The storing of a cache block is accomplished by globally broadcasting the body of the cache block in-between a pair of cache-control operations demarcating the beginning and the end of the cache block. For F-family I-cache variants, $L + 2$ broadcast instructions are needed to store a block of L machine code instructions. The sequence of $L + 2$ instructions used to store the cache block into the cache is called the *preamble.* In F-family caches, the final instruction of the preamble is stored directly in the cache as a sentinel identifying the end of the cache block. Alternatives would have been to store block-bounds information in a special part of the cache, or to designate the bounds on each activation of a cache block. Each of these alternatives requires storage and/or control logic comparable in size to the single cache location it would save.

A question that arises in relation to storing the cache block is where in the program to place the preamble. The general answer to this question seems to be that it doesn't matter where in the program the preamble appears, subject to the following constraints:

- The preamble should be executed as infrequently as possible, because doing otherwise amounts to redundantly re-storing the block in the cache.
- The preamble should be executed as late as possible before the cache block is executed, because doing otherwise may cause the cache block to over-write another cache block that is still usefully resident in cache.

4.5.6 Step 6: Activate Cache Blocks

Upon activation of a cache block, all instructions globally broadcast prior to completion of the cache block's execution are not executed by the PEs. The number of system clock cycles for the execution of a cache block with an F-family I-cache variant is known statically. The system controller is free to perform useful work while a cache block is active. For example, for two-port I-cache variants, the system controller could globally broadcast another cache block to be pre-stored through the second cache memory port. Such pre-storing is not possible for F-family I-cache variants, as they have but a single cache memory port.

With F_0 I-cache variants, the globally broadcast instruction activating a cache block supplies no parameters, because the length of the block is determined by an embedded delimiter, the starting location of the block is always 0, and there is no iteration of the cache block. The number of system clock cycles that occur during execution of an F_0 cache block of length L is given in Equation 4.16:

$$\text{duration of } F_0 \text{ cache block execution} = \left\lceil \frac{L + 1}{\rho_b} \right\rceil \text{ system clock cycles} \tag{4.16}$$

The additional instruction time is that spent recognizing the cached sentinel instruction demarcating the end of the cache block.

With F_2 I-cache variants, the globally broadcast instruction activating the cache block specifies an iteration count. The number of system clock cycles that occur during execution of an F_2 cache block of length L iterated I times is given in Equation 4.17:

$$\text{duration of } F_2 \text{ cache block execution} = \left\lceil \frac{I * (L + 1)}{\rho_b} \right\rceil \text{ system clock cycles} \tag{4.17}$$

4.6 Examples of Static I-Cache Management

This section illustrates static I-cache management for F-family I-cache variants for a program containing multiple cachable instruction sequences whose executions alternate. Consider the program **`twine`**:

```
program twine;
   for i = 1 to I do
      for j := 1 to J do
        A
      end;
      for j := 1 to K do
        B
      end;
      for j := 1 to L do
        C
      end;
   end;
end twine;
```

Executions of instruction sequences **A**, **B**, and **C** alternate during the course of the computation. All three are cached in a multi-block I-cache variant with sufficient cache size. For one-block I-cache variants, or for multi-block variants with insufficient cache size, the three instruction sequences compete for cache space and replace one another each time round the outer loop. In this case, the cache block for **A** would be used J times before being overwritten by the cache block for **B**, which would be used K times before being over-written by the cache block for **C**, which would be used L times before being over-written once again by the cache block for **A**.

Note that a simply nested loop structure is a special case of **`twine`**. For example, if $I > 1$, J=1, $K > 1$, and L=0, then **A** represents the outer loop body and **B** represents the inner loop body.

Static Management of `twine` for an F_0 I-cache Variant

There are many possible ways to manage **`twine`** for a "one-block, one-shot" I-cache variant. The best way depends on the I-cache speedups for the individual instruction sequences. If all three sequences yield large I-cache speedups, then the best way to use an F_0 I-cache variant is shown as program **`twine_`**F_0**`_all`**, wherein each block is re-stored in cache on each iteration of the outer loop:

```
program twine_F0_all;
   for i = 1 to I do
      store A's cache block
      for j := 1 to J do
         activate A's cache block
      end;
      store B's cache block
      for j := 1 to K do
         activate B's cache block
      end;
      store C's cache block
      for j := 1 to L do
         activate C's cache block
      end;
   end;
end twine_F0_all;
```

It might be the case, however, the greatest speedup is obtained by placing just one of the sequences in cache and leaving it there undisturbed throughout the computation. This would be the case, for example, if sequences **A** and **C** were subsystem-bound but sequence **B** were not. In this case, the best way to use an F_0 I-cache variant is shown as program **twine_**F_0**_best**, wherein the cache block for **B** is stored just once, at the outset of the computation:

```
program twine_F0_B_best;
    store B's cache block
    for i = 1 to I do
        for j := 1 to J do
            A
        end;
        for j := 1 to K do
            activate B's cache block
        end;
        for j := 1 to L do
            C
        end;
    end;
end twine_F0_B_best;
```

Static Management of **twine** for an F_1 I-cache Variant

A "multi-block" I-cache variant, including F_1, F_3, and F_7, is ideal for loop structures such as that exhibited by **twine**. Subject to capacity limitations, a multi-block I-cache is able to contain all three cache blocks at once, so the I-cache speedup from caching each sequence can be realized while paying the store overhead just once per block, at the outset of the computation. The result of static cache management for F_1 is illustrated in program **twine_**F_1:

```
program twine_F1;
    store A's cache block
    store B's cache block
    store C's cache block
    for i = 1 to I do
        for j := 1 to J do
            activate A's cache block
        end;
        for j := 1 to K do
            activate B's cache block
        end;
        for j := 1 to L do
            activate C's cache block
        end;
    end;
end twine_F1;
```

Static Management of **twine** for an F_2 I-cache Variant

F_2 is a "one-block, multi-shot" I-cache variant. Cache management for F_2 is subject to the same considerations as for F_1. If all three sequences from **twine** are profitably cached, then **twine_**F_2**_all** results:

```
program twine_F2_all;
    for i = 1 to I do
        store A's cache block
        activate J iterations of A's cache block
        store B's cache block
        activate K iterations of B's cache block
        store C's cache block
        activate L iterations of C's cache block
    end;
end twine_F2_all;
```

If the instructions in the individual sequences are such that it is best only to cache sequence **B**, then `twine_`F_2`_B_best` results:

```
program twine_F2_B_best;
    store B's cache block
    for i = 1 to I do
        for j := 1 to J do
            A
        end;
        activate K iterations of B's cache block
        for j := 1 to L do
            C
        end;
    end;
end twine_F2_B_best;
```

Static Management of `twine` for an F_3 I-cache Variant

F_3 is similar to F_1. F_3 has the ability to execute multiple iterations of a cache blocks from a single activation. If cache memory is sufficiently large to hold all three blocks, then program `twine_`F_3 results:

```
program twine_F3;
    store A's cache block
    store B's cache block
    store C's cache block
    for i = 1 to I do
        activate J iterations of A's cache block
        activate K iterations of B's cache block
        activate L iterations of C's cache block
    end;
end twine_F3;
```

Static Management of `twine` for an F_7 I-cache Variant

F_7 is the most complex member of the F-family. The entire program body may be stored intact with an F_7 I-cache variant:

```
program twine_F7;
    store A's cache block
    store B's cache block
    store C's cache block
    store the outer loop's cache block, which activates the others
    activate I iterations of the outer loop's cache block
end twine_F7;
```

Chapter 5

I-Cache Evaluation

The complex interactions of I-cache capabilities with logical properties of programs and electrical characteristics of SIMD computers make it difficult to discern *a priori* the I-cache speedup for a given SIMD computation. This analytical difficulty motivates empirical evaluation. Measurements of I-cache speedup against varying computation parameters provide a basis for evaluating the factors that affect I-cache speedup.

Designs for F_0 and F_2, two of the simplest members of the F-family, have been evaluated empirically on a detailed model of SIMD computation. Speedup measurements have been performed over a set of SIMD computer variants for a diverse collection of sample programs. The SIMD computer variants were chosen to represent a range of existing SIMD computers. The PE datapath widths in the SIMD computer variants range from 1 bit to 32 bits. The sample programs for which evaluations were performed were chosen to span a broad range of properties of data-parallel problems. Together, the SIMD computer variants and sample problems on which I-cache variants were simulated cover a large region of the space of SIMD computations in which to explore I-cache speedups, limitations, and costs.

Results obtained using the simulator show that the multi-clock generator, introduced to provide the multiple high-rate clocks needed for I-cache, does not provide substantial speedups on its own without I-cache. However, even the simplest I-cache variant (F_0) yields substantial speedups. An F_2 I-cache has the additional capability of iterating cache blocks. The measured results confirm that the ability to iterate cache blocks smoothes the quantization effect and yields higher overall speedups. For the subject problems solved on a SIMD computer with wide-word PEs, the evaluations show that these simple I-cache variants do not need to be able to store more than 50 instructions. For some sample problems, the simple I-caches yield speedups near the highest possible, while for others the speedups are much less than maximum. Measurements of the sensitivities of I-cache speedup to programs' MCS-intensiveness show the surprising result that increasing MCS-intensiveness either increases or decreases I-cache speedup, dependent upon the MCS' electrical characteristics.

5.1 A Simulator for SIMD Computations

The throughput of a SIMD computation is measured using a detailed simulator for the basis computer that is described in Appendix A. Programmed in machine code, the simulator represents the state variables of a SIMD computer explicitly, and it updates the state variables on each clock phase. The simulator is parameterized so as to be able to capture the characteristics of a broad range of generic SIMD computers. I-cached SIMD computers with F_0 or F_2 I-cache variants are also simulated. To check whether apparent speedup is due to I-cache or merely to the clocking of all subsystems at their highest rates, the simulator also represents multi-clock SIMD computers. A multi-clock SIMD computer has all of the elements of an I-cached SIMD computer, excluding cache memory and cache controller in the PE chip.

In the basis computer, there is no program control for the PE FU inside the PE chip. A machine code instruction controls one PE clock cycle of FU activity. Sequences of single-cycle machine code instructions are needed to perform some assembly language operations. For example, addition of 32-bit operands requires 8 single-cycle instructions on a 4-bit FU.

The "Control and pin access" block within the PE chip (shown in Figure 3.2) constitutes special-purpose local control for the MCSs. When an MCS operation requires multiple clock cycles to complete, this block provides the control for the MCS in clock cycles intervening between a pair of instructions initiating and terminating the multiple-cycle MCS operation. Why assume that there is MCS control inside the PE chip when there is no program control inside the PE chip? The typical simplicity of the control and pin access circuit, as for example that illustrated for local external memory access in Figure A.2, facilitates its inclusion in the PE chip. By contrast, the sequencing of primitive instructions to realize arithmetically complicated FU operations is not necessarily simple. The PE chips of some existing SIMD computers include some FU control, while in others there is no on-chip MCS control. The assumptions made for the basis computer, reflected in the machine code instruction set, introduce a degree of "machine-dependence" with respect to the severity of the global instruction broadcast rate limitation, because the FU is always instruction delivery-bound, whereas an MCS is not necessarily so. Providing FU control inside the PE chip would lessen the apparent instruction delivery-boundedness of programs, whereas assuming no MCS control inside the PE chip would increase the apparent instruction delivery-boundedness of programs. The machine-dependent assumptions made for the basis computer are a reasonable middle ground.

A further machine-dependent aspect of the basis computer is the genericity of MCSs. The MCS abstraction (which was introduced in Section 3.4) limits the faithfulness with which MCSs of some SIMD computers are represented. On the other hand, this generic representation allows a wide variety of specific MCSs to be accommodated in the one simulator.

Despite using a specific machine code instruction set, the simulator is parameterized so that it may reflect closely the widely varying characteristics of existing and foreseeable VLSI-based SIMD computers. In simulating generic SIMD computations, the following characteristics are parameters:

- program loop structure,
- program data-dependence,
- problem size,
- inter-PE communication, as determined by the topological relationship between sub-problems of a data-parallel problem and the inter-PE communication network,
- PE memory usage, as determined by the allocation of problem data and intermediate results to registers and to locations in local external memory,
- number of PEs in the computer,
- number of registers per PE,
- PE datapath width,
- PE FU circuit complexity, and
- number of PEs per PE chip.

Values for some of these parameters are explicit in the assembly language program used to describe an operationally structured subject computation. Those parameter values that determine operation stepcounts[1] are specified as inputs to the automatic translation of an assembly language program into the machine code program that controls the simulated computation.

A ρ-set[2] provides five additional parameters used in the simulation of multi-clock SIMD computations. A ρ-set reflects both the VLSI implementation technique characteristics that determine electrical propagation characteristics of wires and the MCS topologies that determine wire lengths and electrical loads. A ρ-set has the form $\{\rho_\mathrm{b}, \rho_\mathrm{r}, \rho_\mathrm{i}, \rho_\mathrm{c}, \rho_\mathrm{l}\}$, wherein each value is the ratio of the PE clock rate to the rate of a clock regulating an MCS. The values in a ρ-set specify the clock rates of global instruction broadcast, response, system memory data I/O, inter-PE communication, and local external memory, respectively. ρ-set values guide the translation from assembly language into machine code.

F_0- and F_2-enhanced SIMD computations are also simulated. For this purpose, the appropriate cache-control protocol instructions are available in the instruction set. Cache size, the total number of cache memory locations, is an additional parameter used in the simulation of I-cached SIMD computation.

Having verified assembly language program correctness and, where necessary, having measured data-dependent iteration counts through complete simulation of a computation for a given data set, throughput can then be measured statically for different ρ-sets through simple instruction counting. Measuring throughput in this way saves considerable simulation time.

5.2 Speedup Measurement Method

The empirical method used to measure I-cache speedup is illustrated in Figure 5.1. The method comprises the following steps:

1. Evaluation begins at the upper-most, left-most shaded box in Figure 5.1. A generic SIMD computation is represented as an assembly language program. The assembly language program used as the starting point reflects problem parameters (including data-dependence and the topology of sub-problem data sets) and hardware parameters (including inter-PE communication network topology, PE count, and PE register count). The assembly language program specifies a sequence of PE and MCS operations and their dependence relationships. The assembly language program is thus an *operationally structured* description of the computation.

2. The subject computation is then transformed into *physically structured* variants represented as machine code programs. Operation latencies are explicit in a physically structured computation. The machine code program therefore reflects PE parameters including FU circuit complexity and datapath width, and MCS parameters including PE chip pin time-sharing and communication network diameters. A distinct throughput baseline is established by simulating the computation described by each distinct set of parameter values.

3. The assembly language description of the subject computation is then again transformed, this time into physically structured variants of multi-clock SIMD computation. A unique variant is defined by each unique ρ-set. Simulations yield throughput measurements that are compared against the baseline for that computation.

4. The assembly language description of the subject computation is then modified to reflect enhancement with either F_0 or F_2. The modification includes adding cache-control instructions needed for static management of I-cache, as discussed in Section 4.5.

---

[1] As defined in Section 3.6, the stepcount of an operation is the number of clock cycles taken to perform it.

[2] The use of ρ-sets to characterize relative subsystem clock rates is introduced in Section 3.9.

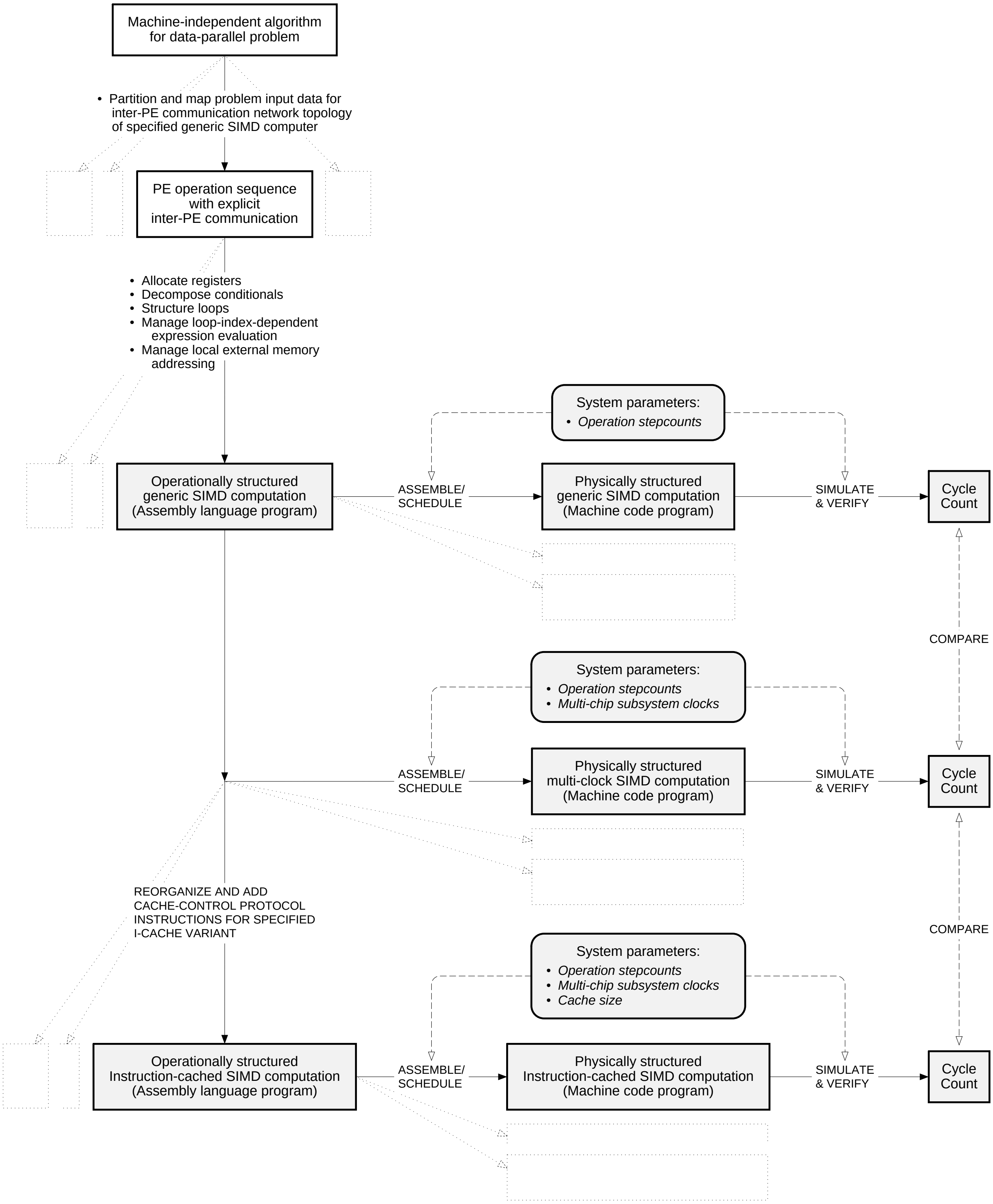


Figure 5.1: Method for Measuring I-Cache Speedup

5. The resulting assembly language description of the I-cached computation is then transformed into physically structured variants that reflect hardware parameters including cache size. Simulations yield throughput measurements that are compared against the baseline for that computation and against same-ρ-set multi-clock counterparts.

Dotted arrows and ghost boxes in Figure 5.1 indicate that a number of alternative transformations may be applied at a given step. The large branching factor in the tree of transformations shown in Figure 5.1 makes apparent the enormity of the I-cached SIMD computer design space.

The translation from assembly language into machine code has a slight impact on results. The assembler/scheduler that produces machine code from an assembly language program attempts to achieve the greatest degree of overlap among flow-independent instructions. The scheduling algorithm is heuristic and imperfect, and variations in the optimizer's success inject a degree of "noise" into throughput measurements. In general, good scheduling algorithms are difficult to write [49]. Although imperfect, the scheduling algorithm used uniformly for all translations to machine code shown in Figure 5.1 happens not to be a very bad one. On one hand, the variability introduced by idiosyncrasies of that scheduling algorithm adds to the realism of the simulation results. On the other hand, the scheduler's imperfections limit the generality of the results. The results reported here, based on detailed simulations under realistic assumptions, indicate the speedups that would likely be obtained were a contemplated I-cached SIMD computer actually constructed and its throughput measured relative to its generic counterpart.

5.3 Preparing A Subject Computation

The assembly language program that is the starting point for I-cache evaluation is an input to the method illustrated in Figure 5.1. The assembly language program describes an operationally structured generic SIMD computation. That program is prepared outside of the scope of the empirical speedup measurement mechanism. Assembly language programs were written by hand for each of the 8 sample problems for which results have been obtained.

In a subject computation, an explicit mapping from problem input and output data sets to PEs has already been established. In general, appropriate mappings are difficult to find, although there are automatic means for finding good mappings in some cases [84].

In a subject computation, the manner in which the PE registers and local external memory are used has already been established. The register allocation problem for SIMD PEs is an instance of the corresponding problem in uniprocessor computation. In general, the best register allocation is difficult to find, although there are algorithms for finding good ones [14].

In an operationally structured subject computation, program branches that are conditional on PE data has already been converted to PE context management instructions. How little of the entire program is executed within the scope of PE context management instructions is one measure of the appropriateness of a problem for execution on a SIMD computer. There is a systematic syntactic schema for converting conditional constructs into PE context management instructions [27](Sec.IV.B).

The system controller both sequences the program controlling a SIMD computation and evaluates loop-index-dependent expressions for the PEs. The use of the system controller is explicit in the assembly language program describing a subject computation.

5.4 Four SIMD Computer Variants

The experimental method sketched in Figure 5.1 is used to evaluate I-cache added to each of 4 SIMD computer variants. The computers differ primarily in the numbers of PEs per chip, in PE FU widths (in bits), and in VLSI implementation technique. Each is based on an existing SIMD computer, with the exception of the last, which is an extrapolation from the VLSI implementation technique used for a recent microprocessor.

This section describes the SIMD computer variants by giving representative stepcounts for assembly language operations. The complete set of operations is given in Appendix A. For each SIMD computer, stepcounts are given in this section for operations representative of a class of operations applied to 32-bit operands in an instance of the SIMD computer containing 1024 PEs. For example, NOR is representative of bit-wise logical operations, ADD is representative of carry-chain-based arithmetic operations, and MULT is representative of more complex arithmetic operations. The following table describes some representative operations:

Operation Name	Meaning
NOR	bit-wise NOR
ADD	addition
MULT	multiplication
LC_PUSH_EQ	context management
LITERAL	global broadcast literal
LOAD	local external memory read
LDN0	neighbor-to-neighbor inter-PE communication

Variant SIMD-A is based on Blitzen [36]. SIMD-A's PE chip contains 128 1-bit PEs, each with 1K register bits.[3] Variant SIMD-B is based on MP-1 [62]. SIMD-B's PE chip contains 32 4-bit PEs, each with 1K register bits. Variant SIMD-C is based on SLAP [27]. SIMD-C's PE chip contains 4 16-bit PEs, each with 512 register bits. Variant SIMD-D is based on the technology used in a modern uniprocessor implementation [21]. SIMD-D's PE chip contains 2 32-bit PEs, each with 8K register bits. Physical characteristics of the PE chips correspond to those shown in Figure 3.2. The following table summarizes the characteristics of the 4 SIMD computer variants:

	SIMD-A	SIMD-B	SIMD-C	SIMD-D
λ	0.5μm	0.8μm	1.0μm	0.375μm
PEs per chip	128	32	4	2
FU bit-width	1	4	16	32
32-bit registers per PE	32	32	16	256
NOR stepcount	32	8	2	1
ADD stepcount	32	8	2	1
MULT stepcount	1056	263	34	1
LC_PUSH_EQ stepcount	32	8	2	1
LOAD stepcount	128	32	4	2
LDN0 stepcount	32	8	2	1

5.4.1 Sensitivity of Speedup to SIMD Computer Variant

One consequence of the assumption in the simulation model that a new machine code instruction is required for each clock cycle of an FU operation taking multiple steps is that every FU operation is necessarily instruction delivery-bound. The effect of this assumption is most marked for PEs with narrow-word FUs, wherein many clock cycles are needed to perform operations on 32-bit operands. Therefore, SIMD-A (with 1-bit FUs) is the most instruction delivery-bound of the 4 SIMD computer

---

[3] The CM-2 is a better-known computer whose PEs are also 1-bit wide [18]. The CM-2 was not used as a basis here because its design does not exploit VLSI implementation technique for the PEs. 3 local external memory accesses are need in CM-2 for each single-bit full-adder step performed by the PEs [22](p.20). Also, some of the arithmetic operations are performed outside of the PE chip in CM-2.

variants, while SIMD-D (with 32-bit FUs and single-step multiply) is the least instruction delivery-bound of the 4 SIMD computer variants. The sensitivity to SIMD computer variant is apparent among the I-cache speedup measurements for each problem presented in Appendix E.

5.5 Eight Sample Problems

This section introduces the 8 sample problems for which I-cache speedups have been measured. While the problems presented here vary over a broad range in their program characteristics, this collection should not be construed to represent comprehensively all data-parallel problems. The 8 problems were selected from among those discussed in the literature for their simplicity and for their diversity. For example, problems were chosen that map conveniently to various inter-PE topologies and which appeared to possess a variety of degrees of data-dependence and subsystem-boundedness.

The assembly language programs are included in Appendix D. Most of the sample programs begin with a "prologue", wherein the input data set is moved to the PEs from system data memory, continue with a "kernel" computation within and amongst the PEs, and conclude with the transfer of the output data set from the PEs to system data memory.

Each of the sample programs has been evaluated for computations using very large data sets. The largest data sets contain about one million elements, the number of pixels in a 1K by 1K image. The objective in choosing large data set sizes was to avoid potentially mis-leading constants that typically obtain for smaller data sets. Unfortunately, for half of the problems studied, a large data set under a datum-per-PE mapping means that there is an enormous number of PEs in the simulated computer. For computations including some simulations of physical systems, the validity of results grows with the size of the data set. For example, the finest tractable granularity, and thus the largest possible data set size, is often desirable in finite-element analysis computations [66]. However, despite the apparently good reasons for building computers containing millions of PEs, their cost is still prohibitive today.

5.5.1 Tree-Summation (**tree**)

Logarithmic-time summation is a common operation on arrays. **tree** is based on the algorithm sketched in [41](Fig. 1), wherein P PEs are arranged as a tree with P leaves. Each PE is assigned one element of the array and summation takes $O(\log P)$ steps. In the iterated loop body, the $(2i+1)^{\text{st}}$ active PE sends its accumulated partial sum to the $2i^{\text{th}}$ active PE, the number of active PEs is halved by de-activating the odd-indexed PEs, and the still-active PEs add the newly received value into their partial sums. **tree** achieves the desired data communication using a routed inter-PE communication network.

5.5.2 Plus-scan (**scan**)

The parallel-prefix sum $\vec{s}$ of a vector $\vec{v}$ is defined as follows:

$$s[j] = \sum_{i=0}^{j-1} v[i]$$

Plus-scan (also called "tree +-scan" in [9](p.1535) and abbreviated "**scan**" herein) uses routed inter-PE communication in calculating the parallel-prefix sum of a P-element vector using P PEs. **scan** comprises $\log P$ iterations of an up-sweep followed by $\log P$ iterations of a down-sweep. Each of the sweeps' loop bodies is structured similarly to the loop body in **tree**. **scan**'s complexity is greater than that of **tree** because of the need for PEs to carry forward in local memory partial sums from the up-sweep to the down-sweep (as suggested in Fig. 13 of [9]).

5.5.3 Linear Array Bubble Sort (`bubble`)

This algorithm is a straightforward generalization of the uniprocessor bubble sort. There is no asymptotically faster sorting algorithm on a linear array.

On each iteration of `bubble`'s loop body, first all pairs of PEs indexed $2i$ and $2i + 1$ compare their values, swapping where PE $2i + 1$ contains the lower value. Then pairs of PEs indexed $2i$ and $2i - 1$ compare their values, swapping where PE $2i$ contains the lower value. That the two PEs indexed 0 and $P - 1$ must be disabled during the second half of the loop body is slightly inconvenient. The loop iterates until the values are ordered from least to greatest within the PEs. The number of iterations depends on the initial permutation of the input data. There are at most $\frac{P}{2}$ iterations.

5.5.4 Mesh Row-Column Sort (`rowcol`)

This algorithm, described in [55](Lec.5,p.16), sorts P numbers on a $\sqrt{P}\times\sqrt{P}$ mesh in time $O(\sqrt{P}\log P)$. The loop body of `rowcol` first sorts the rows in alternating directions and then sorts the columns upward. The individual row and column sorts use bubble sort, and so take time $O(\sqrt{P})$, the length of a row or column. Sorting requires at most $\log(\sqrt{P}) = O(\log P)$ iterations of the loop body, so the asymptotic complexity of `rowcol` is $O(\log P\sqrt{P})$. In `rowcol`, as in `bubble`, the iteration counts depend on the initial permutation of the data.

5.5.5 Bitonic Sort (`bitonic`)

`bitonic` is an in-place merge sort that runs in $O(\log^2 P)$ time. `bitonic` uses a routed inter-PE communication network to realize the communication pattern given in the description of Batcher's algorithm in [52](p.112).

The algorithm works as follows: an unsorted P-element array is considered initially to be P sorted 1-element sub-arrays. The inner loop body of `bitonic` cuts in half the number of sorted sub-arrays while doubling the size of each; therefore $\log P$ iterations of the loop body yield the desired sorted P-element array.

The basic operation of the bitonic sort is an in-place merge that produces a sorted N-element array from 2 sorted $\frac{N}{2}$-element arrays. The in-place merge exploits the fact that the two initial arrays are sorted, so that individual element-comparisons yield information about sub-ranges of the arrays. The fascinating aspect of `bitonic` is that the addresses of PEs pairwise-involved in element comparisons depends only on N, P, and the PE indices. The in-place merge comprises $O(\log N)$ comparisons, and while the swaps are conditional on the compared data, the schedule of comparisons is independent of the values of the input data.

In the `bitonic` loop body, the in-place merge is performed concurrently for all of the sorted sub-arrays. The asymptotic complexity of the sort is therefore given as the product of the number of iterations of the loop body ($O(\log P)$) and the time of each in-place merge step (ranging from $O(1)$ up to $O(\log P)$ and thus $O(\log P)$ on average). The asymptotic complexity is $O(\log^2 P)$.

`bitonic` shares the property of `bubble` and `rowcol` that data are sorted in-place without requiring additional intermediate storage. However, of the three sorts, `bitonic` alone possesses the property that its runtime does not depend on the original permutation of the data to be sorted.

5.5.6 Matrix Multiply (`matmul`)

Matrix multiplication is a classic calculation-intensive data-parallel problem. `matmul` multiplies two $P \times P$ matrices to yield a third $P \times P$ matrix. `matmul` uses a P-element linear array, wherein each PE calculates one column of the result. `matmul` is discussed in detail in Appendix C.

5.5.7 Mesh Sobel Filter (`sobel`)

The Sobel filter is a simple edge-detection operation on gray-scale images [2](p.76). At each pixel in the input image, local gradients are calculated in the horizontal and vertical directions, and the value of the output image at each pixel is the square-root of the summed squares of the local gradients.

`sobel` runs on a mesh containing one PE per pixel, wherein each PE calculates a weighted sum of a neighborhood of pixel values. Repetition of an instruction sequence occurs only in the integer square-root calculation at the end of the program. The square-rooting works by successive refinement of an initial estimate until the value converges within an error threshold; this calculation iterates a data-dependent number of times, until all PEs' values have converged.

5.5.8 Linear Array Median Filter (`median`)

Median filtering replaces each pixel of a $P \times P$ image by the median of its local (3×3) neighborhood. **`median`** performs the computation on a P-element linear array, wherein each PE calculates a column of the output image, following the scan-line algorithm presented in [39](Fig.5.3)[4].

`median` has the most complex loop structure of the 8 sample problems, comprising 3 repeated instruction sequences used for small numbers of iterations in an interleaved manner. The 3 repeated instruction sequences are used to sort 3 pixel values, to skip the least value (by updating pointers) from among 3 sorted lists of values, and to select the output pixel value as the least remaining in the three lists.

5.5.9 Summary of Program Characteristics

Figure 5.2 summarizes the sample programs' characteristics relevant to I-cache speedup.

5.6 ρ-Sets for I-Cache Speedup Bounds

A ρ-set of the form $\{N, 1, 1, 1, 1\}$ characterizes a SIMD computer wherein the maximum operation rates of all MCSs other than global instruction broadcast are equal to the highest PE clock rate, while the global instruction broadcast rate is N times lower than that. A given operationally structured computation is least likely to be subsystem-bound when the operation rate of the subsystem in question is high, so a ρ-set of the form $\{N, 1, 1, 1, 1\}$ corresponds to the least subsystem-boundedness for a given operation sequence. Therefore, speedups obtained with a ρ-set of the form $\{N, 1, 1, 1, 1\}$ represent upper bounds for speedup obtained with I-cache.

At the other extreme, a ρ-set of the form $\{N, N, N, N, N\}$ characterizes a SIMD computer wherein the highest operation rates of all MCSs are no higher than the global instruction broadcast rate, which is N times lower than the PE clock rate. A given operationally structured computation is most likely to be subsystem-bound when the operation rate of the subsystem in question is low, so a ρ-set of the form $\{N, N, N, N, N\}$ describes a SIMD computer which shows the greatest possible subsystem-boundedness for a given operation sequence. Therefore, speedups obtained with a ρ-set of the form $\{N, N, N, N, N\}$ represent lower bounds for speedup obtained with I-cache.

The sensitivity of I-cache speedup to ρ-set ratio suggests that I-cache speedup is a rough measure of the FU-to-MCS ratio. In the basis computer simulated in the evaluations, the PE chip contains local control only for MCSs but not for the FU, such that FU operation sequences tend to be instruction delivery-bound but MCS operation sequences tend not to be instruction delivery-bound. This "machine-dependent" characteristic of the basis computer leads to the expectation that, all else being equal, programs with the highest FU-to-MCS ratios should exhibit the highest I-cache speedups.

---

[4] In that algorithm, the index into array **`ad`** appearing on the 6th line up from the bottom should be **`dptr`**, not **`aptr`**.

	`tree`	**`scan`**	**`bubble`**	**`rowcol`**	**`bitonic`**	**`matmul`**	**`sobel`**	**`median`**
Loop Structure								
nested loops				√	√	√		√
global conditionals			√	√			√	
# cachable seq's	1	2	1	3	2	2	1	3
alternating seq's				√				√
# alternating seq's				2				3
Data-Dependence								
local conditionals	√	√	√	√	√	√	√	√
mem address calc		√				√		√
comm address calc	√	√			√			
Inter-PE-Communication-Boundedness								
inter-PE topology	R	R	L	M	R	L	M	L
data set size	2^{20}	2^{20}	2^{12}	2^{12}	2^{20}	2^{20}	2^{20}	2^{20}
PE count	2^{20}	2^{20}	2^{12}	2^{12}	2^{20}	2^{10}	2^{20}	2^{10}
FU-to-MCS ratio	≈ 1	≈ 1	< 1	< 1	< 1	> 1	> 1	> 1
Other Subsystem-Boundedness								
uses local ext mem		√			√	√		√
uses response net			√	√			√	
Asymptotic Complexity (Order-# system clock cycles)								
N is data set size	$\log N$	$\log N$	N	$\log N\sqrt{N}$	$\log^2 N$	N	1	$\sqrt{N}$

Figure 5.2: The sample programs have diverse characteristics. (For inter-PE communication topology, "R" denotes routed inter-PE communications, "L" denotes linear array, and "M" denotes mesh.)

5.7 Speedups for Multi-Clock SIMD Computers

The local controller of a multi-clock SIMD computer is the same as the local controller of an I-cached SIMD computer, with the exception of the absence of the cache mechanism itself. Specifically, the multi-clock local controller contains a multi-clock generator that regulates each subsystem at its maximum rate. The multi-clock local controller also contains a means of adapting globally broadcast instructions for single re-broadcast within the PE chip in-phase with the PE clock.

A multi-clock SIMD computer should be somewhat faster than its generic counterpart, because MCSs are no longer necessarily rate-limited by the system clock. A multi-step MCS operation may take place at a higher rate in a multi-clock SIMD computer than in a generic SIMD computer.

Table 5.1 shows the speedup bounds for a multi-clock SIMD computer for the various sample problems.

ρ_b		**tree**	**scan**	**bubble**	**rowcol**	**bitonic**	**matmul**	**sobel**	**median**
2	min	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	max	1.42	1.38	1.00	1.02	1.19	1.14	1.01	1.10
4	min	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	max	1.80	1.69	1.00	1.02	1.31	1.14	1.01	1.10
8	min	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	max	2.01	1.85	1.00	1.02	1.36	1.14	1.01	1.10
16	min	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	max	2.14	1.95	1.00	1.02	1.39	1.14	1.01	1.10

Table 5.1: Speedups on a Multi-Clock Variant of SIMD-D at a range of Values for ρ_b. Speedup upper bounds are obtained with ρ-set $\{\rho_b, 1, 1, 1, 1\}$, and speedup lower bounds are obtained with ρ-set $\{\rho_b, \rho_b, \rho_b, \rho_b, \rho_b\}$. These values show very modest speedups for multi-clocking alone.

Table 5.1 shows that, as a lower bound, there is no speedup for any of the sample programs. This fact is not surprising, because the lower bound is obtained when the MCS clocks are all as slow as the system clock. In other words, when the ρ-set is $\{\rho_b, \rho_b, \rho_b, \rho_b, \rho_b\}$, the MCSs are not artificially rate-limited by the system clock.

The multi-clock speedup upper bounds for the sample programs range between factors of 1 and 2, even at ρ_b=16. That there is some speedup at the limiting ρ-set $\{\rho_b, 1, 1, 1, 1\}$. indicates that the computation is subsystem-bound by some subsystem that becomes faster relative to the system clock as ρ_b increases. In most cases, there is no increase in speedup beyond ρ_b=2. This observation indicates that, after having made the bounding MCS twice as fast relative to the system clock, the computation reverts to being subsystem-bound by global instruction broadcast.

In comparison to I-cache speedups, the speedups from multi-clock SIMD computers shown in Table 5.1 are negligible in most cases. This observation confirms that most of the speedup attributed to I-cache is in fact due to I-cache and not merely to the presence of the multi-clock apparatus in the PE chip's local controller.

5.8 I-Cache Speedup Bounds

A complete set of speedups measurements for each of the 8 sample problems on each of the 4 SIMD computer variants are provided in Appendix E. For each problem, there are four graphs shown on two consecutive pages of Appendix E, one per SIMD computer variant. Plotted on each graph, over a range of ρ_b=1 . . . 16 are the upper and lower bounds for the F_0 and F_2 I-cache speedups. The upper bound is the speedup attained with ρ-set $\{8, 1, 1, 1, 1\}$, wherein the rate of the clock controlling each MCS equals the maximum operation rate attainable within the PE chip. The speedup lower bound is attained with ρ-set $\{8, 8, 8, 8, 8\}$, wherein each MCS is regulated at the rate of global instruction broadcast. The I-cache speedup obtained with any other ρ-set lies between these two bounding curves. Superimposed on the measured data is a "simple-equivalent" speedup curve, whose significance is discussed in Section 5.10.

This section discusses I-cache speedup bounds measured on SIMD-D. The range of cache sizes required to obtain the I-cache speedups is shown as a range in the text accompanying each graph. Larger values in a range of cache sizes occur for larger values of ρ_b.

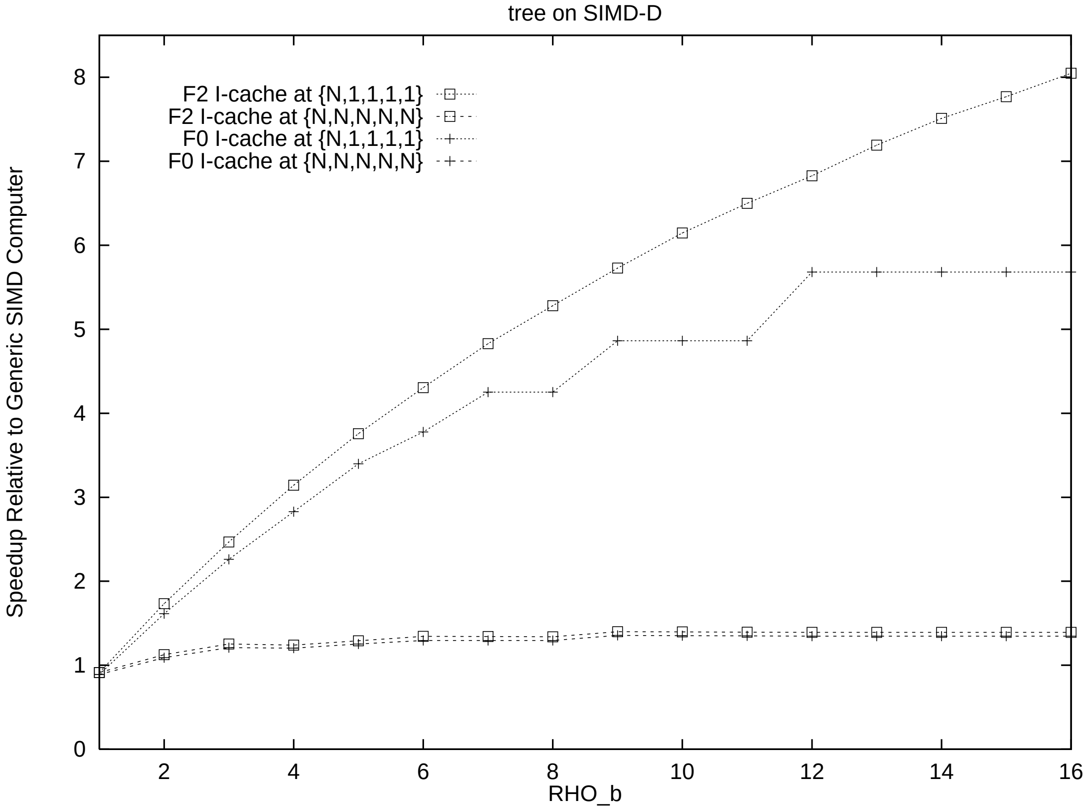


Figure 5.3: I-Cache Speedup Bounds for **`tree`** on SIMD-D

5.8.1 `tree`

Figure 5.3 shows the measured speedup bounds for **`tree`**. The required cache size lay in the range 20...23.

The loop structure of **`tree`** is very similar to that of the program **`simple`** introduced in Section 4.4, consisting of a short prolog followed by single iterated loop. The difference between the F_0 speedup upper bound and the F_2 speedup upper bound in Figure 5.3 illustrates the severity of the quantization effect for non-iterating I-cache variants. There is very little difference between the lower bounds for the two I-cache variants, indicating that the computation is subsystem-bound for ρ-sets of the form $\{\rho_b, \rho_b, \rho_b, \rho_b, \rho_b\}$.

Problems such as tree-summation are generally expected to be inter-PE-communication-bound, because of the small amount of calculation to be performed (an addition) per inter-PE communication operation. In part, the surprisingly significant I-cache speedups apparent in Figure 5.3 are due to the sequence of address calculations and context management operations that are associated with the inner loop's communication step. Such calculation-intensive instruction sequences are made faster with I-cache, which is why the upper bounds are high. However, when the communication operation is of long duration and it overlaps the calculations, speeding up the calculations does not decrease the time to execute the loop body, so there is little I-cache speedup.

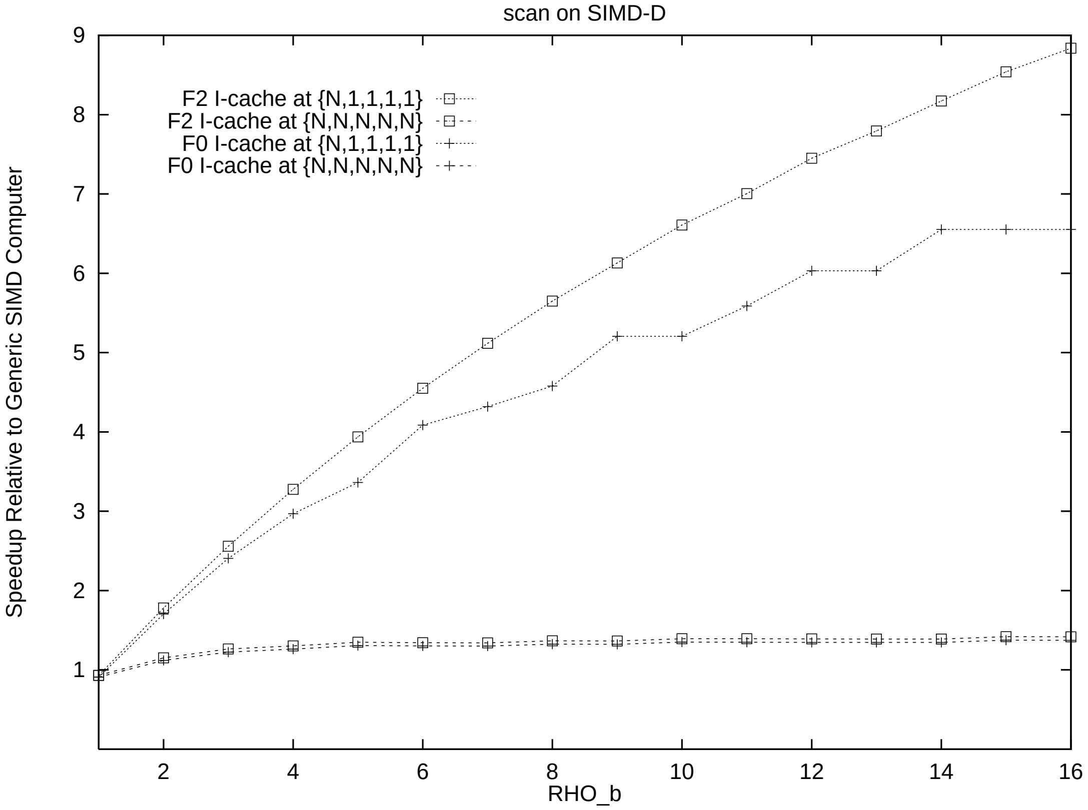


Figure 5.4: I-Cache Speedup Bounds for **`scan`** on SIMD-D

5.8.2 `scan`

Figure 5.4 shows the measured speedup bounds for **`scan`**. The required cache size lay in the range 27 . . . 35.

The results for **`scan`** are very similar to those for **`tree`**. This similarity is not surprising, because their loop structures are very much alike. Whereas **`tree`** contains a single loop, **`scan`** has two loops. However, the executions of the two loops occur one after the other, so the loop bodies do not conflict in cache memory. More simple loops means a greater number of repeat instructions, which is why the I-cache speedups for **`scan`** are slightly higher than those measured for **`tree`**.

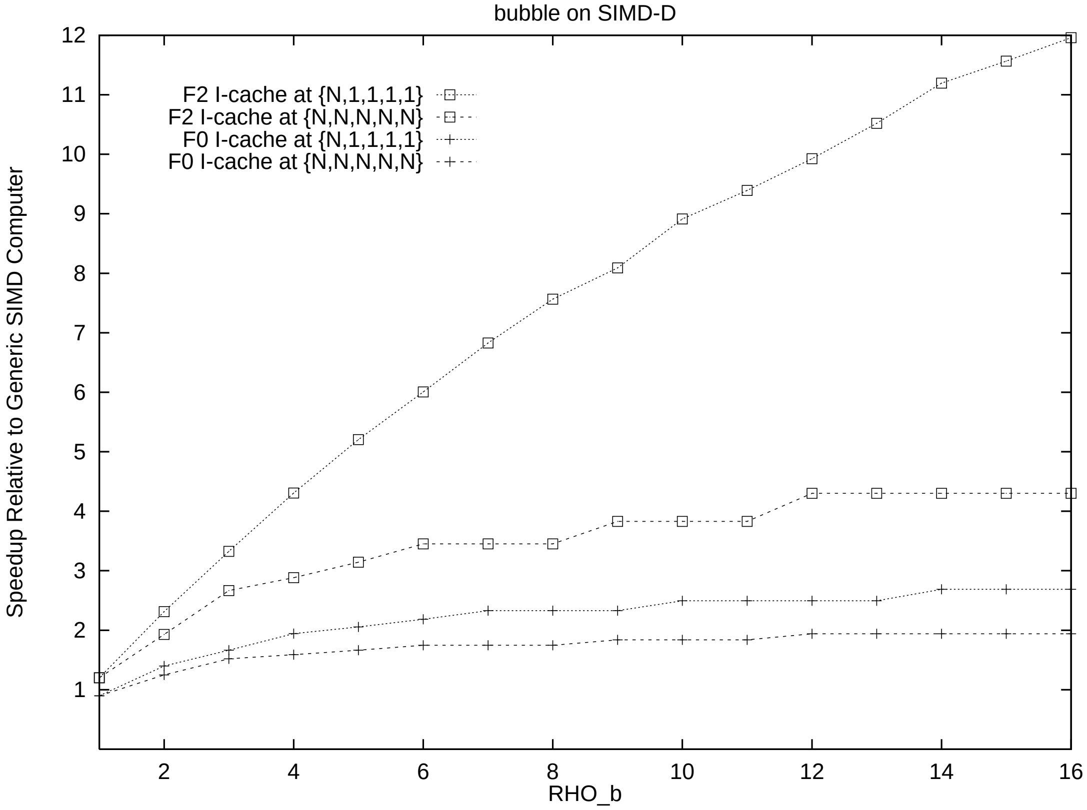


Figure 5.5: I-Cache Speedup Bounds for **`bubble`** on SIMD-D

5.8.3 `bubble`

Figure 5.5 shows the measured speedup bounds for **`bubble`**. The required cache size lay in the range 28...53.

The iteration count for the inner loop of **`bubble`** is globally data-dependent and varies between 1 and $N-1$. The local controller of an F_2 I-cache variant lacks the ability to make data-dependent iteration decisions. How then to use in F_2 I-cache, wherein a fixed number of iterations of the cached inner loop body are activated for a single globally broadcast cache-control instruction activating the cache block?

The F_2 management for which results are shown is to activate 10 iterations of the cached inner loop body at a time. The original program tests for completion of the sort after every iteration. The computation is complete when the data is everywhere locally ordered. Completion detection uses a response operation, by which the PEs signal completion to the system controller. The time for the response network to settle and for the system controller to make a branch decision is considerable. The static I-cache management choice for **`bubble`** for F_2 represents an algorithm change, because completion tests occur once every 10 iterations instead of every iteration. This change means that the inner loop may be executed up to 9 too many times, but this slight inefficiency is compensated by the increased rate at which the groups of 10 iterations execute. The fact that the F_2 speedup is greater than 1 even when ρ_b=1 suggests that the generic SIMD computation's throughput would be improved by unrolling the inner loop, perhaps by a factor of 10.

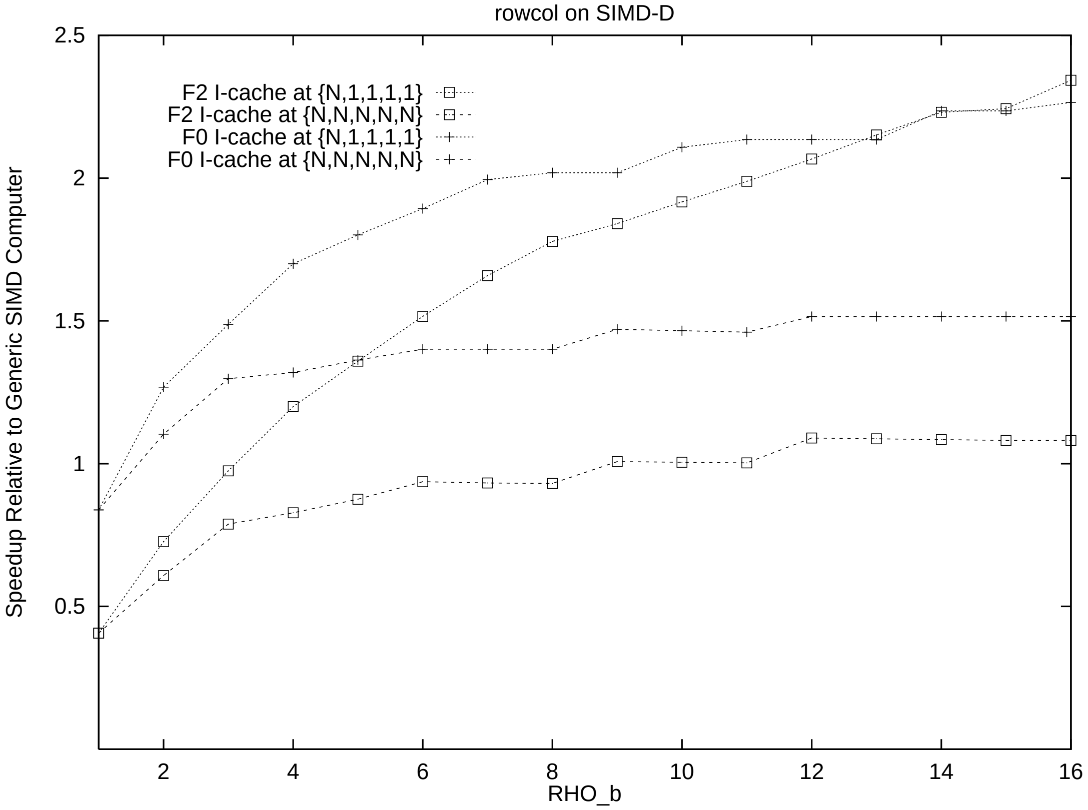


Figure 5.6: I-Cache Speedup Bounds for **`rowcol`** on SIMD-D

5.8.4 `rowcol`

Figure 5.6 shows the measured speedup bounds for **`rowcol`**. The required cache size lay in the range 32 . . . 53.

As is the case for **`bubble`**, the individual row and column sorts each require a data-dependent number of iterations. In an attempt to take advantage of F_2's iteration capability, the loop bodies have been “unrolled” by a factor of 4 in the F_2 program variant. The inferior performance of the F_2 I-cache apparent in Figure 5.6 illustrates that, if improperly managed, an F_2 I-cache variant performs less well than an F_0 I-cache variant.

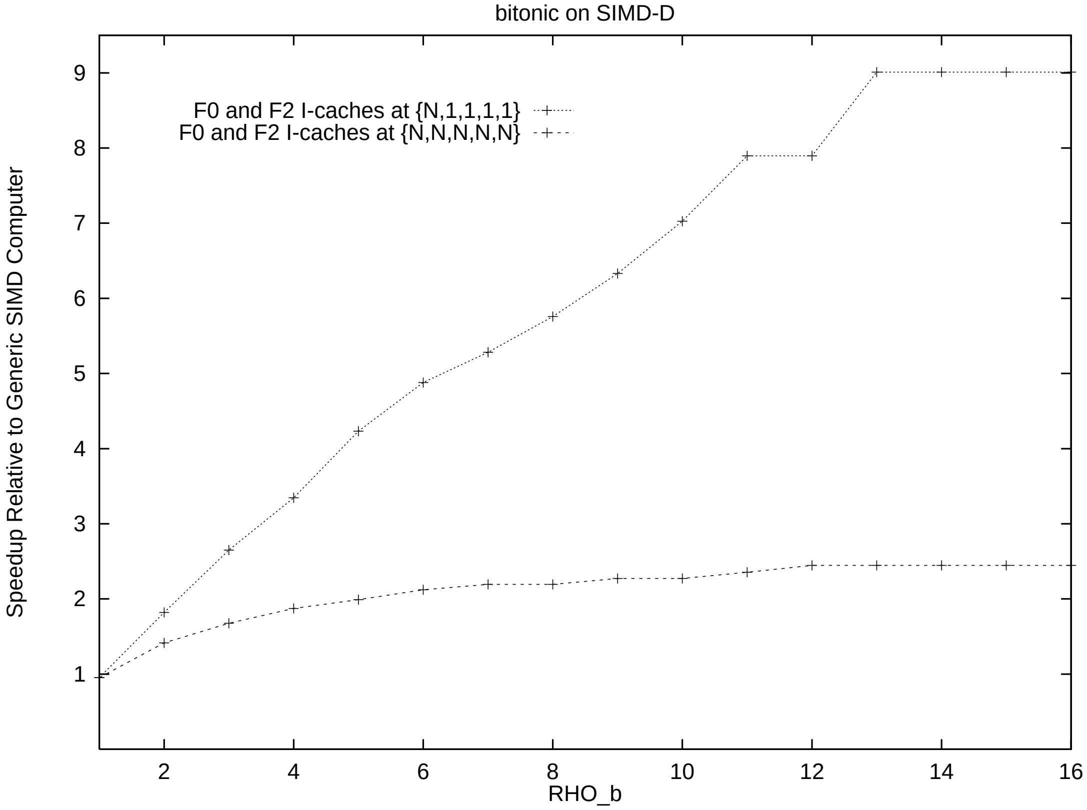


Figure 5.7: I-Cache Speedup Bounds for **`bitonic`** on SIMD-D

5.8.5 **`bitonic`**

Figure 5.7 shows the measured speedup bounds for **`bitonic`**. The required cache size lay in the range 50 . . . 53.

The number of iterations of the inner loop of **`bitonic`** varies on each iteration of the outer loop as a function of the outer loop index and the input data set size. The basis computer's system controller, described in Appendix A, lacks the capability to specify dynamically calculated iteration counts for cache block activation. A program using an F_2 I-cache specifies a single, fixed number of iterations for a cache block to be executed each time it is activated. This limitation makes it impossible to take advantage of F_2 I-cache for a program like **`bitonic`**, wherein inner loop iteration counts vary from activation to activation. Therefore, results obtained for F_2 I-cache are identical to those for F_0 I-cache. For clarity, only one set of speedups is plotted in Figure 5.7.

Although clearly suffering from quantization, the speedup upper bound is large. The speedup lower bound increases steadily as ρ_b increases, suggesting that even with the lowest MCS clock rates, there is some room for improvement from faster instruction delivery using I-cache.

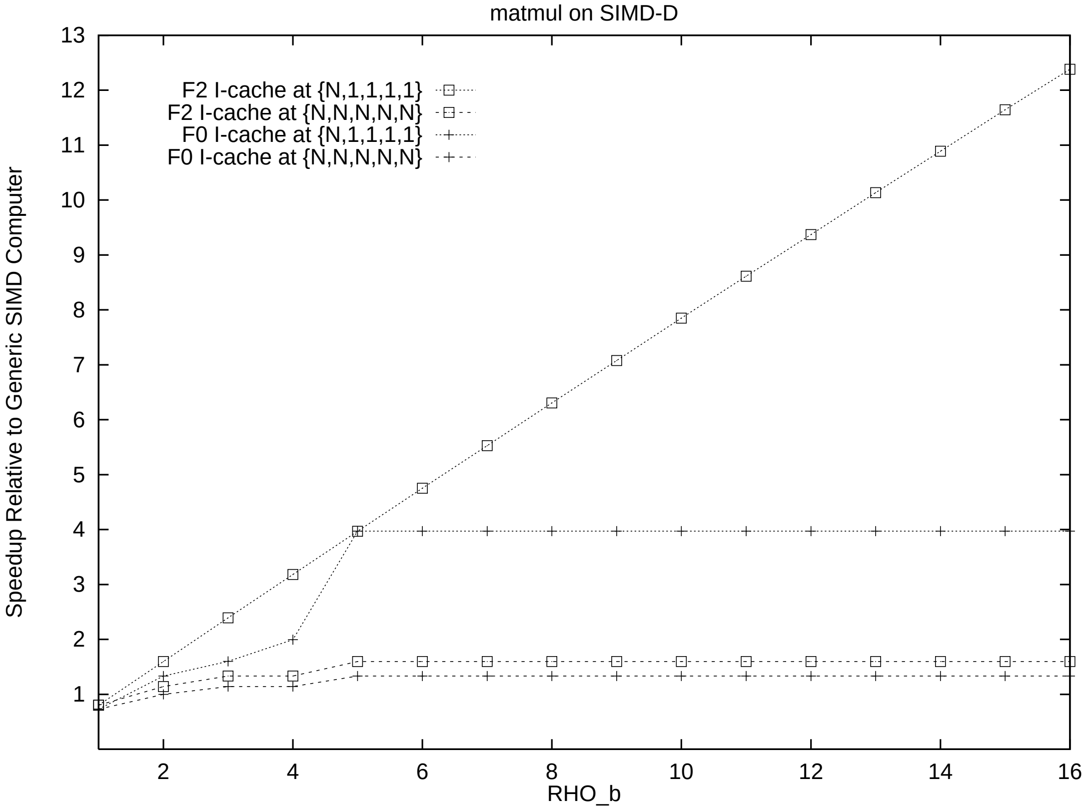


Figure 5.8: I-Cache Speedup Bounds for **`matmul`** on SIMD-D

5.8.6 **`matmul`**

Figure 5.8 shows the measured speedup bounds for **`matmul`**. The required cache size lay in the range 10 . . . 19.

The upper bound speedups for **`matmul`** are the best demonstration of the potential severity of the quantization effect. The F_0 speedup is identical to the F_2 speedup at ρ_b=5, but the F_0 speedup is flat after that value of ρ_b. The difference is due solely to quantization. Presumably at some $\rho_b > 16$, the F_0 speedup would jump up to meet the F_2 speedup again.

The F_2 speedup upper bound for this calculation-intensive data-parallel problem scales linearly with ρ_b.

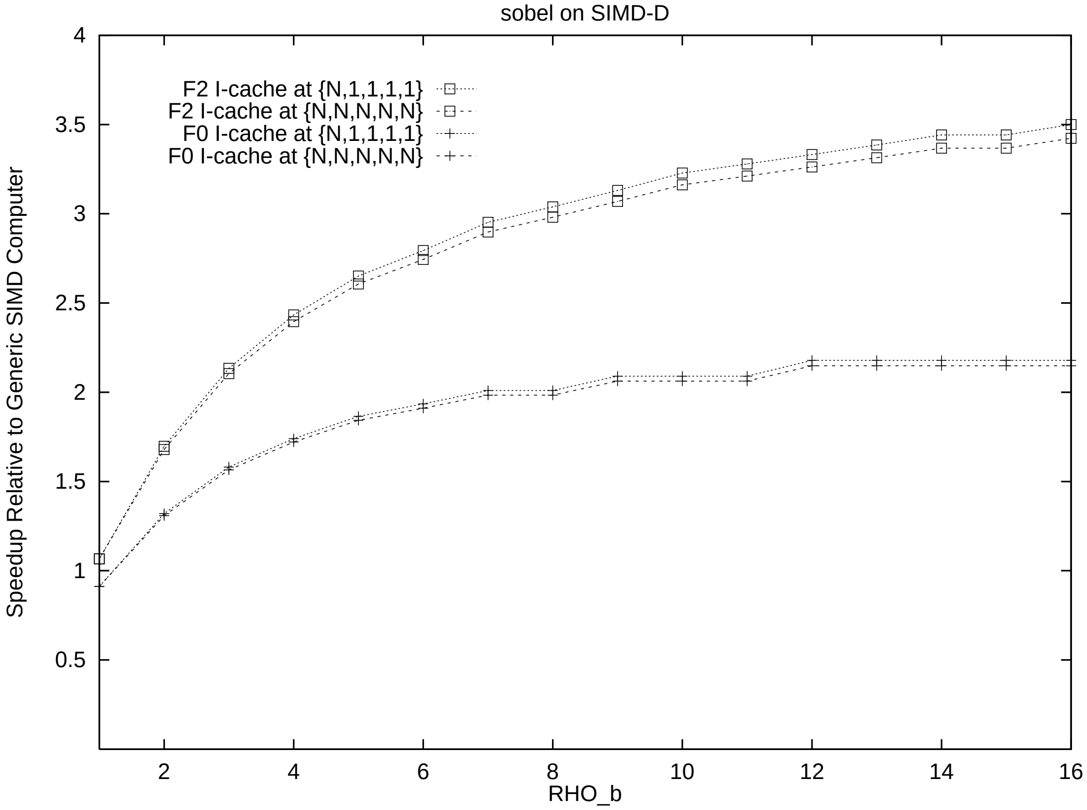


Figure 5.9: I-Cache Speedup Bounds for **sobel** on SIMD-D

5.8.7 **sobel**

Figure 5.9 shows the measured speedup bounds for **sobel**. The required cache size was 35 at all points.

sobel comprises a weighted sum that is performed at each pixel of an image. On a mesh-connected SIMD computer, wherein there is a unique PE for each image pixel, the sum calculation itself is not iterated. The only repeated instruction sequences arise in the root-sum-square calculation at the end of the program. The number of iterations of the square-root estimate-refinement loop is data-dependent. The static management decision for F_2 was to iterate individual activations of cache block corresponding to the square-root loop body 4 at a time. The F_2 speedups plotted in Figure 5.9 show that this slight algorithm change has a large impact, even yielding a small improvement at ρ_b=1.

The small differences between upper and lower speedup bounds confirm that the iterated part of the program is extremely calculation-intensive. The small total iteration count and the slow response operation used to detect completion keep the I-cache speedups modest for **sobel**.

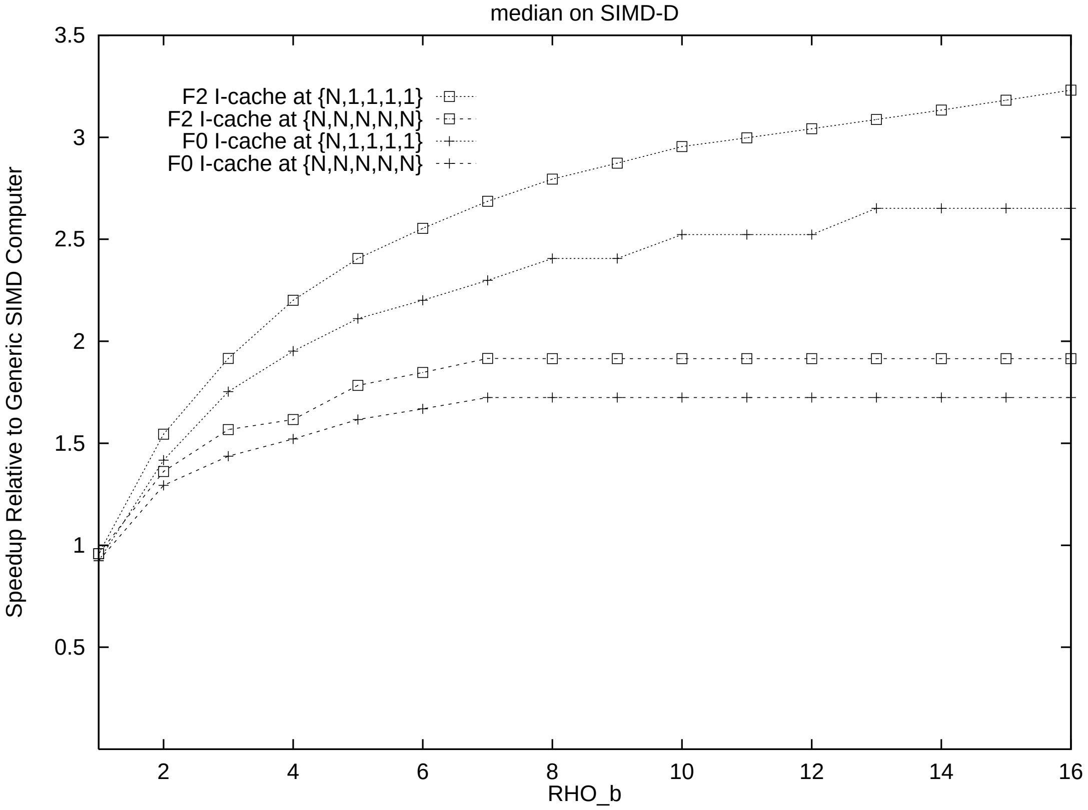


Figure 5.10: I-Cache Speedup Bounds for **median** on SIMD-D

5.8.8 **median**

Figure 5.10 shows the measured speedup bounds for **median**. The required cache size lay in the range 39 . . . 67.

The measured speedups are relatively low because of the complex loop structure of the assembly language program. There are three repeated instruction sequences nested within the outer loop, to sort values into local lists and to manipulate those lists to find the median. Executions of these sequences alternate for small numbers of iterations, so they are frequently re-stored in F_0 and in F_2 I-caches, capable of storing but a single block at a time. **median** is an attractive candidate for I-cache variants, including F_1, F_3, and F_7, capable of storing multiple blocks at once.

5.9 Results Summaries

Section 3.8 concludes that 8 is a conservative estimate for ρ_b in existing SIMD computers, and that ρ_b=8 is not unrealistically large for computers with relatively scalable board-level designs. The range of F_0 speedups for each of the sample programs at ρ_b=8 is shown graphically in Figure 5.11, and the corresponding range of F_2 speedups is shown in Figure 5.12. In each of the summary graphs, the speedup upper bound is obtained at ρ-set $\{8, 1, 1, 1, 1\}$, while the lower bound is obtained at ρ-set $\{8, 8, 8, 8, 8\}$. The plotted point for each program on each graph is the I-cache speedup obtained at ρ-set $\{8, 8, 8, 4, 2\}$.

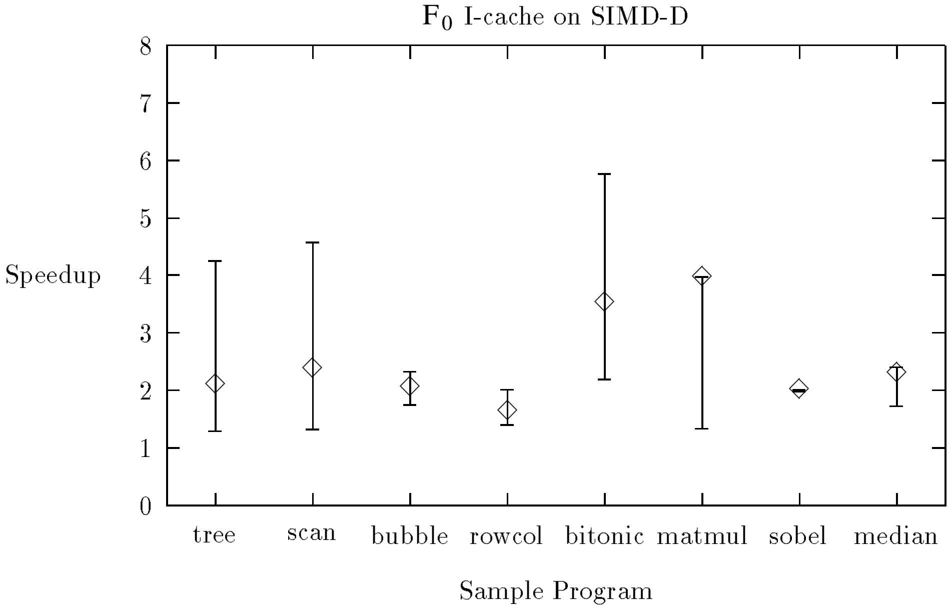


Figure 5.11: Summary of F_0 speedups at ρ_b=8.

5.10 "Simple-Equivalent" Speedups

The quantization evident in the I-cache speedups means that picking values at ρ_b=8 is potentially misleading. Some way of smoothing the results is need, so that it would be possible to summarize an I-cache speedup curve using a single parameter, or perhaps two parameters. It is safer to use the value of a curve that fits the data at ρ_b=8 than it is to use the raw measured point, because the curve's value reflects information from all 16 points, not just 1.

To this end, reconsider the program structure **simple** introduced in Section 4.4:

```
program simple;
  B
  for j = 1 to J do
    A
  end;
end simple;
```

The symbols **A** and **B** in program **simple** denote sequences of instructions. Where the length of sequence **A** is denoted A and the length of sequence **B** is denoted B, the time T_g to run **simple** on a generic SIMD computer is given as

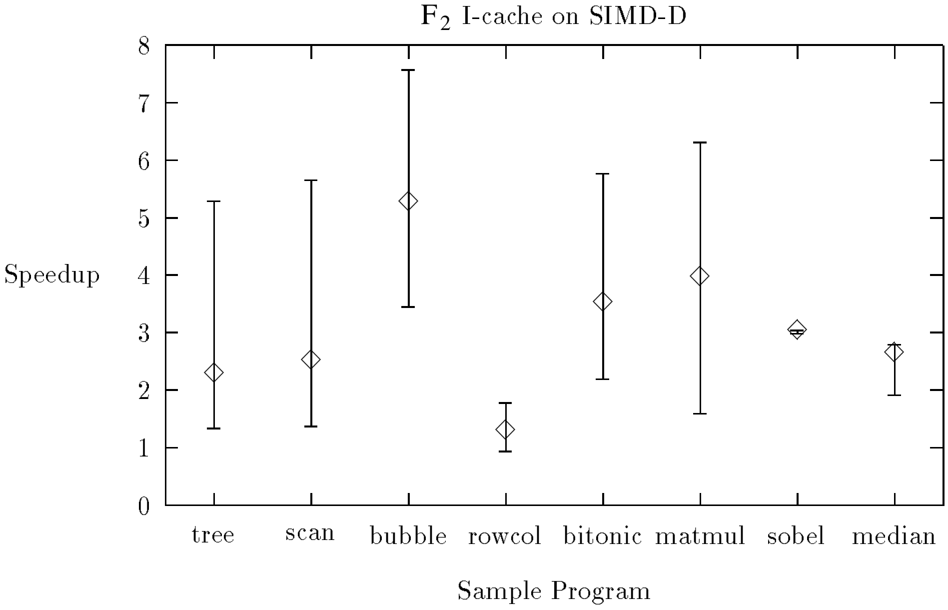


Figure 5.12: Summary of F_2 speedups at ρ_b=8.

$$T_g = B + JA \text{ cycles}$$

The corresponding best time T_c on an ideal I-cache is given as

$$T_c = B + A + J\frac{A}{\rho_b} \text{ cycles}$$

The speedup is the ratio of these times:

$$\begin{aligned}\text{I-cache speedup} &= \frac{T_g}{T_c} \\ &= \frac{B + JA}{B + A + J\frac{A}{\rho_b}}\end{aligned} \tag{5.1}$$

Dividing both top and bottom in Equation 5.1 by $(A + B)$ yields an ugly variant of the speedup equation:

$$\text{I-cache speedup} = \frac{1 + \frac{(J-1)A}{A+B}}{1 + \frac{JA}{(A+B)\rho_b}} \tag{5.2}$$

There is a method to this madness: Let $g=\frac{A}{A+B}$ represent the proportion of the program's instructions that are cachable ($0 \leq g \leq 1$). Then the speedup equation becomes

$$\begin{aligned}\text{I-cache speedup} &= \frac{1 + gJ - g}{1 + g\frac{J}{\rho_b}} \\ &= \frac{(1 + gJ)\,\rho_b}{\rho_b + gJ} - \frac{g\rho_b}{\rho_b + gJ}\end{aligned} \tag{5.3}$$

	F_0 I-cache				F_2 I-cache							
	lower bound		upper bound		lower bound			upper bound			cache size	
Problem	C	sp'up	C	sp'up	C	g	sp'up	C	g	sp'up	(# instrs)	γ
`tree`	3.8	3.3	13.3	5.4	3.9		3.3	14.1		5.5	336	.061
`scan`	1.3	2.0	17.5	5.8	1.3		2.0	18.5		5.9	380	.067
`bubble`	3.8	3.3	52.5	7.1	4.0		3.3	587.2		7.9	540	.090
`rowcol`	2.0	2.4	8.6	4.6	1.1	2.5	1.1	28.0	18.3	2.4	540	.090
`bitonic`	15.9	5.7	64.9	7.2	15.9		5.7	64.9		7.2	1138	.174
`matmul`	8.6	4.6	401.1	7.9	8.7		4.6	859.9		7.9	1196	.182
`sobel`	5.1	3.7	5.3	3.8	5.2		3.8	5.4		3.8	2858	.415
`median`	0.5	1.4	4.0	3.3	0.5		1.4	4.0		3.3	640	.104

Figure 5.13: Summary of I-Cache Speedup Curve Parameters, Speedups at ρ_b=8, and Cache Size at ρ_b=8 on SIMD-A.

For programs for which I-cache speedup can be expected to be greatest, the product gJ, representing the product of the fraction of cachable instructions and the cache block iteration count, should be large. When gJ is very large, then the term $\frac{g\rho_b}{\rho_b+gJ}$ is nearly 0, and the I-cache speedup may be approximated as

$$\text{I-cache speedup} \approx \frac{(1+gJ)\,\rho_b}{\rho_b+gJ} \tag{5.4}$$

If we let C=gJ represent the product of the proportion of repeat instructions and the loop iteration count, then we have the "simple-equivalent" I-cache speedup function:

$$\begin{array}{c}\text{simple-equivalent}\\ \text{I-cache speedup}\end{array} = \frac{(1+C)\,\rho_b}{\rho_b+C} \tag{5.5}$$

Equation 5.5 has a single parameter, C. Fitting Equation 5.5 to the measured I-cache speedup curves yields a single parameter that characterizes the curve. Appendix E plots the measured data and superimposes the result of a least-squares fit of Equation 5.5 over each data set. The fits are excellent, except in the case of F_2 used for **`rowcol`**. Note that the value of Equation 5.5 is never less than 1. In the case of F_2 used for **`rowcol`**, poor static I-cache management causes speedups to be mostly less than 1, so the fits with Equation 5.5 are poor. For that computation, the error term involving g cannot be ignored.

The following set of four tables, one per SIMD computer variant, show the C resulting from the curve fit as shown in Appendix E for both I-cache variants at both bounding ρ-sets, along with the simple-equivalent I-cache speedup corresponding to that value of C. For F_2 I-cache on **`rowcol`**, the parameter g is also shown. Also shown are the cache sizes required to obtain the speedup upper bound. The value of γ in the last column of each table is obtained by substituting the required cache size for N in Equation 4.9, and then substituting the resulting value for δ into Equation 4.6, along with the values for I and A_g appearing in Figure 3.2.

	F$_0$ I-cache				F$_2$ I-cache							
	lower bound		upper bound		lower bound			upper bound			cache size	
Problem	C	sp'up	C	sp'up	C	g	sp'up	C	g	sp'up	(# instrs)	γ
tree	2.0	2.4	11.5	5.1	2.1		2.5	14.1		5.5	92	.081
scan	1.1	1.8	15.1	5.6	1.1		1.8	18.2		5.8	116	.092
bubble	2.9	2.9	13.8	5.4	4.1		3.4	228.2		7.8	156	.108
rowcol	1.6	2.2	5.3	3.8	3.0	2.6	1.0	20.0	13.2	2.2	156	.108
bitonic	8.1	4.5	47.6	7.0	8.1		4.5	47.6		7.0	298	.169
matmul	8.1	4.5	145.2	7.6	8.4		4.6	635.1		7.9	307	.172
sobel	4.7	3.6	4.9	3.7	5.1		3.7	5.3		3.8	720	.347
median	0.5	1.4	3.7	3.2	0.5		1.4	3.9		3.3	184	.120

Figure 5.14: Summary of I-Cache Speedup Curve Parameters, Speedups at ρ_b=8, and Cache Size at ρ_b=8 on SIMD-B.

	F$_0$ I-cache				F$_2$ I-cache							
	lower bound		upper bound		lower bound			upper bound			cache size	
Problem	C	sp'up	C	sp'up	C	g	sp'up	C	g	sp'up	(# instrs)	γ
tree	0.7	1.6	8.9	4.7	0.8		1.6	14.1		5.5	32	.183
scan	0.7	1.6	10.5	5.0	0.7		1.6	16.5		5.7	41	.196
bubble	1.6	2.2	3.6	3.2	4.3		3.4	81.5		7.4	48	.206
rowcol	0.9	1.7	2.3	2.6	3.0	2.7	0.9	10.0	6.7	1.9	55	.215
bitonic	2.9	2.9	29.0	6.5	2.9		2.9	29.0		6.5	86	.258
matmul	4.9	3.7	26.2	6.4	5.9		4.0	201.4		7.7	48	.206
sobel	3.4	3.1	3.5	3.1	4.6		3.6	4.7		3.6	158	.358
median	0.8	1.6	2.1	2.5	0.9		1.7	2.6		2.7	59	.221

Figure 5.15: Summary of I-Cache Speedup Curve Parameters, Speedups at ρ_b=8, and Cache Size at ρ_b=8 on SIMD-C.

	F$_0$ I-cache				F$_2$ I-cache							
	lower bound		upper bound		lower bound			upper bound			cache-size	
Problem	C	sp'up	C	sp'up	C	g	sp'up	C	g	sp'up	(# instrs)	γ
tree	0.4	1.3	7.7	4.4	0.4		1.3	13.8		5.4	20	.005
scan	0.4	1.3	9.5	4.8	0.4		1.3	16.9		5.8	27	.005
bubble	1.0	1.8	1.9	2.3	4.8		3.6	51.8		7.1	28	.005
rowcol	0.5	1.4	1.4	2.0	3.0	2.7	0.9	7.0	4.7	1.8	32	.005
bitonic	1.7	2.2	20.7	6.0	1.7		2.2	20.7		6.0	50	.006
matmul	0.4	1.3	7.7	4.4	0.4		1.3	13.8		5.4	10	.004
sobel	1.3	2.0	1.4	2.0	3.1		3.0	3.3		3.0	35	.005
median	0.9	1.7	2.0	2.4	1.1		1.8	2.7		2.8	40	.005

Figure 5.16: Summary of I-Cache Speedup Curve Parameters, Speedups at ρ_b=8, and Cache Size at ρ_b=8 on SIMD-D.

5.11 Maximum I-Cache Speedup: F_7 Estimates

Looking at the fairly large speedups shown in Figures 5.3 through 5.10, one wonders how the speedups obtained using the simple I-cache variants compare with the maximum possible speedup for each of the sample problems.

The best possible I-cache speedup would be obtained using a large, complex multi-port I-cache variant, wherein all repeat instructions fit in cache memory, all iteration is controlled within the PE chip, and some of the cache store time for instructions overlaps with execution of other cache blocks. Because of the characteristic simplicity of their loop structures, the sample programs do not present much opportunity to exploit prefetching. For the sample programs, nearly the maximum possible I-cache speedup would be obtained using an F_7 I-cache variant that has sufficient capacity to store at one time all repeated instruction sequences. As discussed in Section 4.2, F_7 is the most complex member of the F-family of single-port I-cache variants. The program-control component of an F_7 I-cache may be as complex as the system controller's program-control component. An F_7 I-cache is able to store and sequence entire programs, whose executions may involve loop nesting and data-dependent branching. Unlike the very simple F_0 and F_2 I-cache variants, the F_7 I-cache variant yielding the maximum speedup may well occupy substantial chip area inside the PE chip.

Estimates for F_7 speedups are obtained through a static analysis of assembly language programs for the sample problems. The method is to run each program directly through a modified version of the assembler, without any re-programming for I-cache. The modified assembler schedules all repeated instruction sequences as if they were in cache, assuming an arbitrarily large cache size. Then the run time of the resulting computation is estimated by cycle counting. A repeated instruction sequence contributes an amount of time equal to the loop iteration count times the length of the sequence divided by ρ_b, plus an additional amount of time equal to the number of instructions in the sequence which is required to store the sequence in cache. This method ignores the re-storing that needs to be performed when cache size is exceeded, it ignores all quantization effects, and it ignores the time spent globally broadcasting cache-control instructions that are added to programs with I-cache.

It is important to be aware of the significant difference between these F_7 speedup estimates and the speedup measurements that are the basis of I-cache evaluation. The F_0 and F_2 speedup measurements are based on detailed designs, they account for quantization effects and cache-control overheads, and most importantly, they are simulated to verify their correctness. The F_7 speedups are compiler estimates only, rather than measurements taken from the simulator.

Bearing this caveat in mind, it is interesting to compare the speedup measurements obtained for the simple I-cache variants against estimates for the best possible F_7 speedups. Figures 5.17 through 5.24 show estimates for F_7 speedups at the ρ-sets that give upper and lower bounds. The F_7 speedup estimates are plotted along with F_2 speedup curves, except for the programs with data-dependent iteration counts. For those programs, the static management of F_2 I-cache amounts to an algorithm change, which change makes comparison between F_7 and F_2 unfair. For the programs with data-dependent branching, the F_7 speedup estimates are plotted with the F_0 speedup measurements for comparison purposes.

Where iteration counts are moderate, as for **`tree`** and **`scan`**, F_7 speedup estimates tail over significantly, and F_2 achieves close to this maximum speedup.

For the data-dependently branching programs, including **`bubble`**, **`rowcol`**, and **`sobel`**, the estimate for the best F_7 speedup is low. The reason is that each iteration of the inner loop includes a response operation, whose time is typically long and not appreciably shortened with I-cache. When a significant fraction of the computation cannot be made faster with I-cache, I-cache speedups are low. The difference between the F_0 speedups and the F_7 estimates for these problems is due to quantization and the cache-control overheads.

The three remaining problems, **`bitonic`**, **`matmul`**, and **`median`**, all have high iteration counts and no data-dependent branching. These are ideal conditions for large I-cache speedups, and indeed, the corresponding estimates for F_7 speedup are high. For the first two of these programs, F_2 achieves roughly 75% of the F_7 estimate, suffering from quantization and time spent in cache-control instructions. For **`median`**, however, F_2 speedup is well below the maximum. This disparity highlights the principal shortcoming of an F_2 I-cache variant, namely that it contains only one block at a time. The assembly language program for **`median`** (shown in Figure D.8) contains several repeated instruction sequences whose executions alternate during the computation. A single-block I-cache variant such as F_2 must re-store each cache block before it can be used. The small iteration counts for which each cache block is repeated per activation in **`median`** mean that the time spent re-storing is not amortized over a large number of iterations. **`median`** is one program which demands an F_3 or higher I-cache variant, capable of storing multiple cache blocks at once.

Of course, the maximum F_7 speedup is determined by program characteristics. The F_7 speedup estimates therefore provide one way of characterizing programs. For example, the spread between the upper and lower speedup bounds for F_7 apparent in Figures 5.17 through 5.24 provides a rough measure of the MCS-intensiveness of each program. With the exception of **`sobel`**, all of the sample programs are MCS-intensive, as illustrated by large spreads between upper and lower F_7 speedup bounds estimates. In light of this observation, it is somewhat surprising that the simple I-cache speedup lower bounds lie in the range of 30% to 100% for these programs when ρ_b=8.

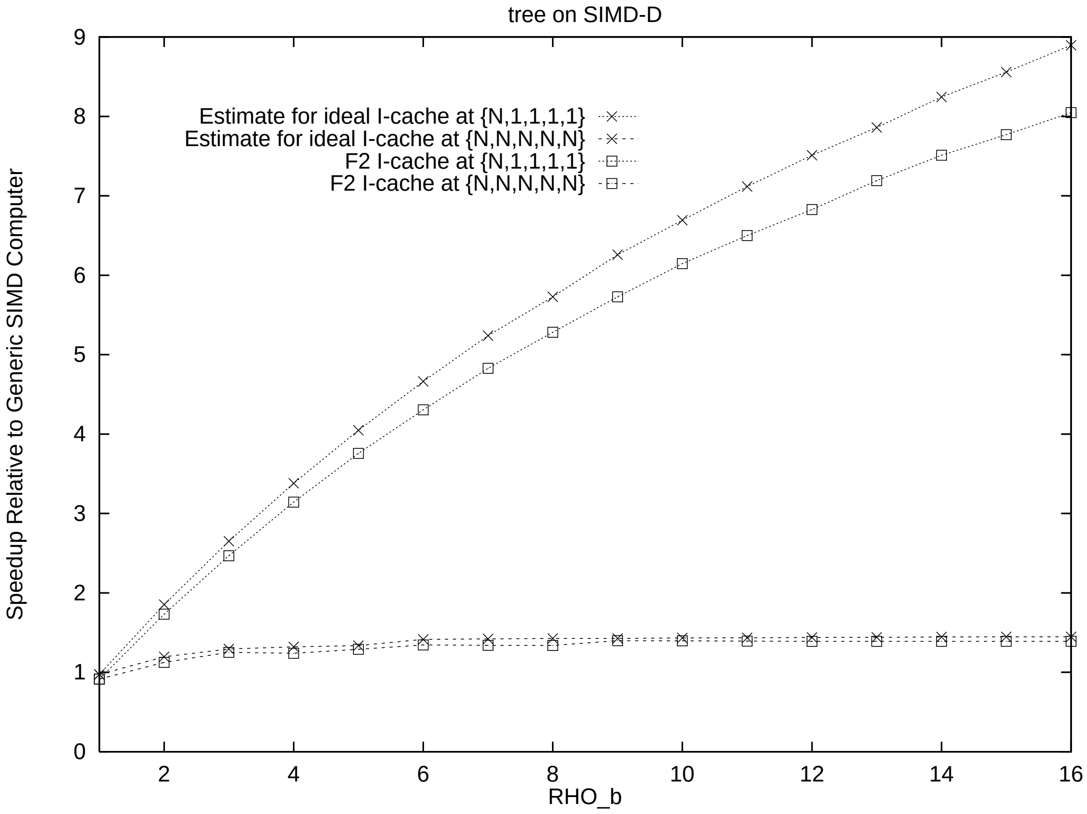


Figure 5.17: Ideal I-Cache Compared with F_2 for **`tree`** on SIMD-D

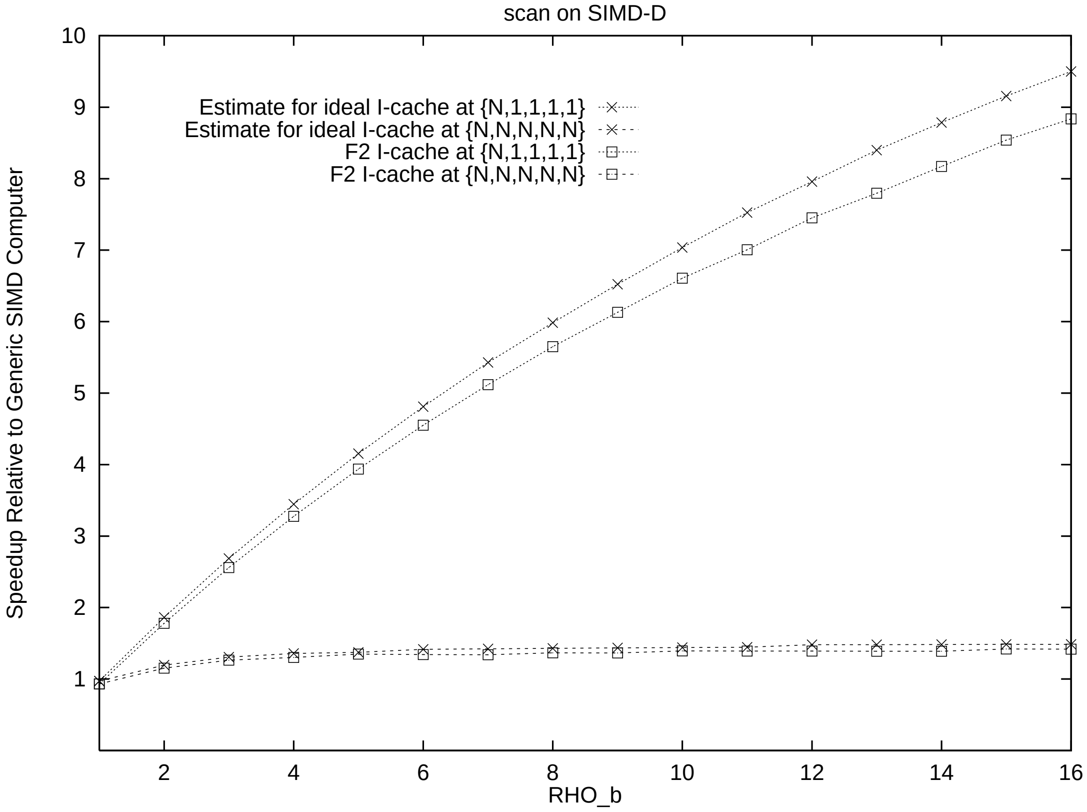


Figure 5.18: Ideal I-Cache Compared with F_2 for **`scan`** on SIMD-D

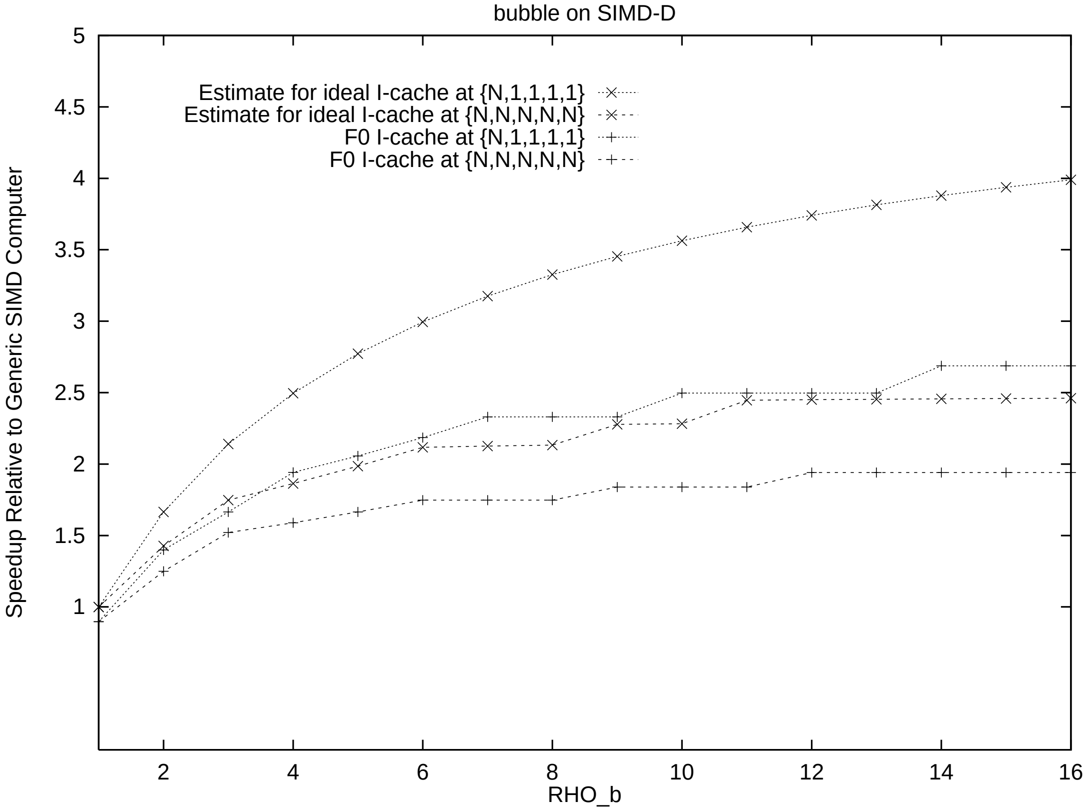


Figure 5.19: Ideal I-Cache Compared with F_0 for **`bubble`** on SIMD-D

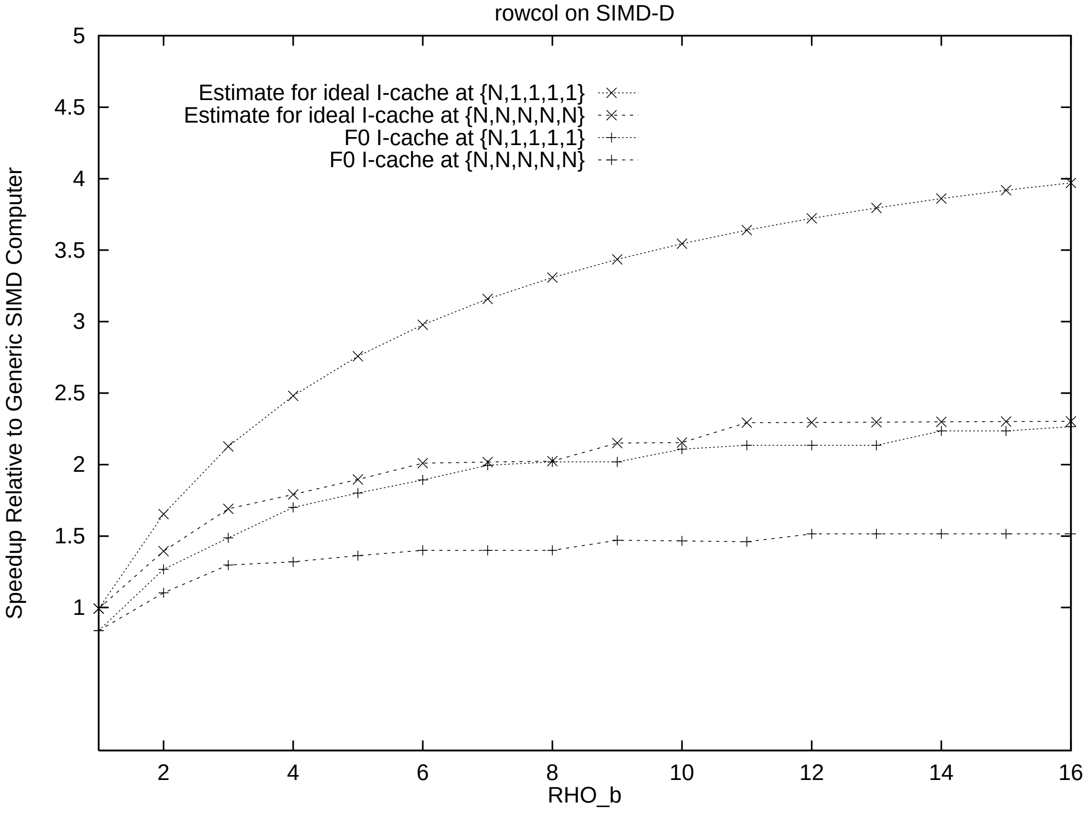


Figure 5.20: Ideal I-Cache Compared with F_0 for **`rowcol`** on SIMD-D

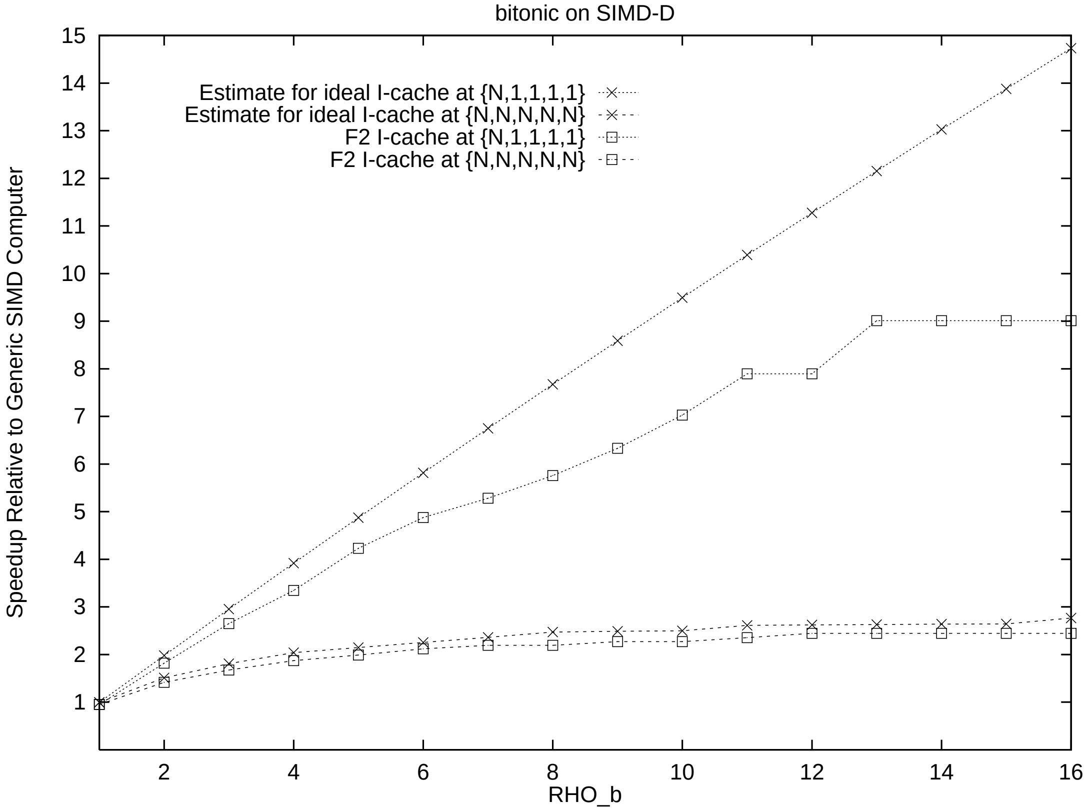


Figure 5.21: Ideal I-Cache Compared with F$_2$ for **`bitonic`** on SIMD-D

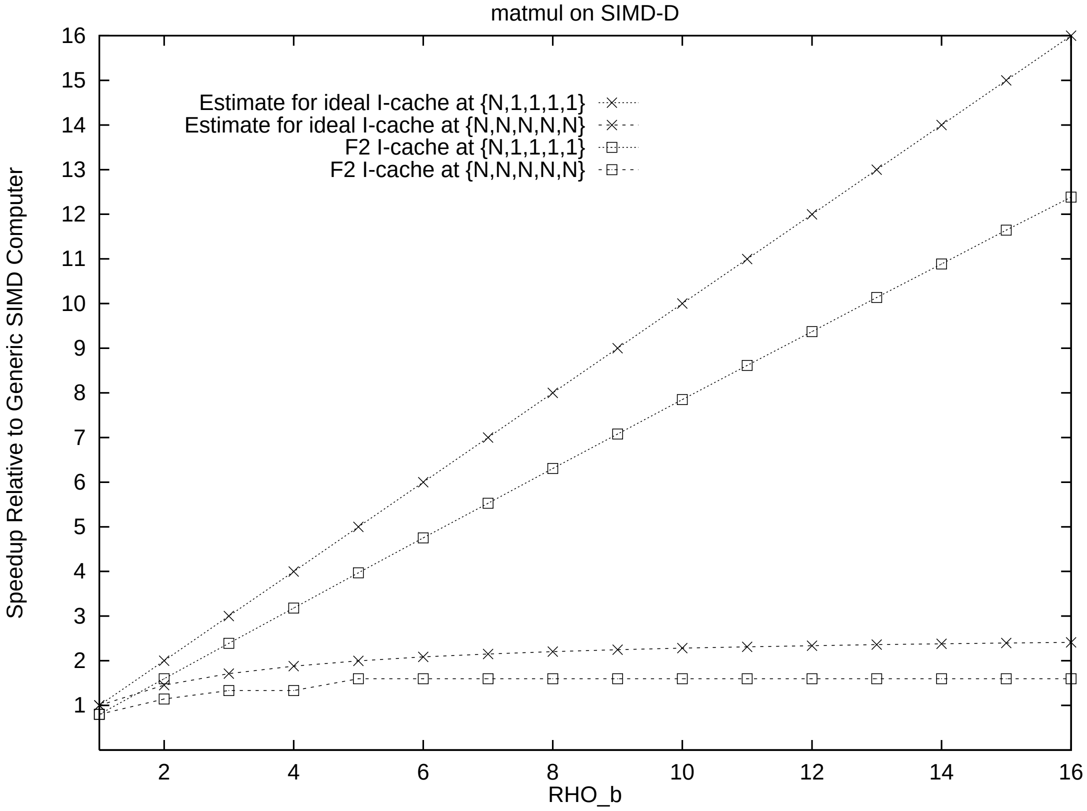


Figure 5.22: Ideal I-Cache Compared with F$_2$ for **`matmul`** on SIMD-D

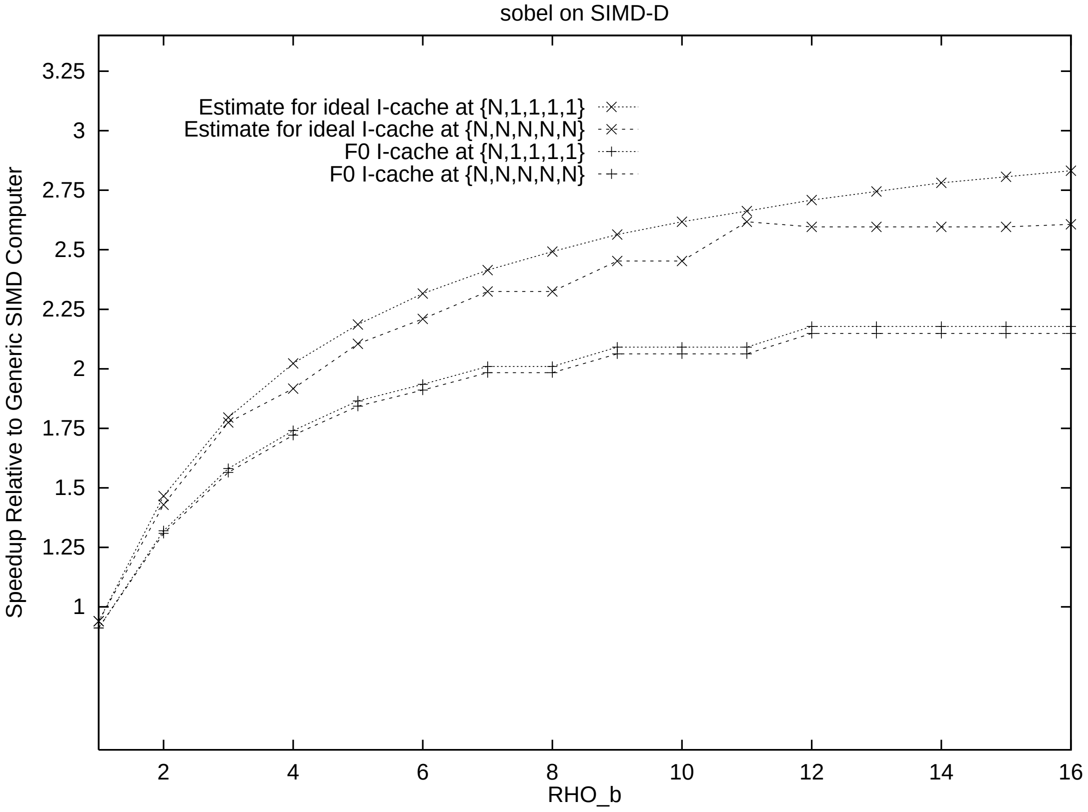


Figure 5.23: Ideal I-Cache Compared with F_0 for **sobel** on SIMD-D

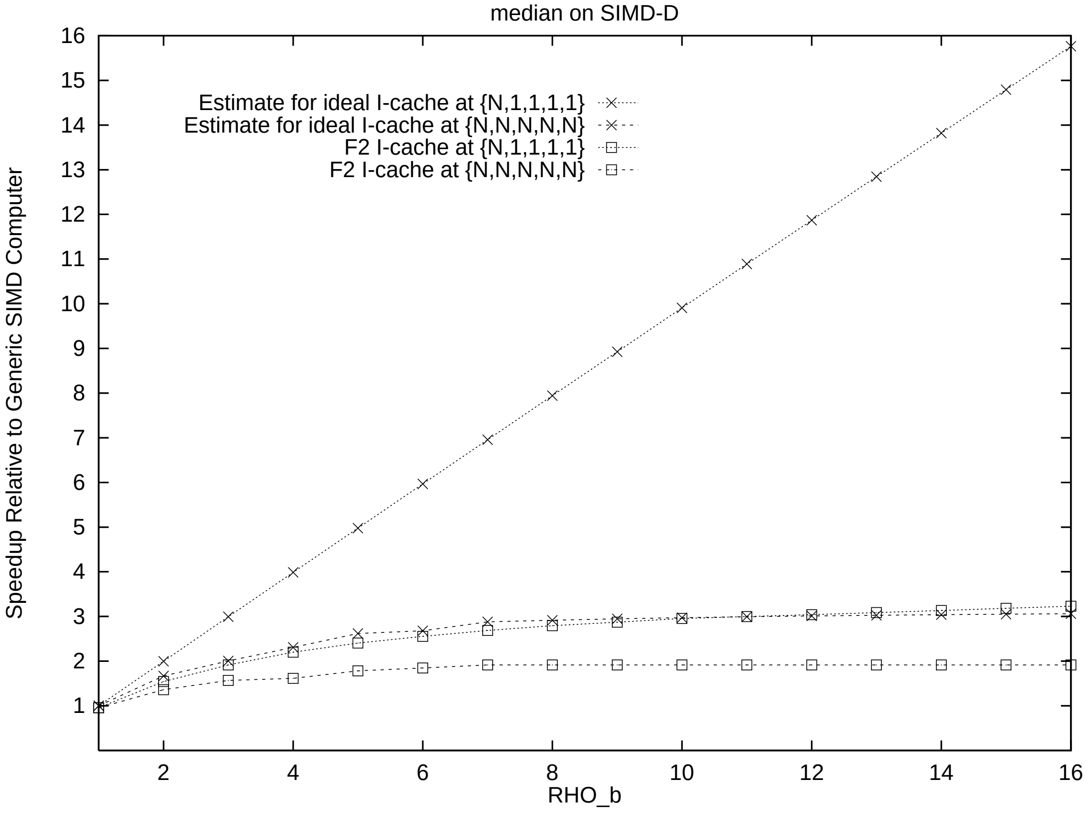


Figure 5.24: Ideal I-Cache Compared with F_2 for **median** on SIMD-D

5.12 Sensitivity of I-Cache Speedup to Inter-PE Communication

How sensitive is I-cache speedup to inter-PE communication intensiveness? The way to investigate the effect of inter-PE communication intensiveness is to vary the stepcount parameters of inter-PE communication operations for each of the subject programs. The stepcount of an operation is the number of subsystem clock cycles taken to perform the operation.

A reasonable expectation is that as inter-PE communication stepcounts increase, a greater proportion of the total computation time is spent in inter-PE communication. Because MCS operations tend not to be instruction delivery-bound in the basis computer used in I-cache evaluations, I-cache speedup would drop off as the inter-PE communication intensiveness drops off.

The actual effect of increased inter-PE communication intensiveness is surprisingly subtle. Figures 5.25 through 5.32 show how I-cache speedups vary versus the number of clock cycles required to perform inter-PE communication operations at a few different ρ-sets. The graphs show that sometimes the I-cache speedup actually increases with the inter-PE communication intensiveness.

The reason for this surprising result is that the effect of increased inter-PE communication on the I-cached SIMD computation is sensitive to the ρ-set. Increasing the inter-PE communication stepcount by 1 increases the generic SIMD computation time by one system clock cycle per each inter-PE communication operation performed in the computation. However, the increase in the time of the I-cached computation is $\frac{\rho_c}{\rho_b}$ system clock cycles per communication operation, because the inter-PE communication subsystem clock rate is $\frac{\rho_b}{\rho_c}$ times greater than the system clock rate. When inter-PE communication steps are faster than global instruction broadcasts, increasing the proportion of inter-PE communication in a computation actually increases the I-cache speedup. For example, when $\frac{\rho_b}{\rho_c}=8$, as at ρ-set $\{8,8,1,1,1\}$, each inter-PE communication step occurs 8 times faster in the I-cached SIMD computation than in the generic SIMD computation. At the other extreme in relative rates, when inter-PE communication occurs at the same rate as global instruction broadcast, the time added by increased inter-PE communication is roughly the same in the generic and I-cached SIMD computations. For example, when $\frac{\rho_b}{\rho_c}=1$, as at ρ-set $\{8,8,8,8,8\}$, each inter-PE communication step occurs at the same rate in the I-cached SIMD computation as in the generic SIMD computation. While flow-dependencies may allow I-cache to overlap other operations with the inter-PE communication, the tendency of increased inter-PE communication is to add the same amount of time to generic and I-cached computations, thus reducing the I-cache speedup.

The graphs in Figures 5.25 through 5.32 plot I-cache speedup versus inter-PE communication stepcount measured for machine variant SIMD-D. The graphs illustrate the ρ-set dependence of the sensitivity of I-cache speedup to inter-PE communication intensiveness. The stepcounts for most of the computations vary from 1 to 10, which, for example, reflects a degree of PE pin time-sharing for neighbor communications on a regular grid. On the computations using routed inter-PE communication, the stepcount for an arbitrary permutation requires $O(\log P)$ steps on P PEs using a log-diameter communication topology such as a hypercube. The sample computations using routed inter-PE communications all use 1×2^{20} PEs, for a nominal inter-PE communication stepcount of 20. The graphs for these computations show stepcounts ranging from 15 to 25.

One curve that apparently departs from the general description given above is the F_2 speedup for **`bubble`** at ρ-set $\{8,8,1,1,1\}$. That curve, shown in Figure 5.27, rises only slightly to the right, at a value just under 8. This behavior really is not surprising in light of the fact that a factor of ρ_b is the maximum possible I-cache speedup, and that factor is already nearly attained for **`bubble`**. At ρ-set $\{8,8,1,1,1\}$, each added inter-PE communication step goes 8 times faster on the I-cache computation. Each added communication step has nearly the same speedup as the rest of the computation, so the added steps barely affect the overall I-cache speedup.

The marked quantization effect for **`matmul`** is apparent in the F_0 I-cache speedup at ρ-set $\{8,8,1,1,1\}$ shown in Figure 5.30. Each time the inter-PE communication stepcount increases at ρ-set $\{8, 8, 1, 1, 1\}$, the time to execute the inner loop's cache block increases by 1 PE clock cycle. The F_0 execution time is constant as the stepcount increases from 1 to 7, and the I-cache speedup arises because the generic SIMD computation time increases steadily over that range. However, when the stepcount increases from 7 to 8, the F_0 cache block's duration becomes just long enough to require another system clock cycle, causing a jump in the execution time and a corresponding marked decrease in speedup.

Most of the speedups curves indicate that I-cache speedup is far more sensitive to ρ-sets than it is to inter-PE communication stepcounts.

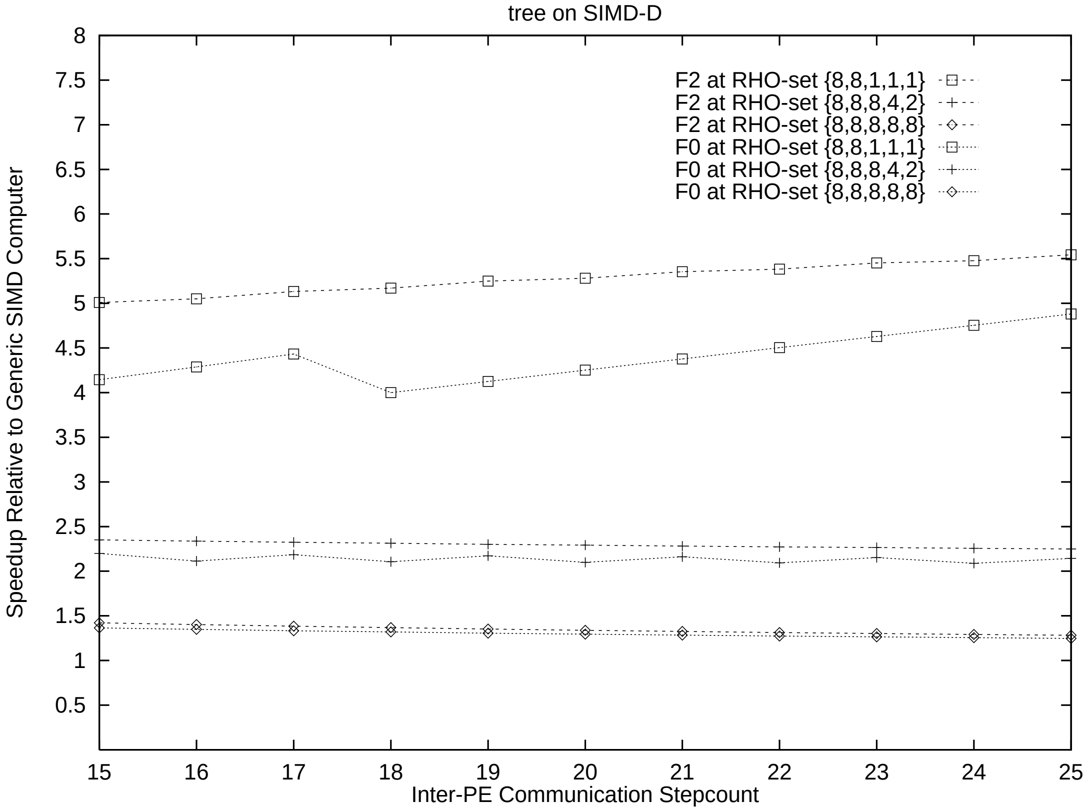


Figure 5.25: I-Cache Speedups v. Inter-PE Communication Stepcounts for **`tree`**

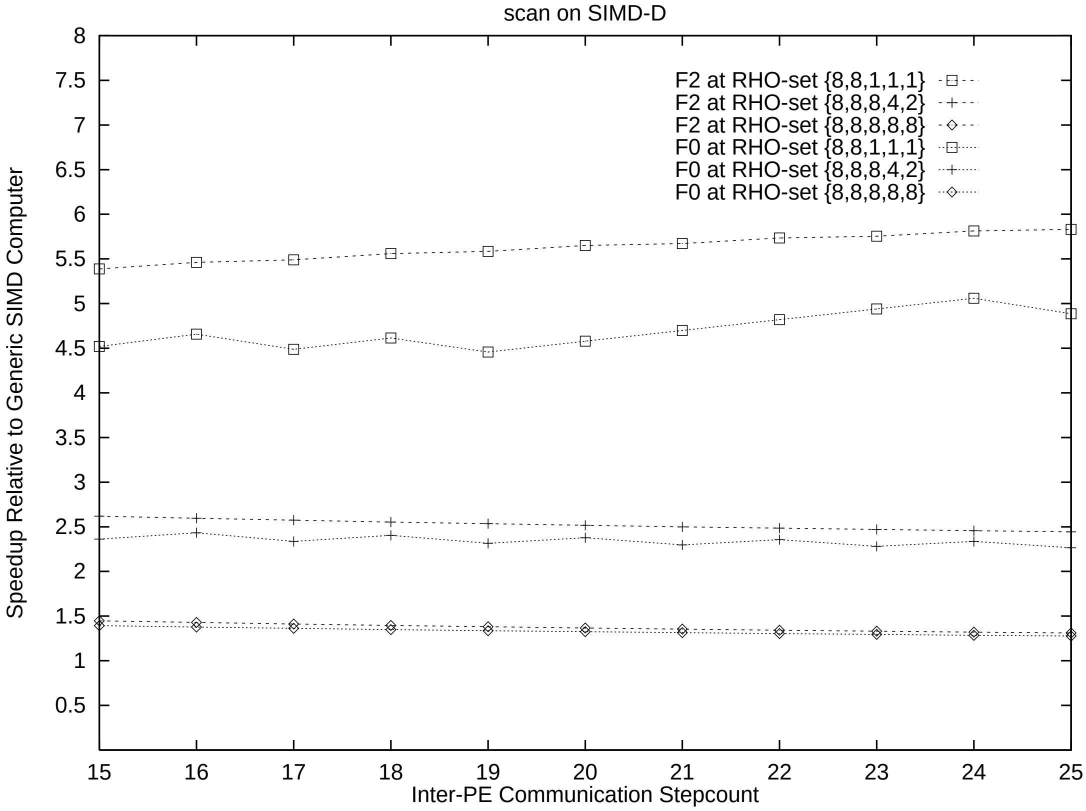


Figure 5.26: I-Cache Speedups v. Inter-PE Communication Stepcounts for **`scan`**

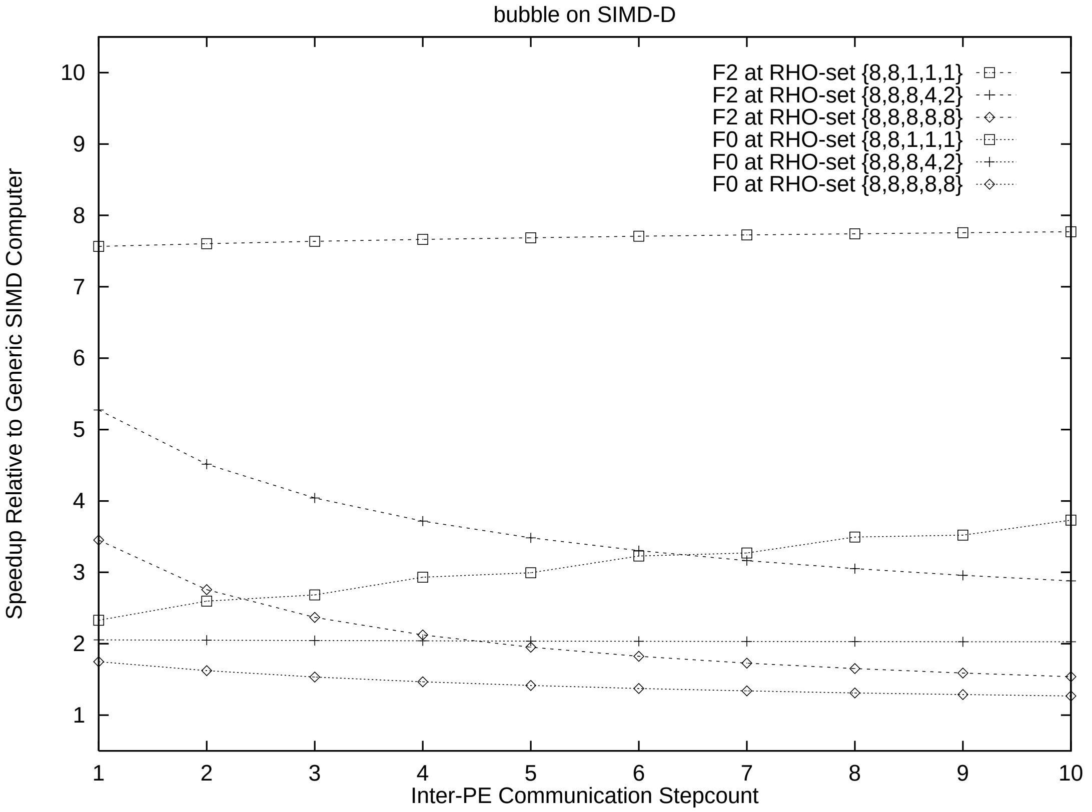


Figure 5.27: I-Cache Speedups v. Inter-PE Communication Stepcounts for **`bubble`**

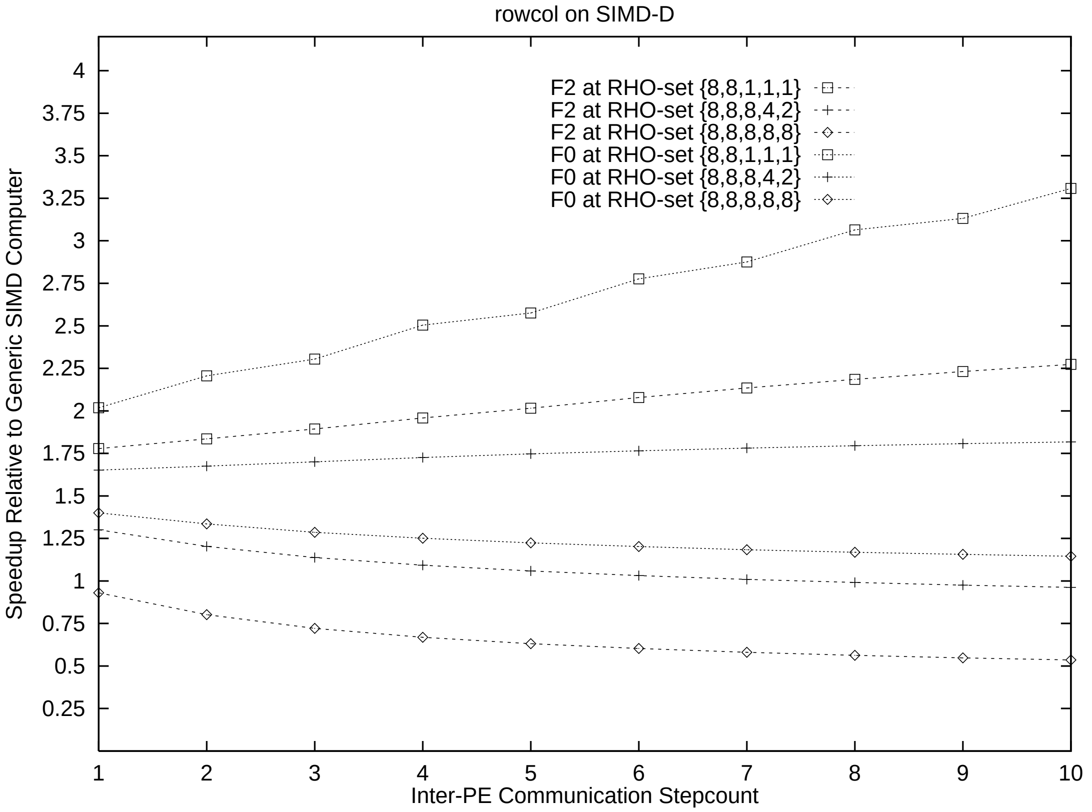


Figure 5.28: I-Cache Speedups v. Inter-PE Communication Stepcounts for **`rowcol`**

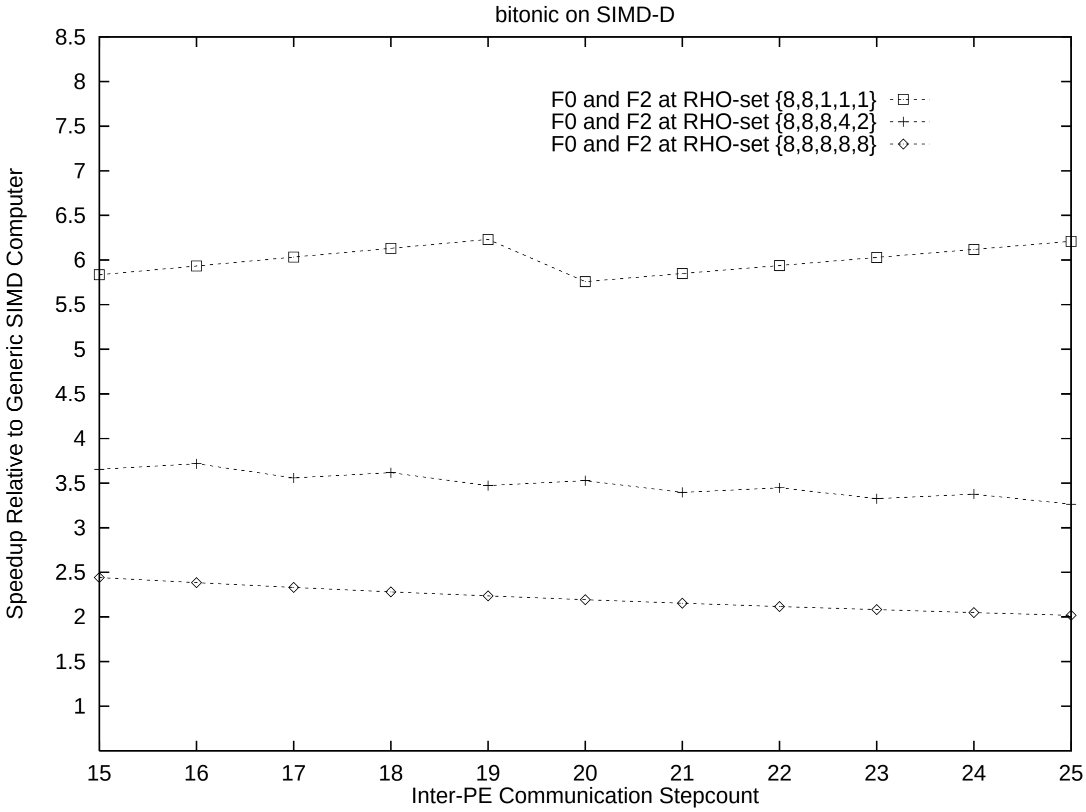


Figure 5.29: I-Cache Speedups v. Inter-PE Communication Stepcounts for **`bitonic`**

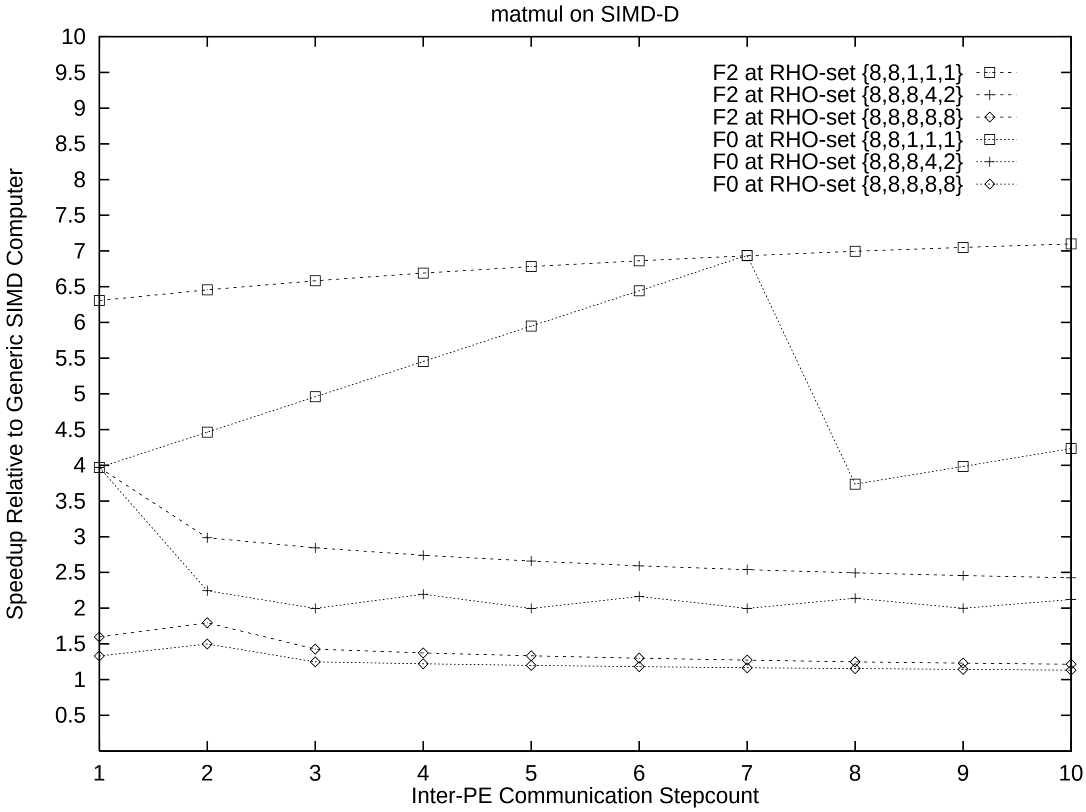


Figure 5.30: I-Cache Speedups v. Inter-PE Communication Stepcounts for **`matmul`**

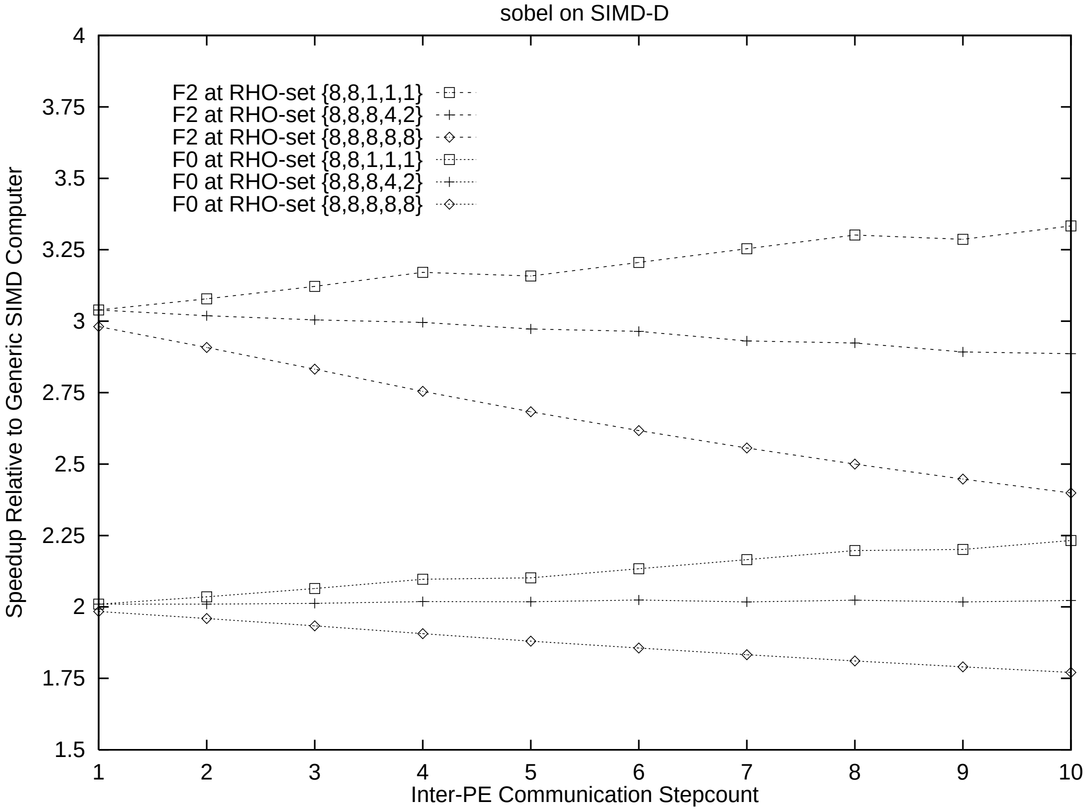


Figure 5.31: I-Cache Speedups v. Inter-PE Communication Stepcounts for **`sobel`**

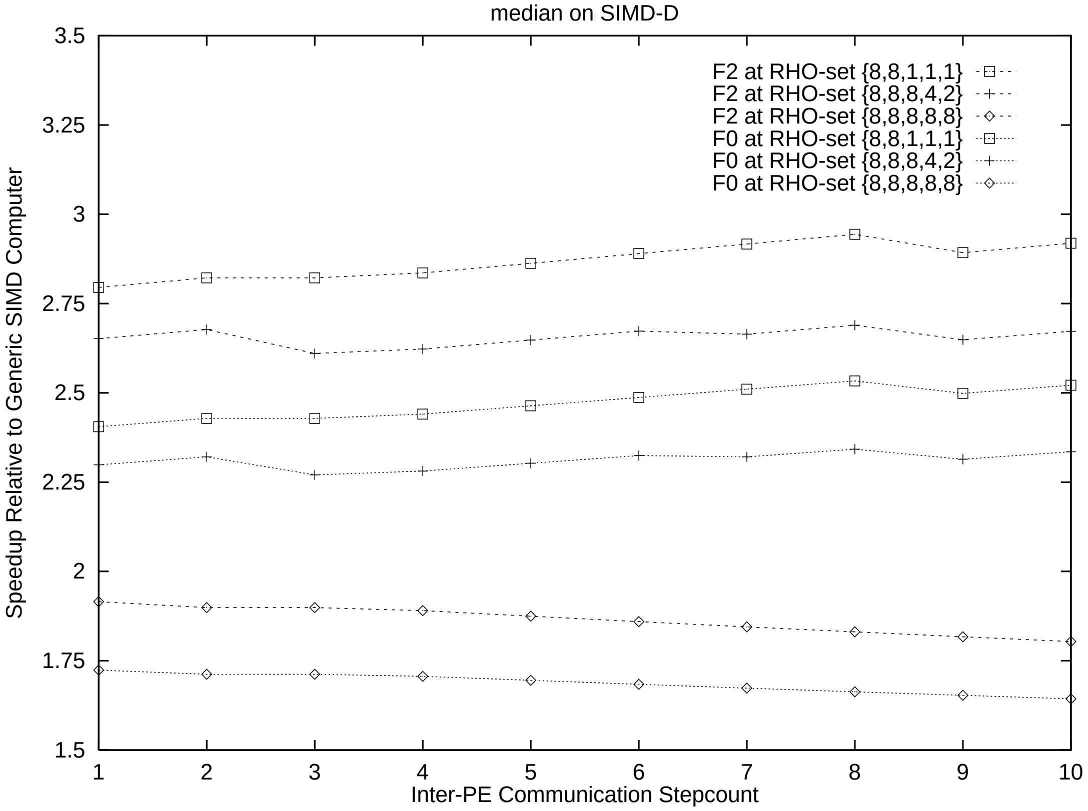


Figure 5.32: I-Cache Speedups v. Inter-PE Communication Stepcounts for **`median`**

5.13 Sensitivity of I-Cache Speedup to Local External Memory Access

How sensitive is I-cache speedup to local external memory access intensiveness? This question is addressed using the same technique used for inter-PE communication sensitivity. By varying the stepcounts of local external memory access instructions, it is possible to vary the memory intensiveness of a sample program. Figures 5.33 through 5.40 show the results.

Of the eight sample programs, only three use local external memory. For the rest of the programs, the PE register file is sufficiently large to accommodate all of the program variables. Of course, the I-cache speedups of the programs that do not use local external memory are flat versus local external memory stepcount. These five include **`tree`**, **`bubble`**, **`rowcol`**, **`bitonic`**, and **`sobel`**.

Of the three programs that do use local external memory, **`scan`** uses it only lightly, and its I-cache speedup is therefore barely affected as memory stepcounts change. **`matmul`** and **`median`** make heavy use of memory. **`matmul`** shows the effects of quantization.

As for inter-PE communication, the effect of increased local external memory stepcounts on the I-cached SIMD computation is sensitive to the ρ-set. Increasing the local external memory access stepcount by 1 increases the generic SIMD computation time by one system clock cycle per each memory access performed in the computation. However, the increase in the time of the I-cached computation is $\frac{\rho_l}{\rho_b}$ system clock cycles per access, because the local external memory subsystem clock rate is $\frac{\rho_b}{\rho_l}$ times greater than the system clock rate. When memory access steps are faster than global instruction broadcasts, increasing the proportion of memory accesses in a computation actually increases the I-cache speedup. For example, when $\frac{\rho_b}{\rho_l}=8$, as at ρ-set $\{8, 8, 1, 1, 1\}$, each local external memory access step occurs 8 times faster in the I-cached SIMD computation than in the generic SIMD computation. At the other extreme in relative rates, when memory access occurs at the same rate as global instruction broadcast, the time added by increasing the number of memory access steps is roughly the same in the generic and I-cached SIMD computations. For example, when $\frac{\rho_b}{\rho_l}=1$, as at ρ-set $\{8, 8, 8, 8, 8\}$, each memory access step occurs at the same rate in the I-cached SIMD computation as in the generic SIMD computation. While flow-dependencies may allow I-cache to overlap other operations with the memory access, the tendency of increased memory usage is to add the same amount of time to generic and I-cached computations, thus reducing the I-cache speedup. These effects are most pronounced in the I-cache speedups of **`median`** shown in Figure 5.40.

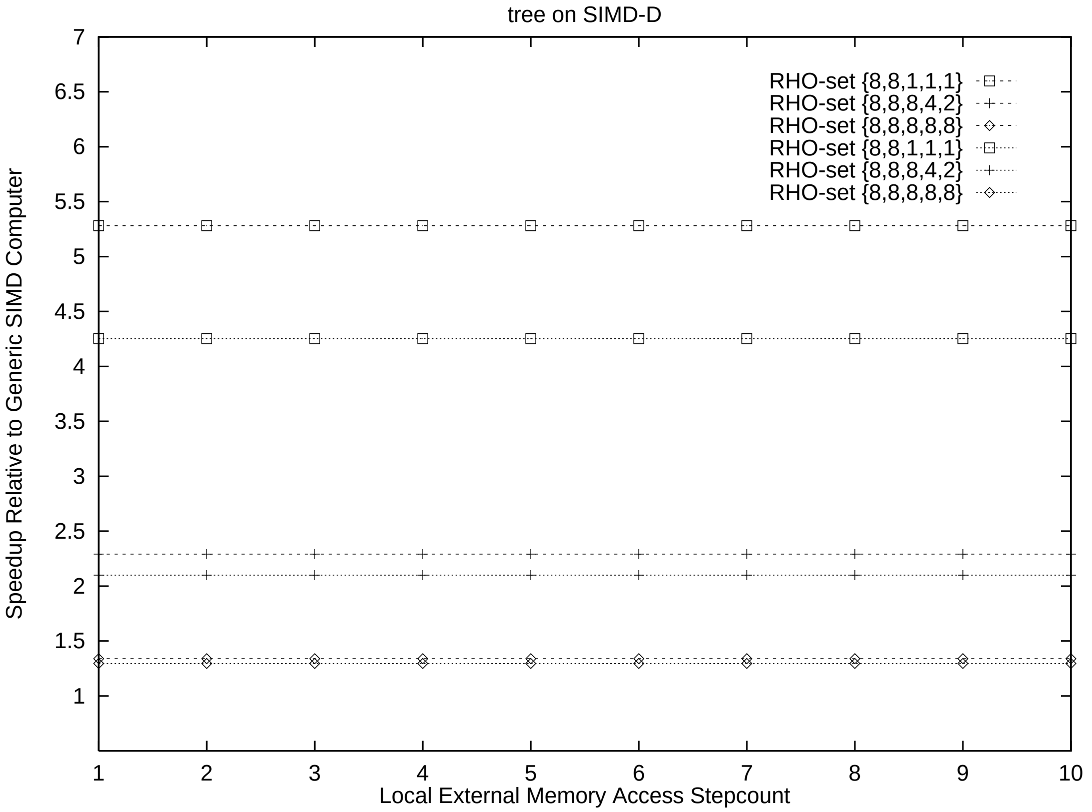


Figure 5.33: I-Cache Speedups v. Local External Memory Access Stepcounts for **`tree`**

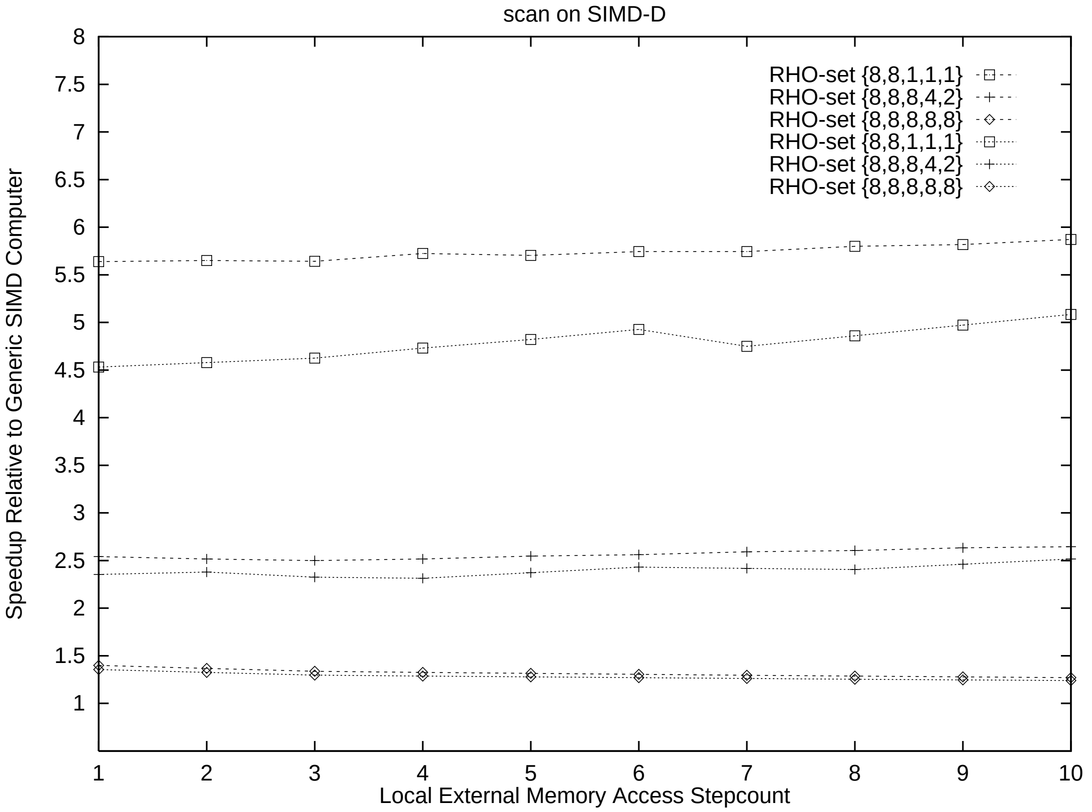


Figure 5.34: I-Cache Speedups v. Local External Memory Access Stepcounts for **`scan`**

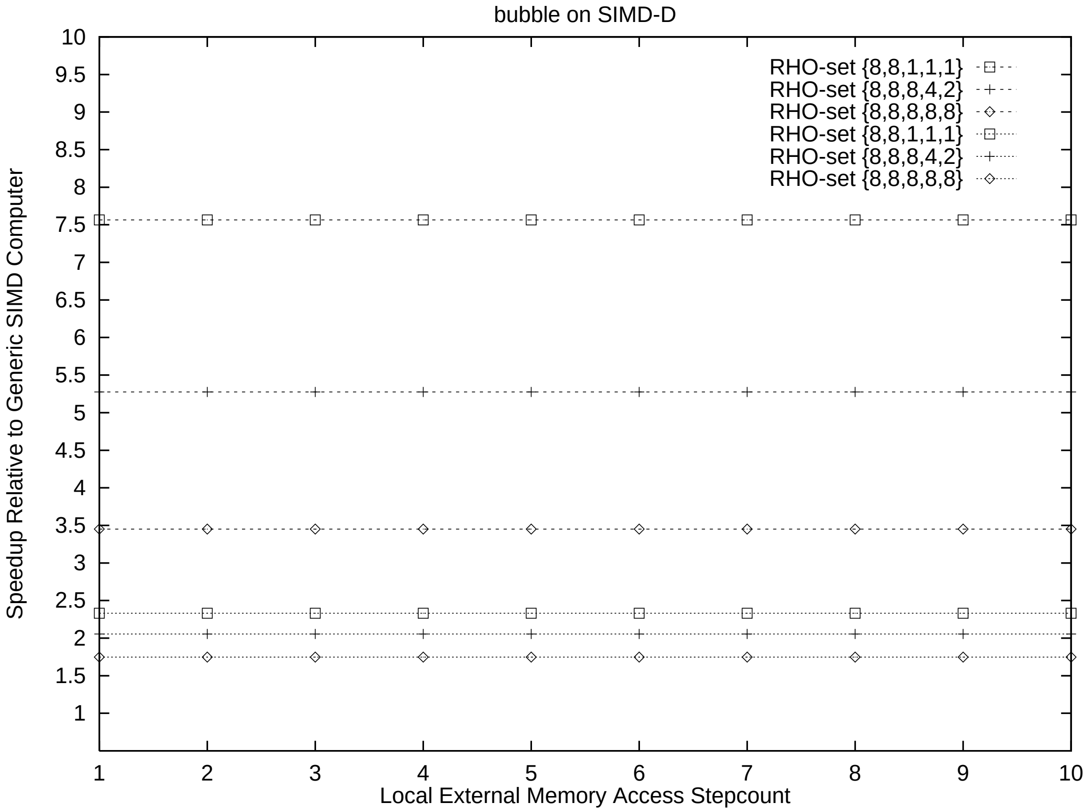


Figure 5.35: I-Cache Speedups v. Local External Memory Access Stepcounts for **`bubble`**

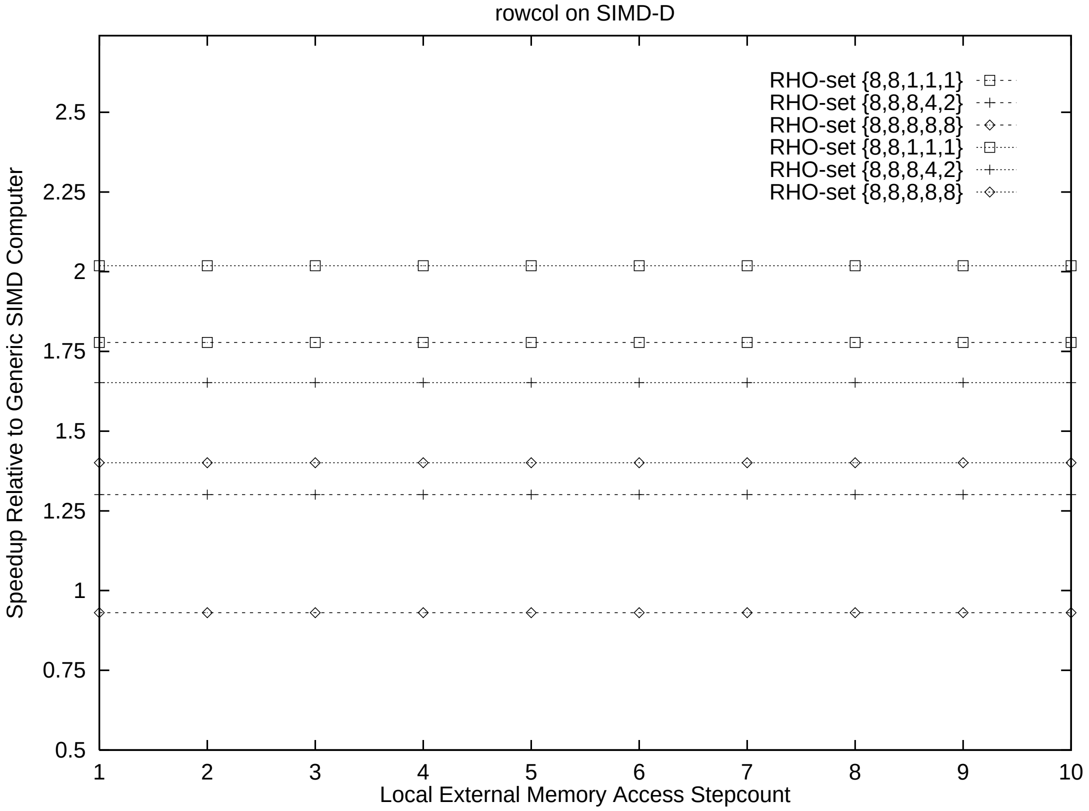


Figure 5.36: I-Cache Speedups v. Local External Memory Access Stepcounts for **`rowcol`**

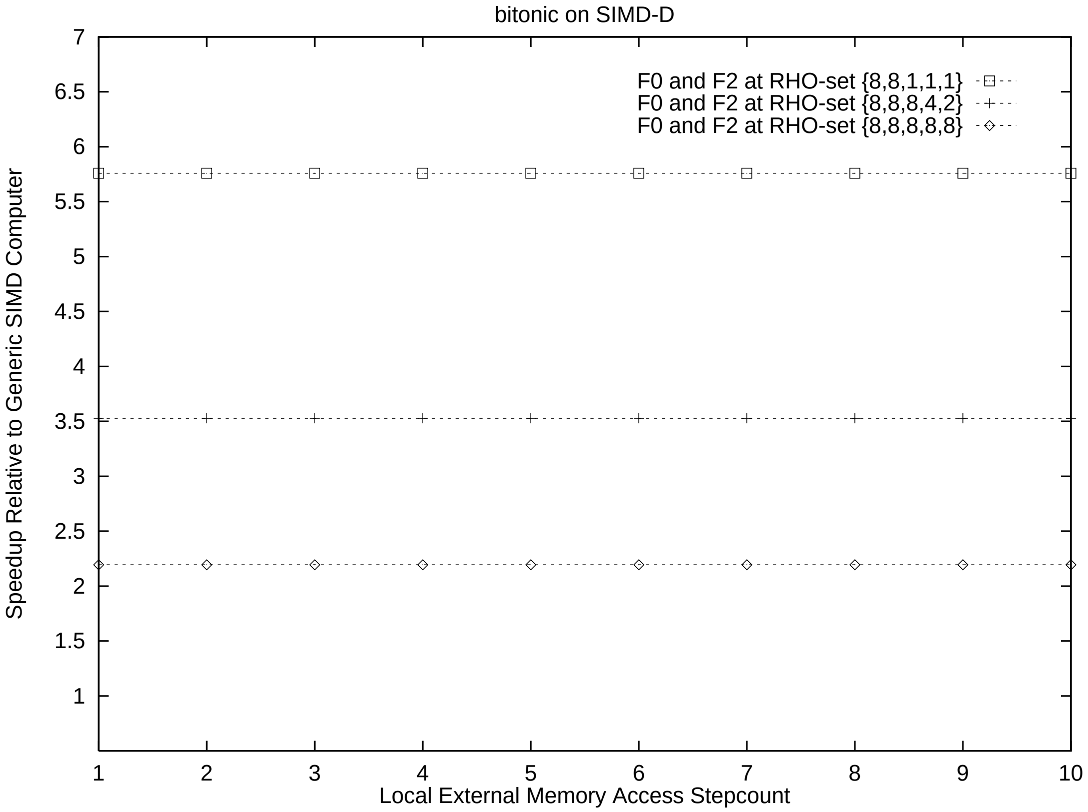


Figure 5.37: I-Cache Speedups v. Local External Memory Access Stepcounts for **`bitonic`**

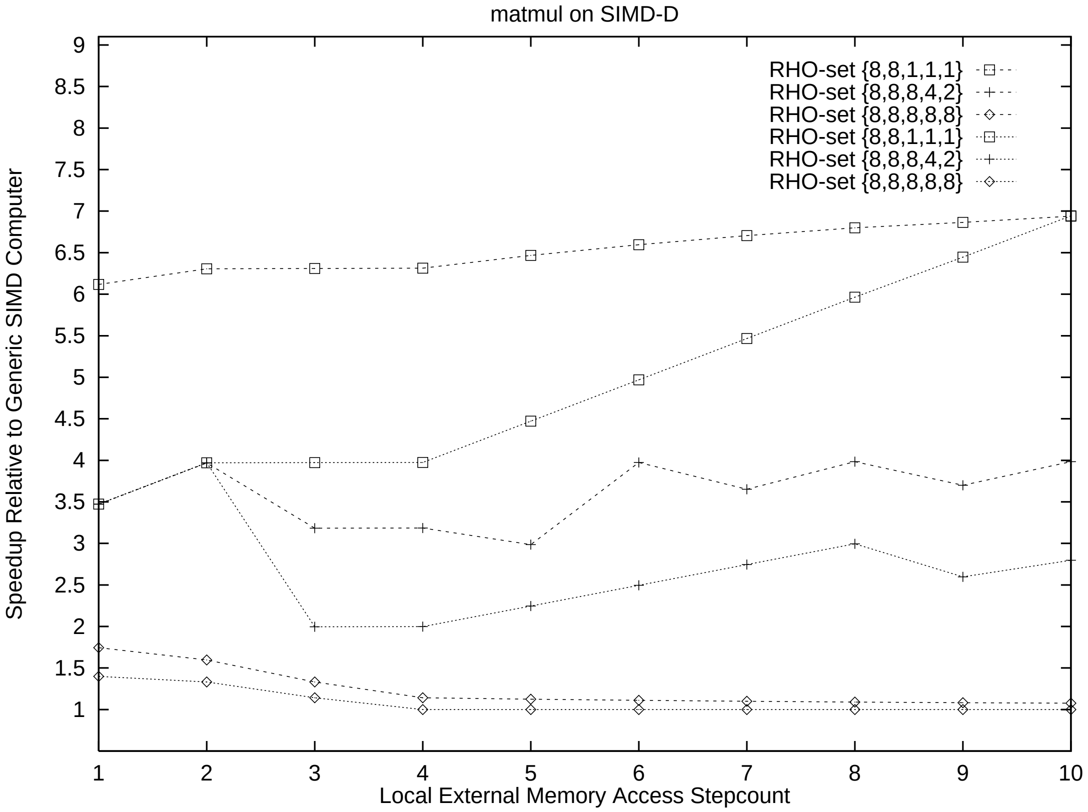


Figure 5.38: I-Cache Speedups v. Local External Memory Access Stepcounts for **`matmul`**

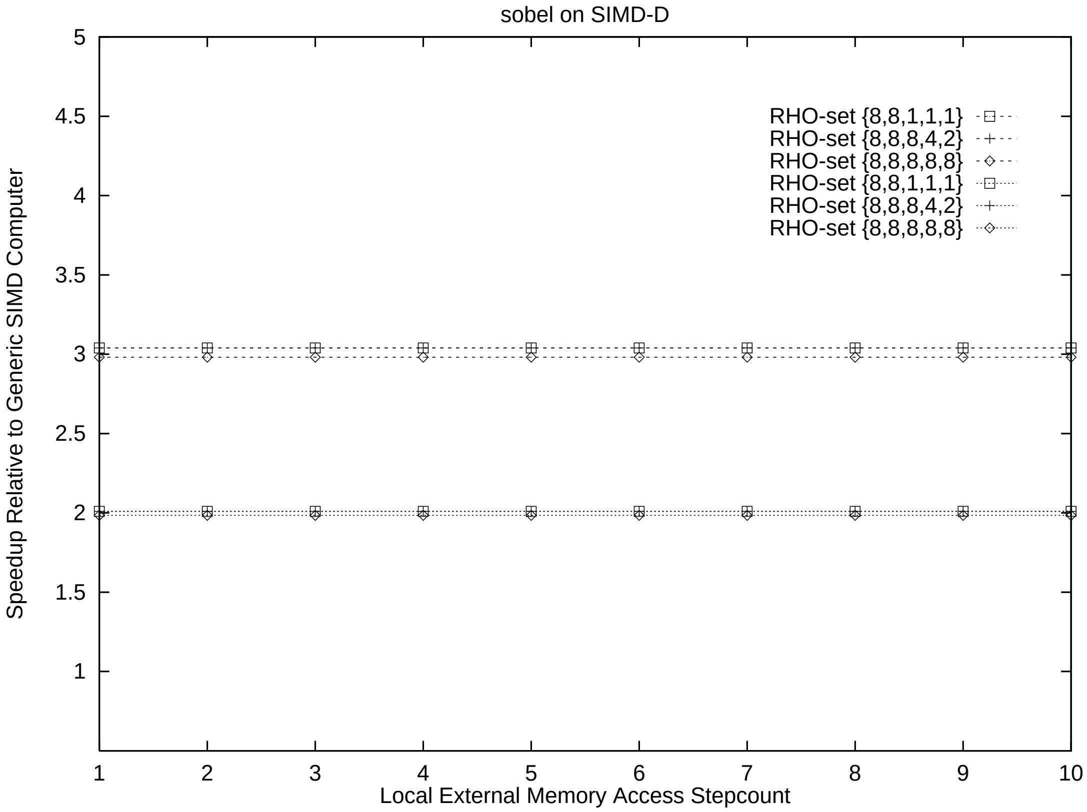


Figure 5.39: I-Cache Speedups v. Local External Memory Access Stepcounts for **`sobel`**

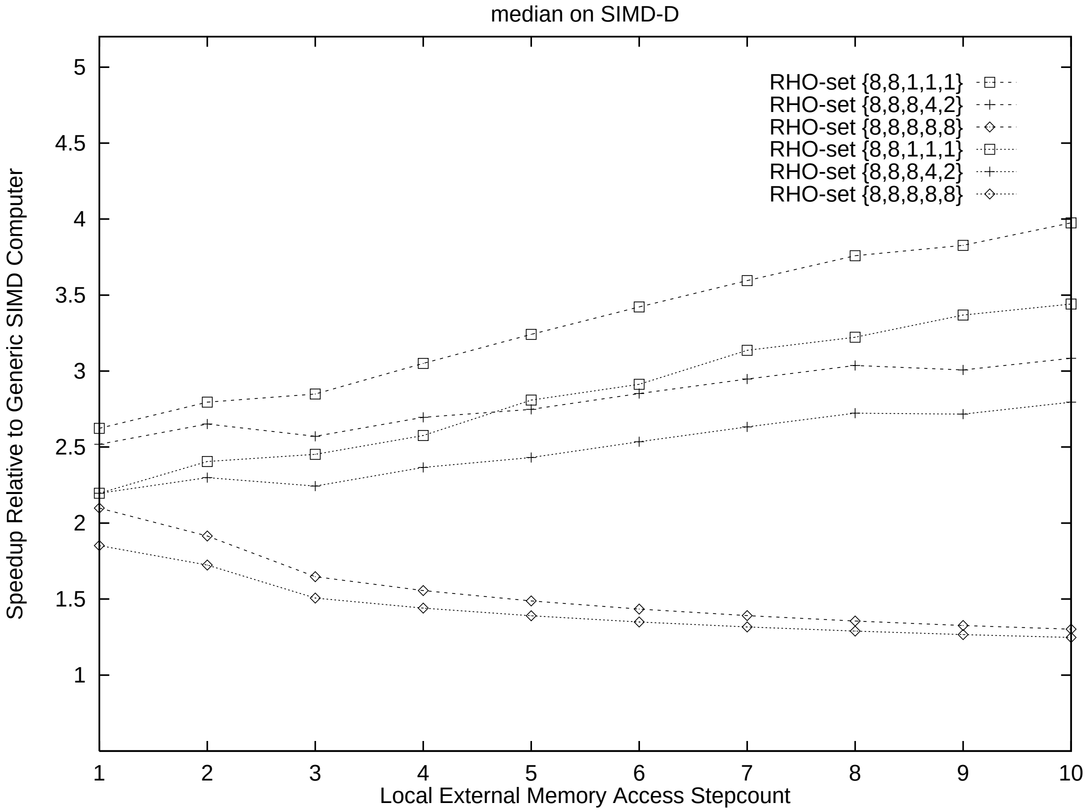


Figure 5.40: I-Cache Speedups v. Local External Memory Access Stepcounts for **`median`**

Chapter 6

Providing Chip Area for I-Cache

I-cache increases the chip area occupied by a PE chip's local controller, due to the addition of a multi-clock generator, cache memory, and a cache controller. The I-cache evaluation discussed in Chapter 5 presupposes that PE chip expands as required to accommodate the local controller with I-cache. Increasing chip area is not always economically feasible, because as the physical dimensions of a chip increase, production yield decreases [80]. Decreased yield means that unit production cost increases, so increasing the size of the PE chip increases its cost. A limited implementation budget therefore imposes a limit on the physical size of the PE chip.

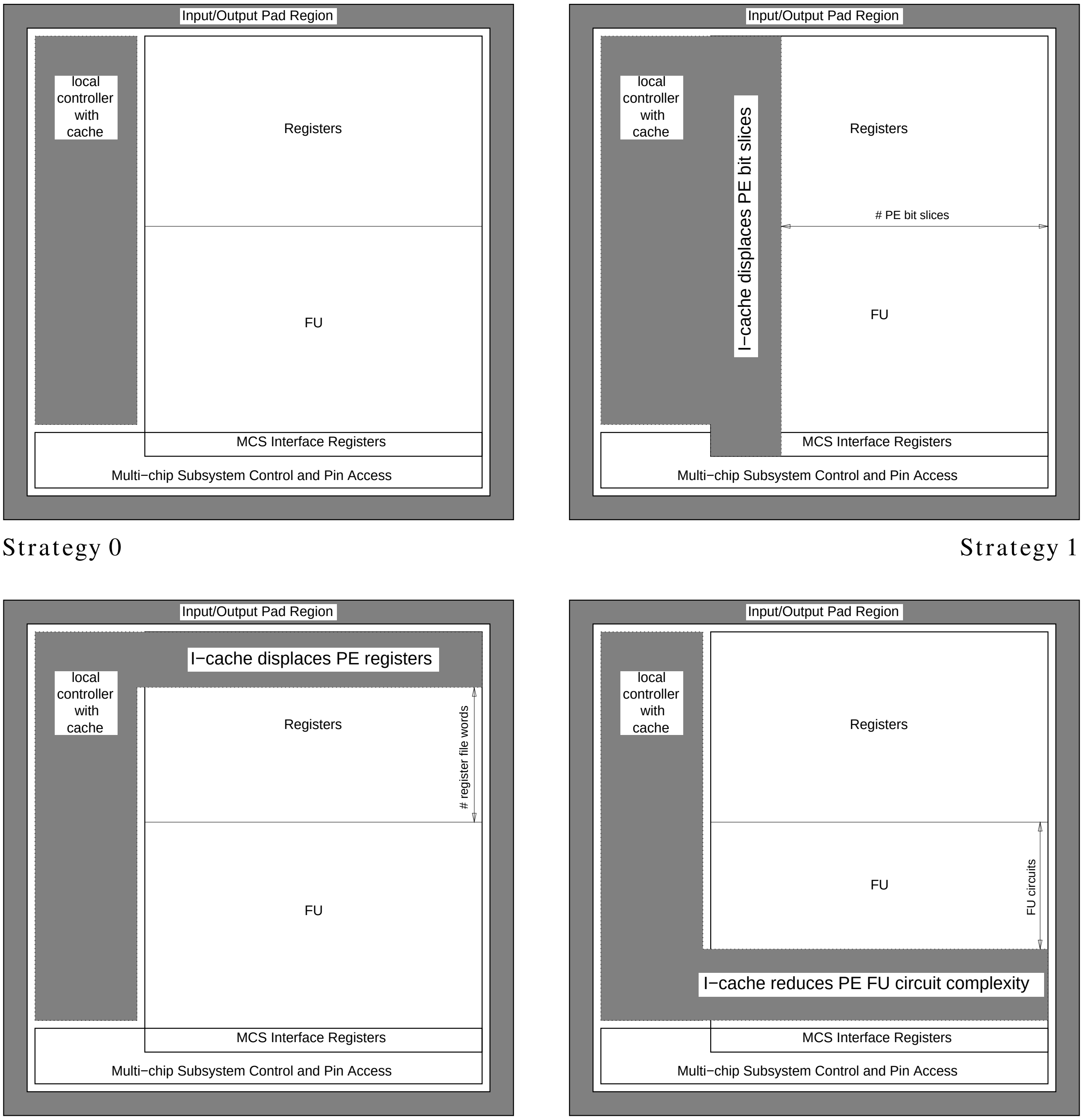
In a PE chip of fixed chip area, cache size is limited. Furthermore, adding I-cache may force the PE chip payload to be reduced, through reducing PE count, PE register count, and/or FU complexity. The payload should be reduced in a way that maximizes the overall speedup, so the best way to provide chip area for I-cache depends on the resource utilization characteristics of a given computation.

This chapter enumerates a variety of ways to accommodate I-cache in a PE chip of fixed physical size and examines the consequences of each. The chip-area use tradeoffs that are apparent in this chapter arise also in the design of PE chips for generic SIMD computers. Here, the question of how best to use the available chip area is re-opened in light of the new requirement to make room for I-cache.

6.1 Strategies for Providing Chip Area for I-Cache

Figure 3.11 shows a floor plan for a PE chip that matches the floor plans of the PE chips in existing SIMD computers. In a PE chip of fixed area and fixed VLSI implementation technique, I-cache is constrained in the chip area it occupies, and adding I-cache may force removing other components from the PE chip. Using the floor plan in Figure 3.11 as a starting point, Figure 6.1 illustrates a collection of *Strategies* for providing chip area for I-cache. The various Strategies, shown in Figure 6.1 as modifications to the floor plan of Figure 3.11, are as follows:

- Strategy 0 is to provide chip area for I-cache by using available interior area that has not already been used for PEs. Strategy 0 presupposes that in the generic SIMD computer's PE chip, substantial areas of the chip interior are unused. Under Strategy 0, that otherwise unused chip area is occupied by I-cache. As do all of the other strategies, Strategy 0 introduces a restriction on the number of instructions that can fit in cache at once. Strategy 0 is obviously the best way to provide chip area for I-cache, because it does not reduce the PE chip payload. Unfortunately, if there is not much "spare" interior area in the PE chip to start with, then Strategy 0 may not allow enough chip area for the resulting cache to be as large as required to realize the maximum speedup for a given computation.



Strategy 0

Strategy 1

Strategy 2

Strategy 3

Figure 6.1: Cache Area-Provision Strategies

- Strategy 1 is to reduce the number of PEs in the PE chip. For a fixed PE chip count, Strategy 1 reduces the number of PEs in the computer. For a fixed problem size, applying Strategy 1 requires re-structuring the computation to reflect reduced PE count: Remaining PEs are assigned sub-problems that would have been assigned to the displaced PEs.
- Strategy 2 is to reduce PE register count. For some computations, the operational structure changes to reflect a reduced number of registers per PE. Some sub-problem data allocated to PE registers in the generic SIMD computation are re-assigned to locations in local external memory under Strategy 2.
- Strategy 3 is to reduce PE FU complexity. As a result of decreased FU complexity, some FU operations take an increased number of clock cycles to complete.

6.2 Method for Measuring Speedups at Fixed Chip Area

The I-cache evaluation method illustrated in Figure 5.1 applies to computations wherein the chip area of the PE chip is permitted to increase to accommodate local controller expansion with I-cache. Figure 6.2 illustrates the method adapted in the following ways to compensate for the local controller's expansion within a PE chip of fixed chip area:

- Before being transformed into a physically structured multi-clock SIMD computation, the assembly language program describing the subject generic SIMD computation is first transformed to reflect reduced PE count or reduced PE register count.
- In addition to the re-organization and addition of cache-control instructions needed to use I-cache, the subject computation's assembly language description is transformed also to reflect reduced PE count or reduced PE register count.
- In transforming operationally structured computations into their physically structured variants with I-cache, operation stepcount parameters may increase to reflect reduced FU complexity, or they may decrease to reflect reduced PE chip pin time-sharing.

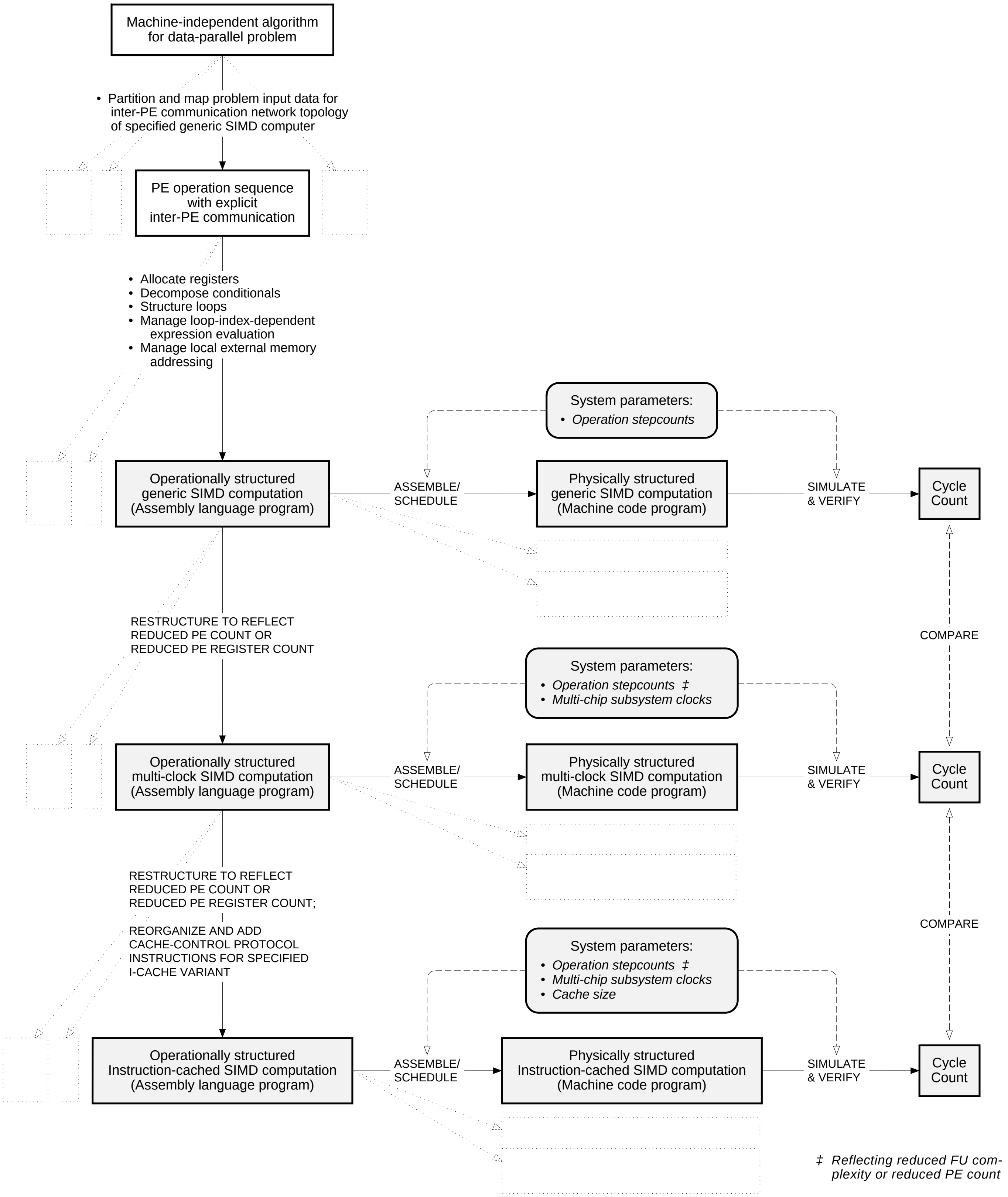


Figure 6.2: Method Adapted for Measuring Speedup at Fixed Chip Area

6.3 Speedups Using Strategy 0

The Strategy 0 case studies show the effects of limited cache size on F_0 speedup. The following 8 graphs (Figures 6.3 through 6.10) show how F_0 speedup of the 8 sample programs varies with cache size. Curves are plotted for each of the ρ-sets $\{8,8,1,1,1\}$, $\{8,8,8,4,2\}$, and $\{8,8,8,8,8\}$. The speedups decrease to the right, as the chip area available for I-cache decreases, thus limiting cache size increasingly severely.

The graphs illustrate the small cache sizes of instructions needed to obtain substantial F_0 speedups for the sample programs on SIMD computer variant SIMD-D. The graphs show that allowing half the maximum cache size required for a given program yields better than half the best speedup for that program at any ρ-set. The graphs also show that the rate at which speedup drops off as cache size is reduced is greatest for small cache memories.

The non-monotonicities of the speedups plotted in Figures 6.3 through 6.10 occur due to interactions of limited cache size with the optimizations performed in the assembler/scheduler. The assembler/scheduler breaks each cache block into two pieces, one sequence of instructions that fits into cache and a remaining sequence that doesn't. Splitting cache blocks in this way requires un-doing some of the re-ordering performed in earlier passes for the purposes of overlapping long-duration MCS operations. A better compiler would certainly handle limited cache size more gracefully, as there is no good reason that having more instructions in cache should slow down a SIMD computation.

The graphs show that for a given cache size, speedups are lower at ρ-sets $\{8,8,8,4,2\}$ and $\{8,8,8,8,8\}$ than at ρ-set $\{8,8,1,1,1\}$. This phenomenon is due to the conservative cachability test applied in the assembler/scheduler. Under that cachability test, an instruction is deemed not to fit in the cache if its latency (in PE clock cycles) exceeds the number of available locations in the cache. Also, once an instruction is excluded, all subsequent instructions in the cache block are also excluded. This cachability test for partial cache blocks is too conservative, as the I-cache variants evaluated here allow timing delays to be represented compactly in cache memory.

The F_0 speedup for Sobel filter, plotted in Figure 6.9, appears to be independent of the ρ-set used. This similarity in the speedup curves for different ρ-sets occurs because the part of the program for which I-cache is used is calculation-intensive, and thus instruction delivery-bound on the simulated model of SIMD computation.

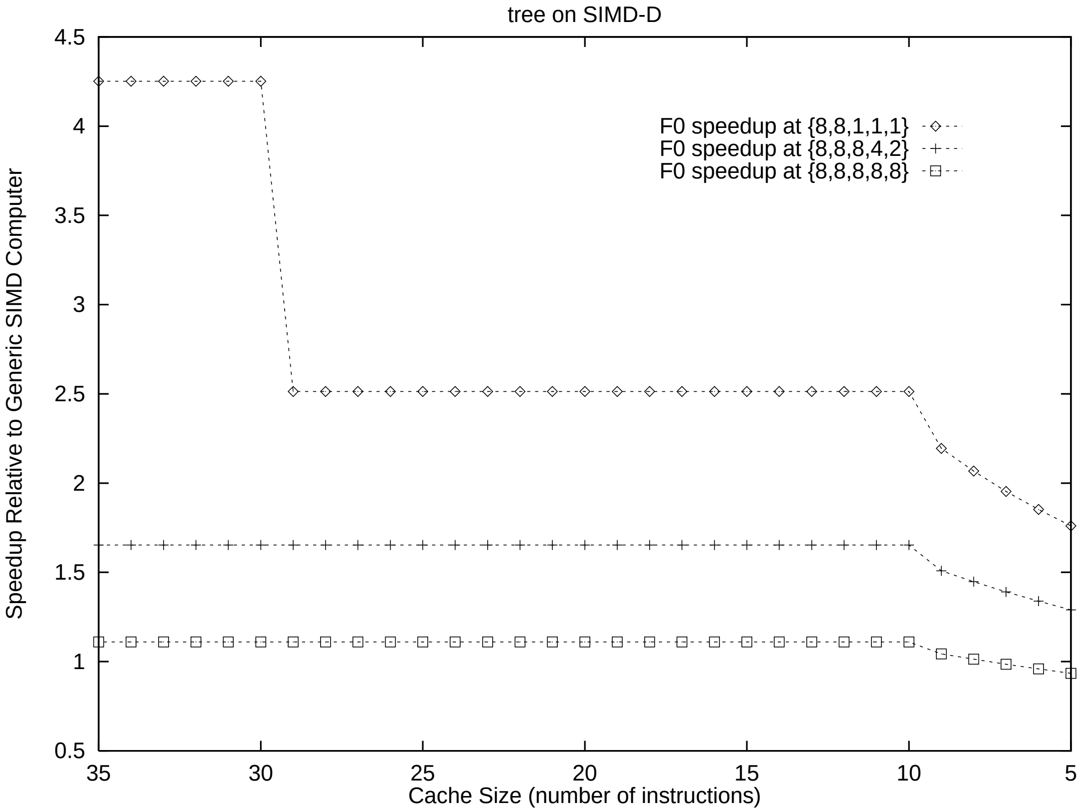


Figure 6.3: F_0 Speedup versus Cache Size for **`tree`**

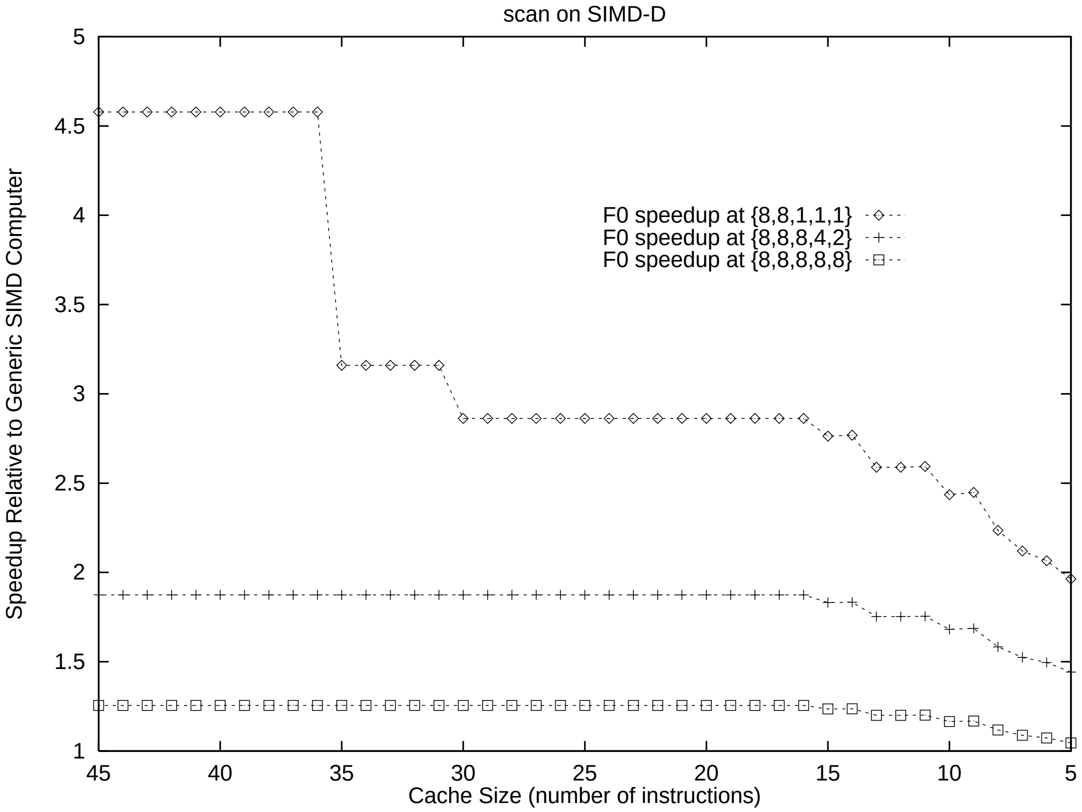


Figure 6.4: F_0 Speedup versus Cache Size for **`scan`**

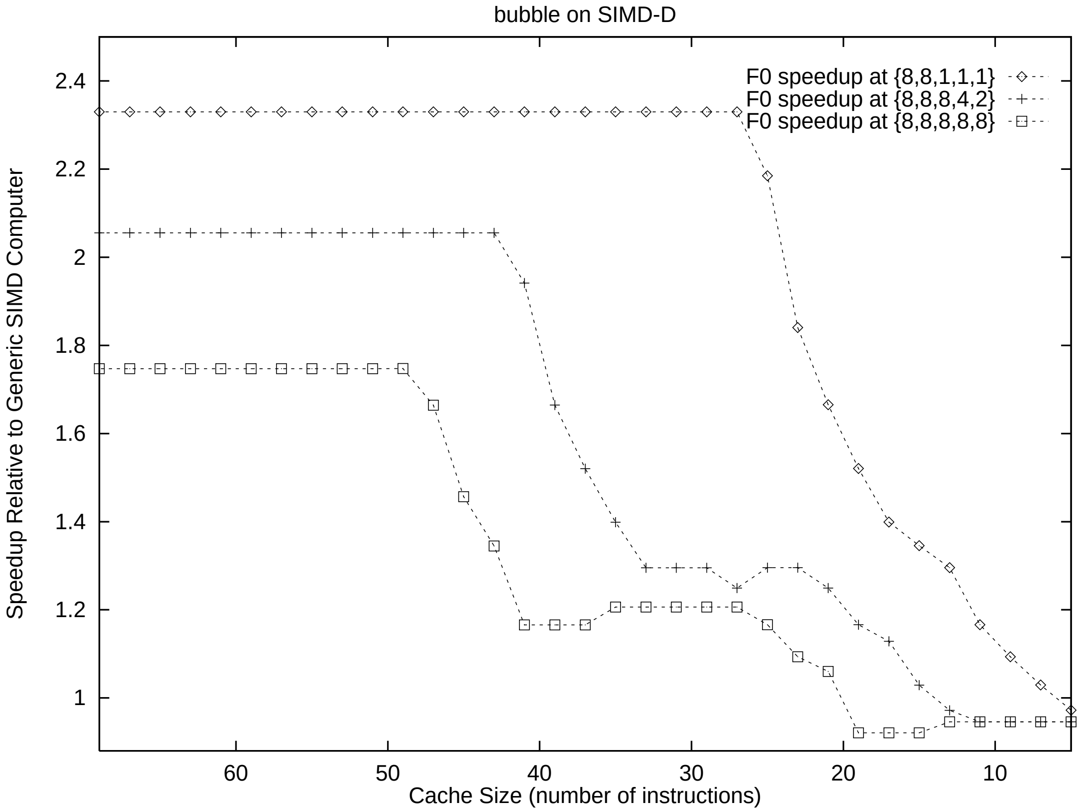


Figure 6.5: F_0 Speedup versus Cache Size for **`bubble`**

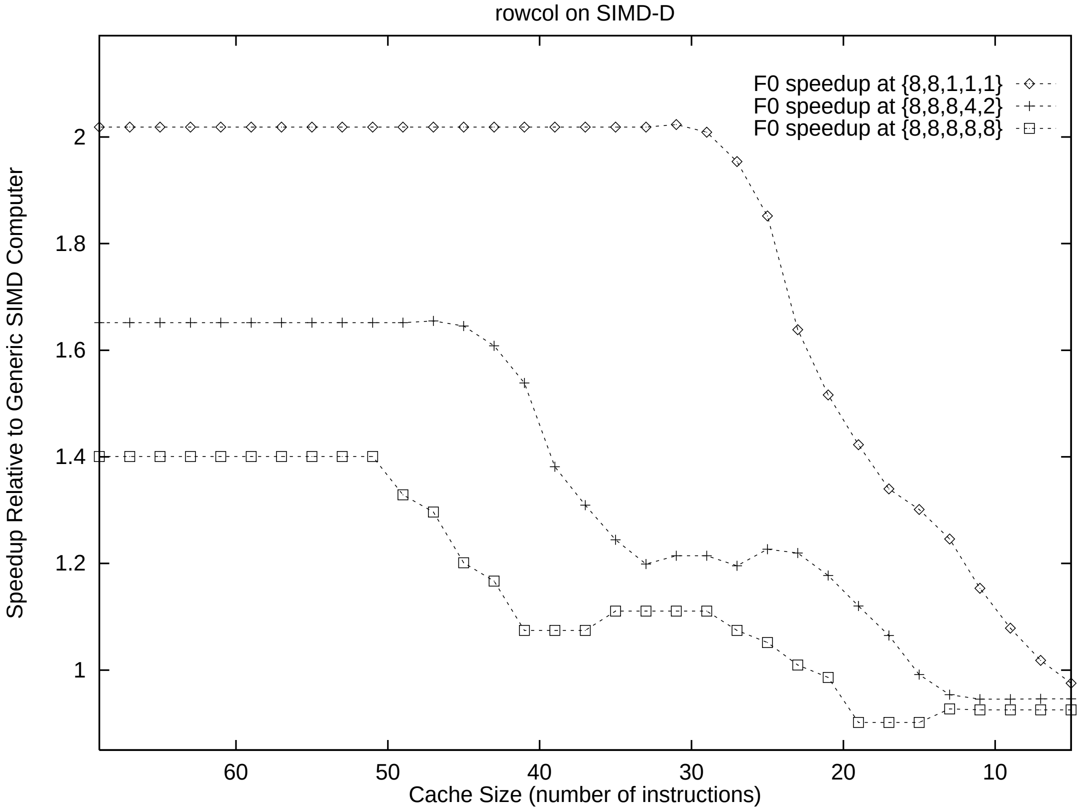


Figure 6.6: F_0 Speedup versus Cache Size for **`rowcol`**

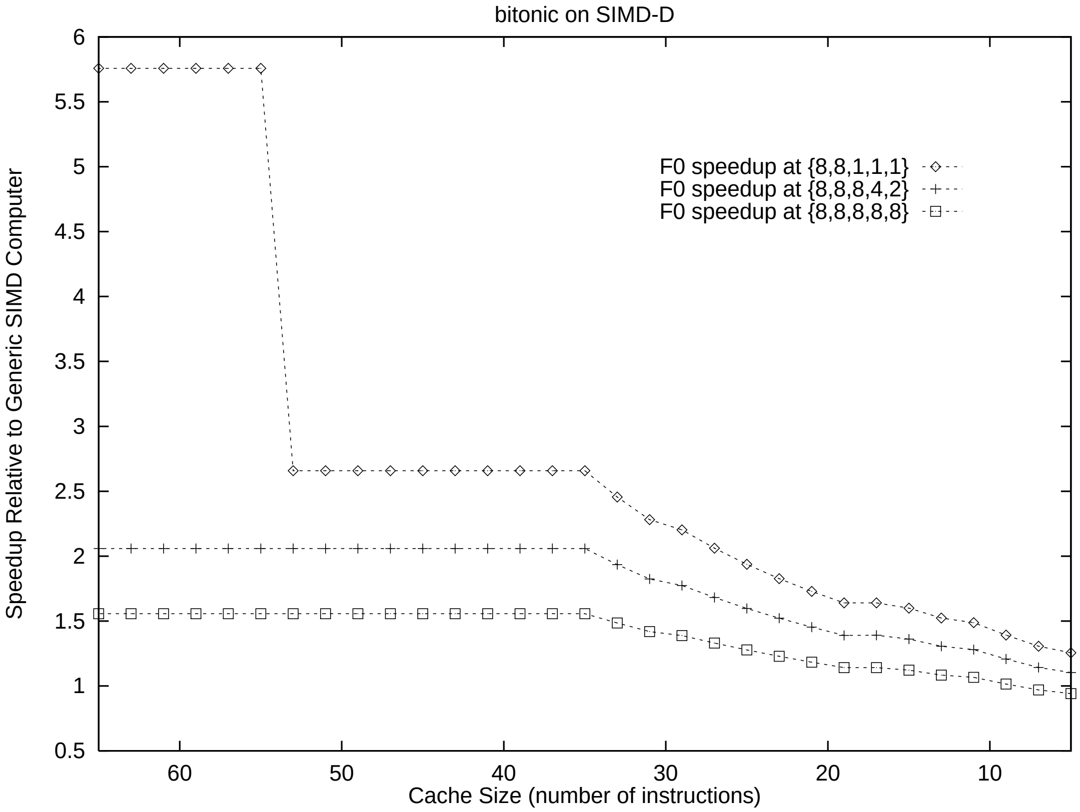


Figure 6.7: F_0 Speedup versus Cache Size for **`bitonic`**

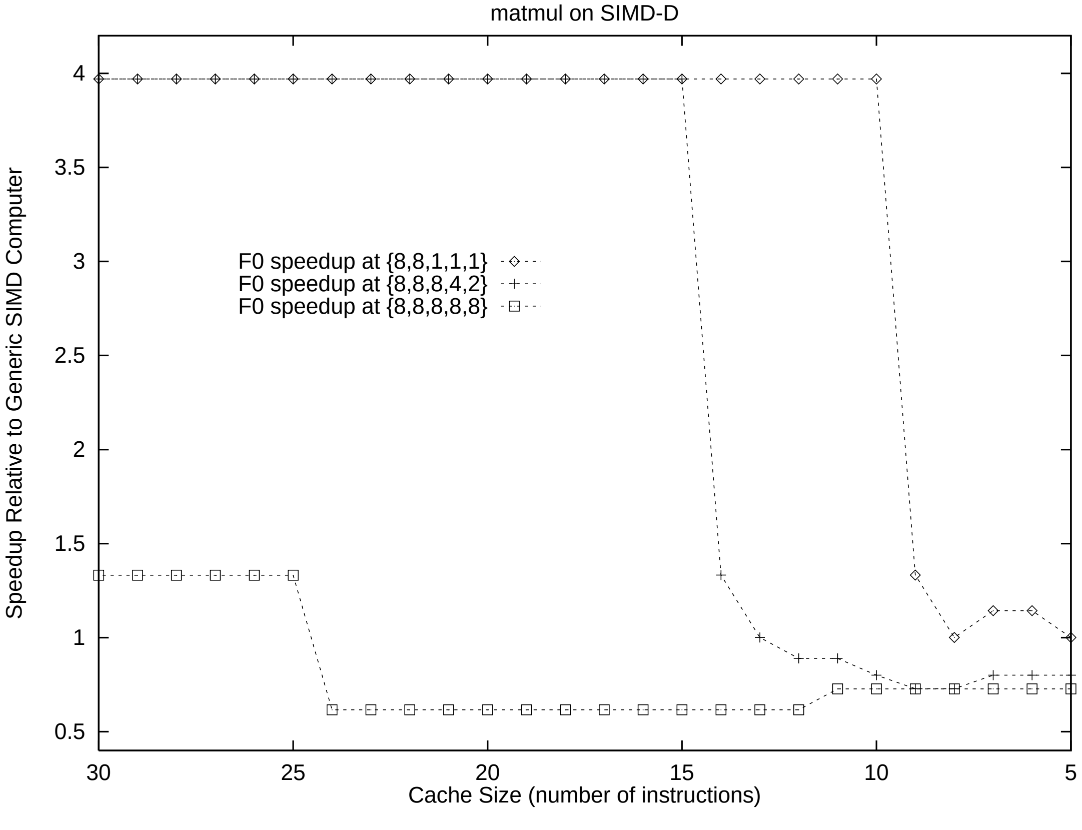


Figure 6.8: F_0 Speedup versus Cache Size for **`matmul`**

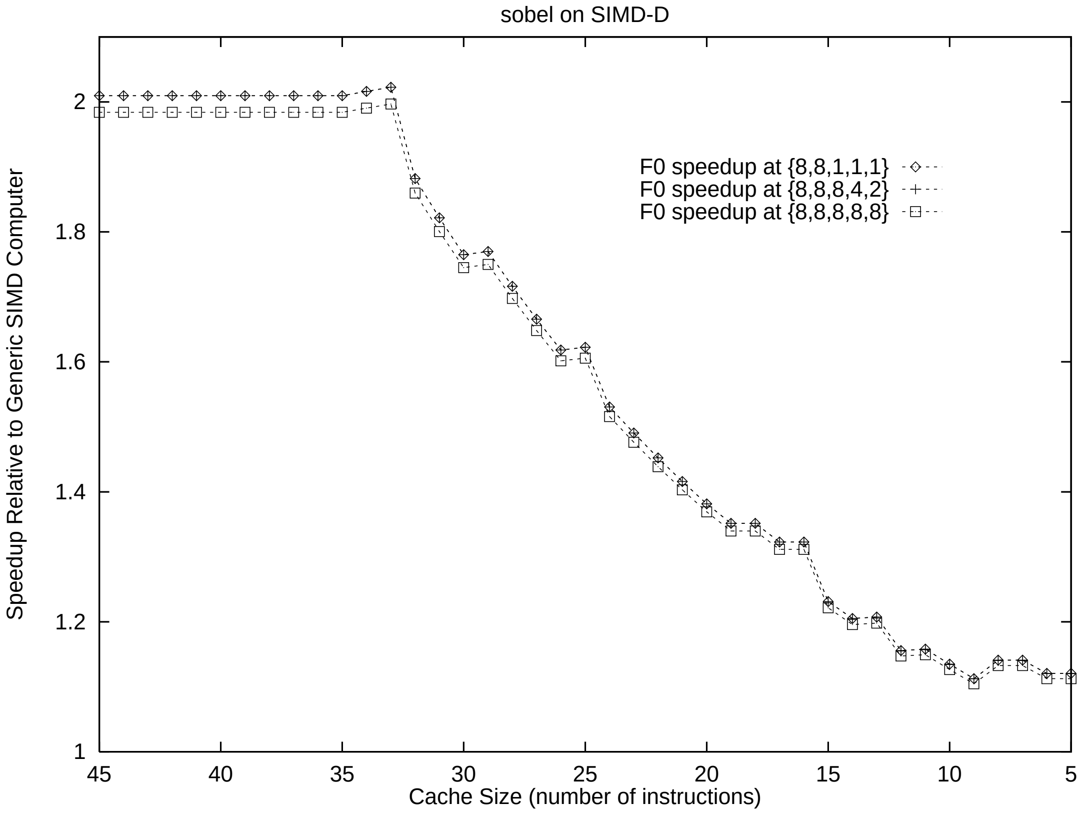


Figure 6.9: F_0 Speedup versus Cache Size for **`sobel`**

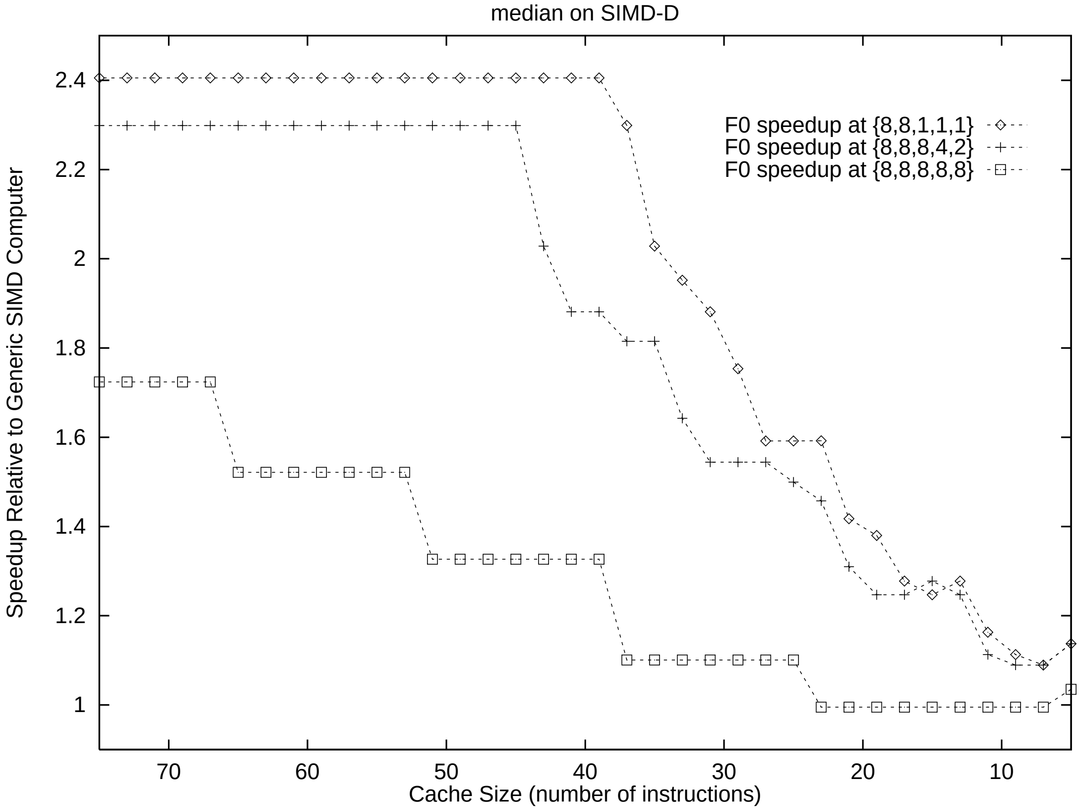


Figure 6.10: F_0 Speedup versus Cache Size for **`median`**

6.4 Speedups Using Strategy 1

Strategy 1 is to displace PEs from the PE chip. Strategy 1 has a variety of effects on both the operational structure and the physical structure of a computation, depending upon the context in which it is applied.

One surprising effect of Strategy 1 is that by reducing the number of PEs time-sharing MCS pins in the PE chip, Strategy 1 tends to reduce stepcounts for MCS operations, including local external memory access and inter-PE communication. All else being equal, this effect of Strategy 1 tends increases throughput. Unfortunately, not all else is equal, because the computation must be restructured to compensate for the PEs displaced under Strategy 1.

The number of PEs P in a generic SIMD computer is given as

$$P = KM$$

where K is the number of PEs per PE chip and M is the number of PE chips in the computer.

When I-cache is added to a SIMD computer using Strategy 1, the number of PEs per PE chip is reduced to a new number K', such that $K' < K$. The number of PEs P' in the I-cached SIMD computer is either less than, equal to, or greater than the original number of PEs P. The consequences of these alternative scenarios are enumerated below:

1. $P' < P$

   This is the case where the number of PE chips cannot be increased. The constraint $M' = M$ applies, for example, where the total physical size of the computer is limited. Under this constraint, it becomes necessary to re-program $K'M$ PEs to do the work of the original P PEs. This reprogramming corresponds to changing the operationally structured description of the computation.

   The ratio of the number of PEs in the model on which a program is written to the number of PEs in the computer is commonly referred to as the *VP-ratio* [18](p.9). Under Strategy 1 I-caching with fixed PE chip count,

$$\begin{aligned} \text{VP-ratio} &= \frac{P}{P'} \\ &= \frac{K}{K'} \end{aligned}$$

   The virtual processors programming abstraction [18](Sec.3.2) facilitates writing programs assuming one PE per data element without regard to the (perhaps lower) number of PEs in a computer. A program written using the virtual processing abstraction is translated into an operationally structured description of the computation in which the available PEs share the workload evenly [41](p.1171).

   Strategy 1 increases loop iteration counts by a factor of $\lceil \frac{K}{K'} \rceil$. While increasing computation time, increasing loop iteration counts actually tends to increase I-cache speedup.

   Large VP-ratios place a high demand on PE memory [84](p.106), because the amount of memory required by the PE increases linearly with its workload, which is proportional to the VP-ratio. Another effect of Strategy 1, which occurs when the required size of PE memory increases to a point where it exceeds the size of the PE register file, is to cause some PE data that was located in registers in the original computation to be re-located to local external memory. Data in local external memory typically is accessed at a lower rate than data in registers, so throughput and I-cache speedup decrease under this effect. This effect obtains if each PE in the original generic SIMD computation has available less than $\lceil \frac{K}{K'} \rceil$ times the number of registers it requires.

By reducing the number of PEs, Strategy 1 reduces the amount of inter-PE communication, replacing those communications with references to local memory. The impact of this effect on I-cache speedup depends on the relative operation rates of the local external memory and inter-PE communication subsystems.

2. $P'=P$

   Some computational problems require a specific number of physical PEs. If K' divides P, then it is possible to increase the number of PE chips so that

   $$\begin{aligned} \frac{M'}{M} &= \frac{K}{K'} \text{ such that} \\ K'M' &= KM \end{aligned}$$

   In this case, adding I-cache does not change the operational structure of the computation. The same number of PEs perform the same amount of work as in the computation without I-cache.

   The physical structure of the computation does change in this case, to reflect decreased time-sharing of PE chip pins. Also, ρ-set values may increase as chips are added, to reflect increases in MCS' inter-chip wire lengths and electrical loads.

3. $P' > P$

   If the computational problem requires at least a certain number of physical PEs and K' does not divide P, then the I-cached SIMD computer is forced to contain $K'M' > P$ PEs. The extra PEs are redundant. Consider shifting data around a linear ring of PEs; additional shift steps are required to move data past redundant PEs in the ring. A further subtle effect in this case is the operational structure of the computation must change to prevent redundant PEs from altering data used by the needed PEs. The redundant PEs are inconvenient. Strategy 1 I-caching in this case introduces unnecessary inter-PE communication operations, which tend to limit I-cache speedup.

There are a large number of ways to apply Strategy 1, and there are many interacting effects of reducing the number of PEs. Considering the Strategy 1 effects raises the question, what is the ideal number of PEs to have in a PE chip? There are many plausible answers to such a question, and no absolute answer. If the number of PE chips may increase, then Strategy 1 may have little impact on the structure of the computation. On the other hand, if there are a limited number of PE chips, then Strategy 1 leads to decreased throughput.

6.5 Speedups Using Strategy 2

Strategy 2 is to reduce the number of PE registers to make room for I-cache. If the removed registers are unused in a given computation, then Strategy 2 makes room for I-cache at no cost. On the other hand, throughput suffers for computations that do use the available registers.

Figures 6.11, 6.12, 6.13, and 6.14 show the typical effect on I-cache speedup of varying the number of PE registers for a program that uses all of the registers that are available. The graphs show the progressively worsening I-cache speedup reduction as increasing numbers of registers are displaced from the PE chip.

The subject computation for which Strategy 2 I-caching results are shown is a variant of **`matmul`**. The subject program uses the number of available PE registers as a parameter; extra registers are used as a register buffer for matrix-column data otherwise stored in local external memory.

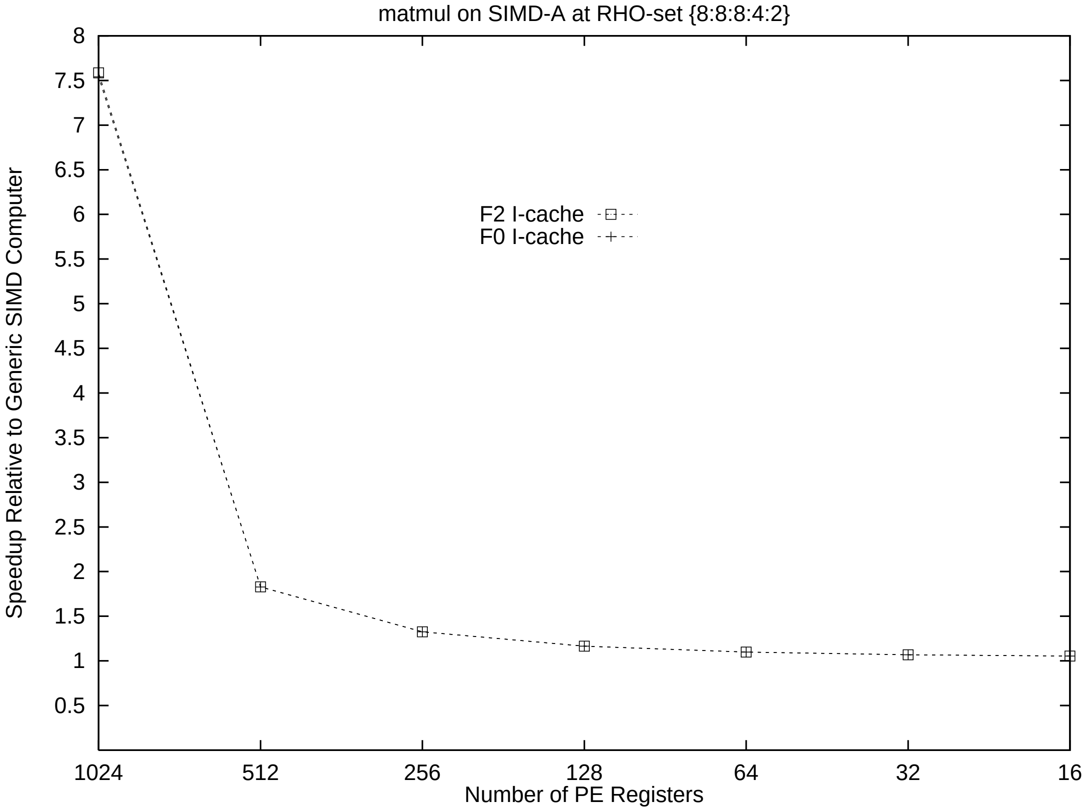


Figure 6.11: Effect of Strategy 2 for **`matmul`** on SIMD-A measured at ρ-set $\{8,8,8,4,2\}$.

The graphs show that the register count reduction bites quickly. Reducing the number of registers by a factor of 2, from 1024 to 512, decreases speedup by more than a factor of 2. By contrast, reducing the number of registers by a factor of 8, from 128 to 16, reduces speedup by a factor of less than 10%.

These results suggest that Strategy 2 has a drastic negative effect on I-cache speedup. For the subject computation, displacing more than half of the PE registers for I-cache defeats most of the I-cache speedup. Given the heavy use made of registers by calculation-intensive programs, Strategy 2 is not a good way of providing chip area for I-cache.

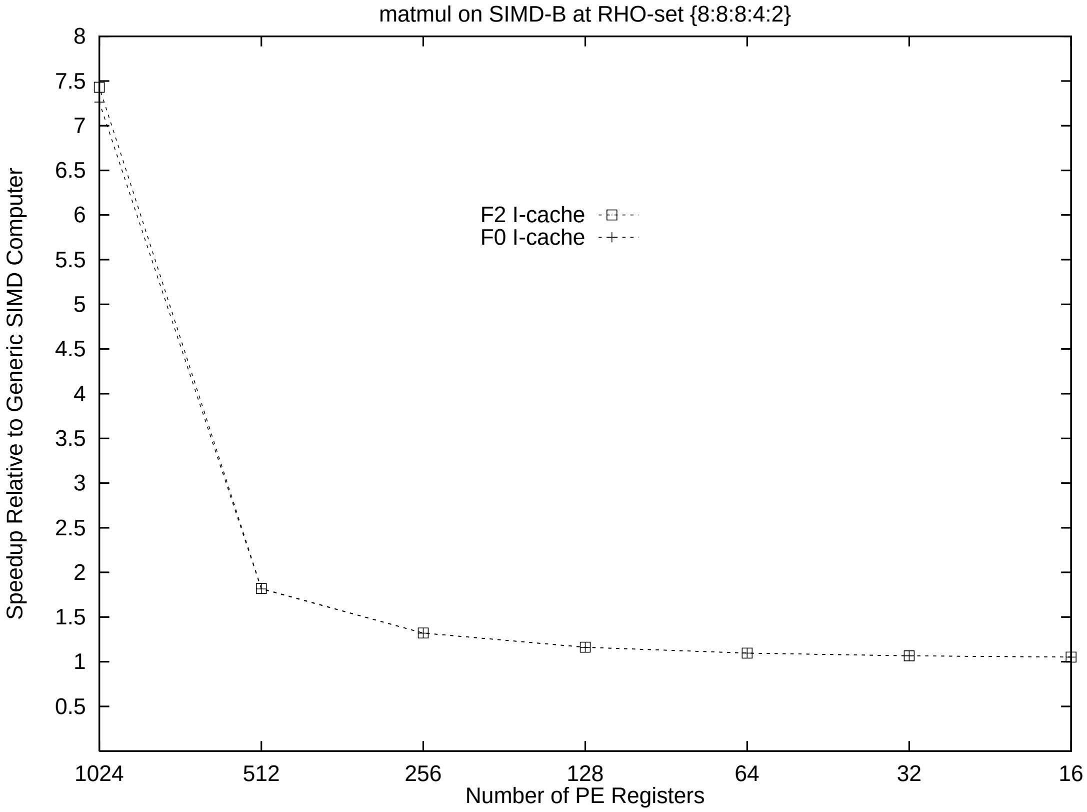


Figure 6.12: Effect of Strategy 2 for **`matmul`** on SIMD-B measured at ρ-set {8,8,8,4,2}.

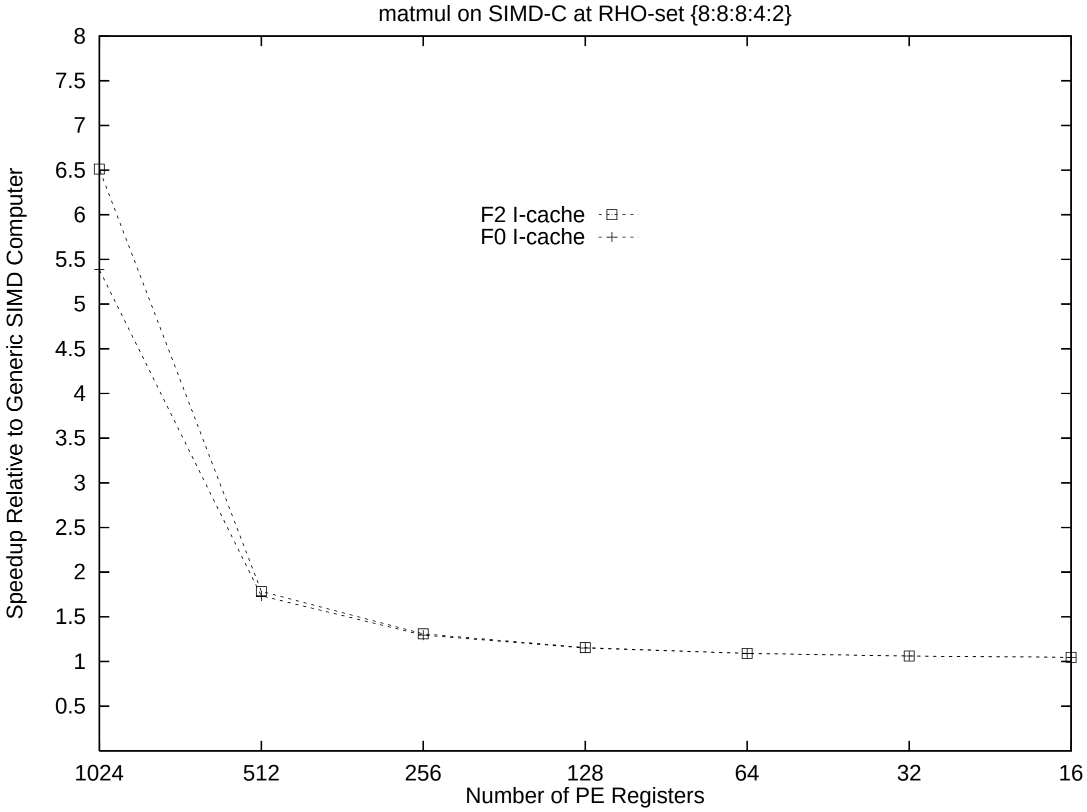


Figure 6.13: Effect of Strategy 2 for **`matmul`** on SIMD-C measured at ρ-set {8,8,8,4,2}.

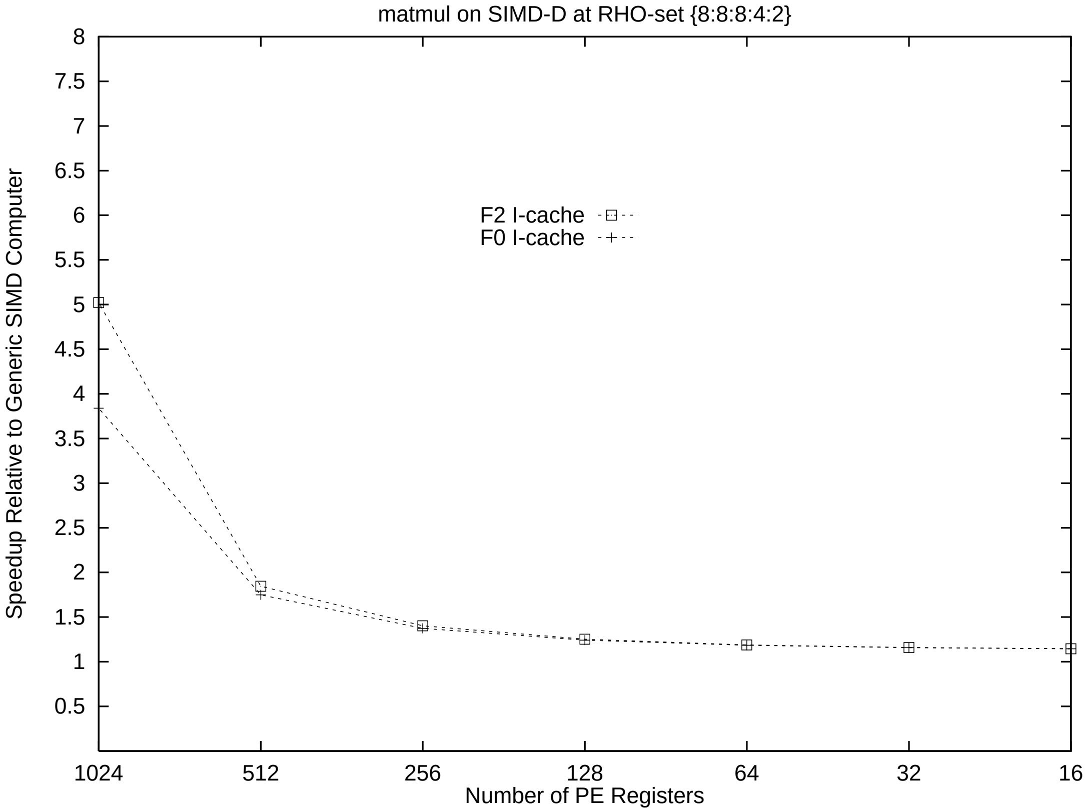


Figure 6.14: Effect of Strategy 2 for **`matmul`** on SIMD-D measured at ρ-set $\{8,8,8,4,2\}$.

6.6 Speedups Using Strategy 3

Strategy 3 is to remove some of the FU circuits to make room for I-cache. Removing circuits simplifies the FU, so that some arithmetic operations take more clock cycles to perform. This effect is reflected as increased FU operation stepcount in the simulation model. Varying FU complexity does not change operationally structured computation descriptions.

Figures 6.15 through 6.22 show the effects of reduced FU complexity on speedup and on required cache size for each of the 8 sample problems on SIMD computer variant SIMD-D measured at ρ-set $\{8, 8, 8, 4, 2\}$.

The upper graph on each page plots F_0 and F_2 I-cache speedup curves. The lower graph on each page plots the corresponding cache size required for each of the two I-cached SIMD computations.

The graphs show that speedup lessens progressively as the FU becomes progressively simpler. However, the slope of the speedup reduction is less than that apparent for Strategy 2. The reason for this gradual speedup decrease is that reduced FU complexity has the effect of lengthening instruction sequences controlling FU operations, and such sequences are instruction delivery-bound in the simulation model. Because I-cache speeds up repeated instruction delivery-bound instruction sequences, the I-cache lessens the impact of FU simplification. However, the fact that the speedups do decrease as FU stepcounts increase indicates that I-cache does not win back all of the calculation speed that was sacrificed to make room for I-cache using Strategy 3.

Note that the required cache size grows nearly linearly with reduced FU complexity for all 8 problems. The required cache size growth is due to the machine-dependent assumption that a new instruction is required on each clock cycle of a multi-step FU operation. On one hand, if the FU part of the instruction set was similar to the MCS parts of the instruction set, then the cache size would not grow as quickly as shown in the graphs. On the other hand, hardware used inside the PE chip to control the FU is redundant and would occupy chip area that could otherwise be used for the calculation components of PEs. Furthermore, for simple FU operations whose stepcounts are 1, inside-the-chip FU control is not needed.

The rapid growth in cache size is perhaps the greatest drawback of Strategy 3 I-caching. FU simplification to make room for I-cache has the counter-productive characteristic of increasing the number of instructions that must be stored in cache to yield the maximum speedup. Given that chip area is precious within the PE chip, this effect could render Strategy 3 infeasible.

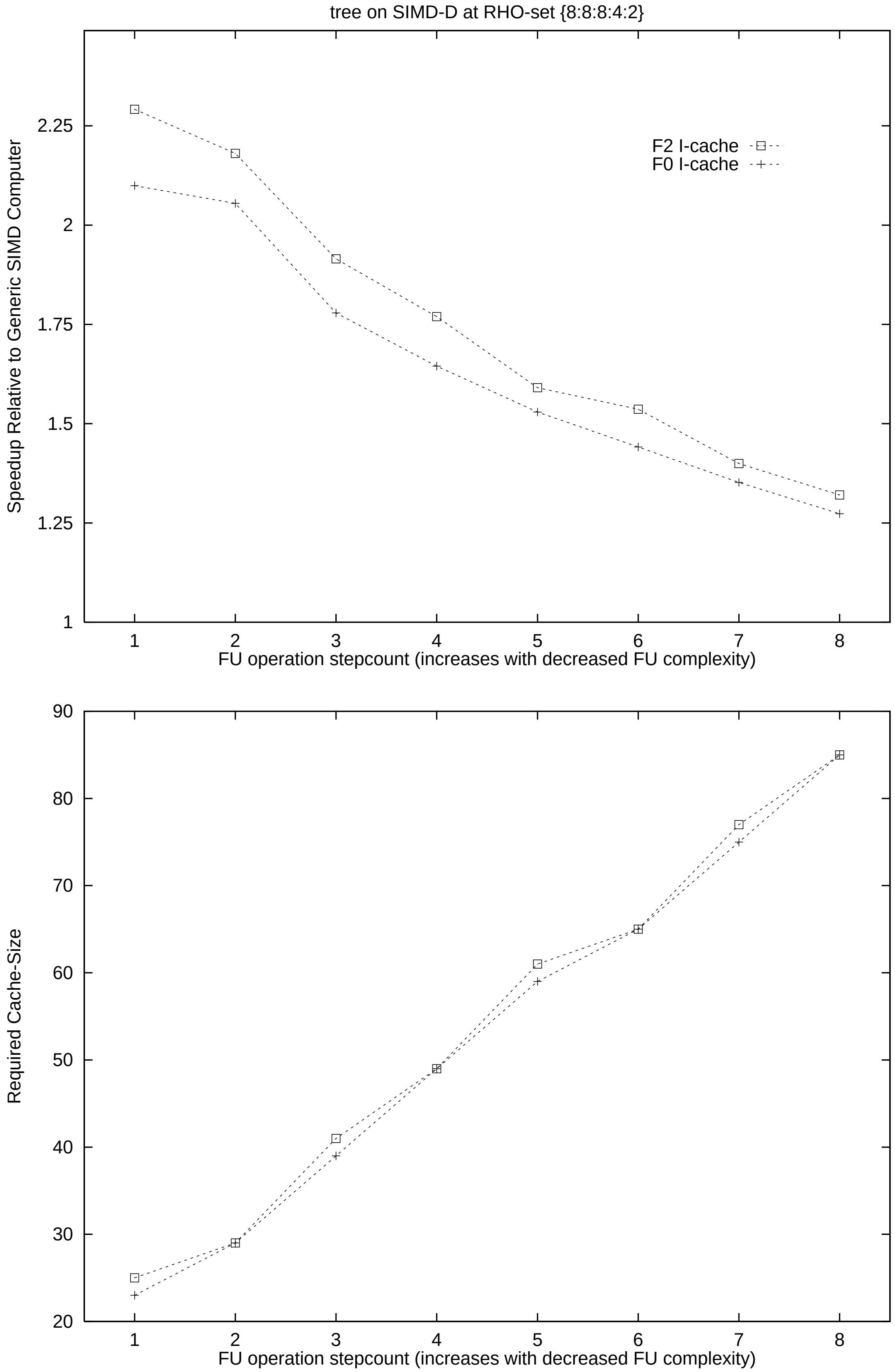


Figure 6.15: Speedup and Cache Size for **`tree`** on SIMD-D v. FU Complexity at ρ-set $\{8,8,8,4,2\}$

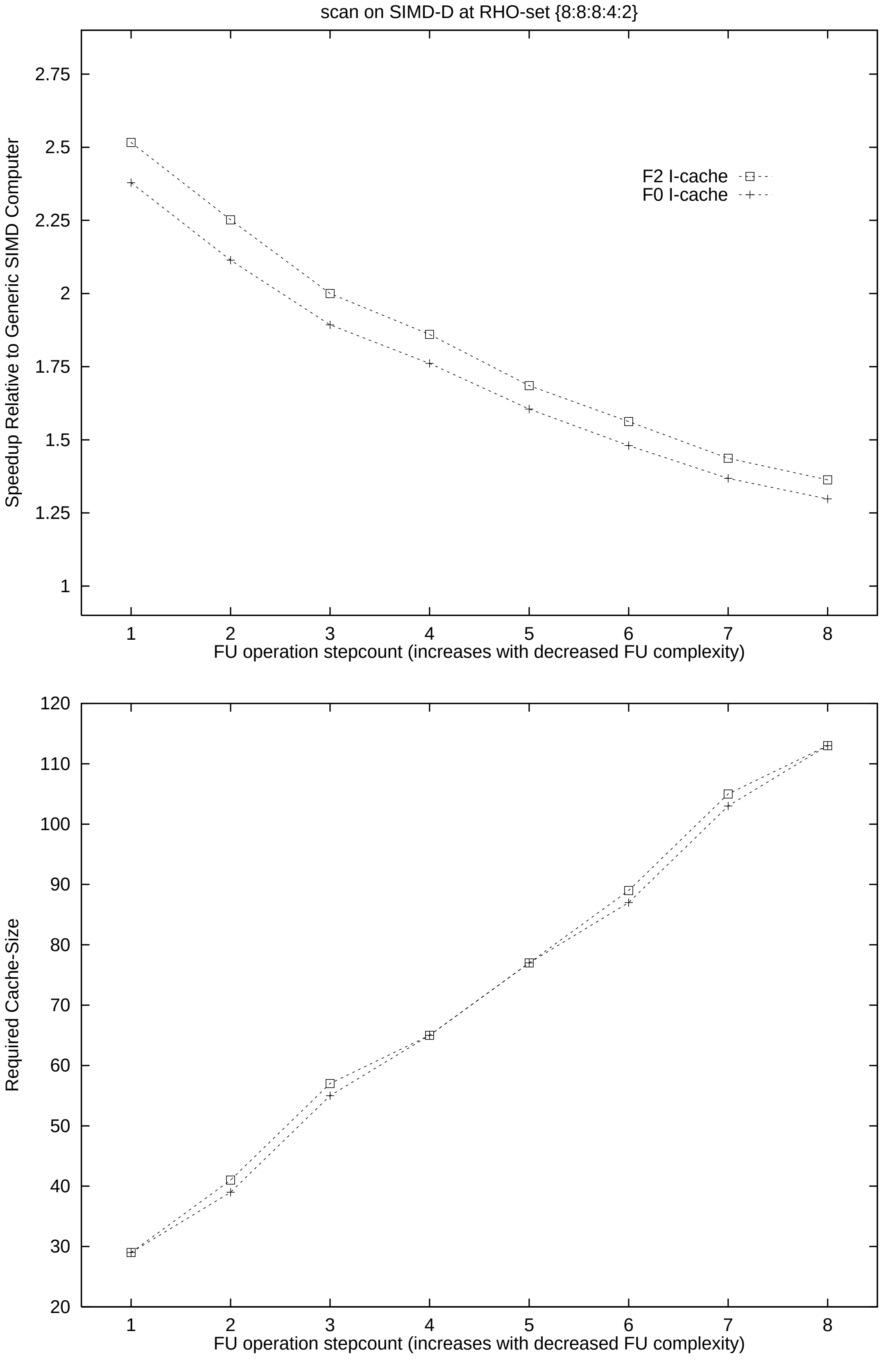


Figure 6.16: Speedup and Cache Size for **`scan`** on SIMD-D v. FU Complexity at ρ-set $\{8,8,8,4,2\}$

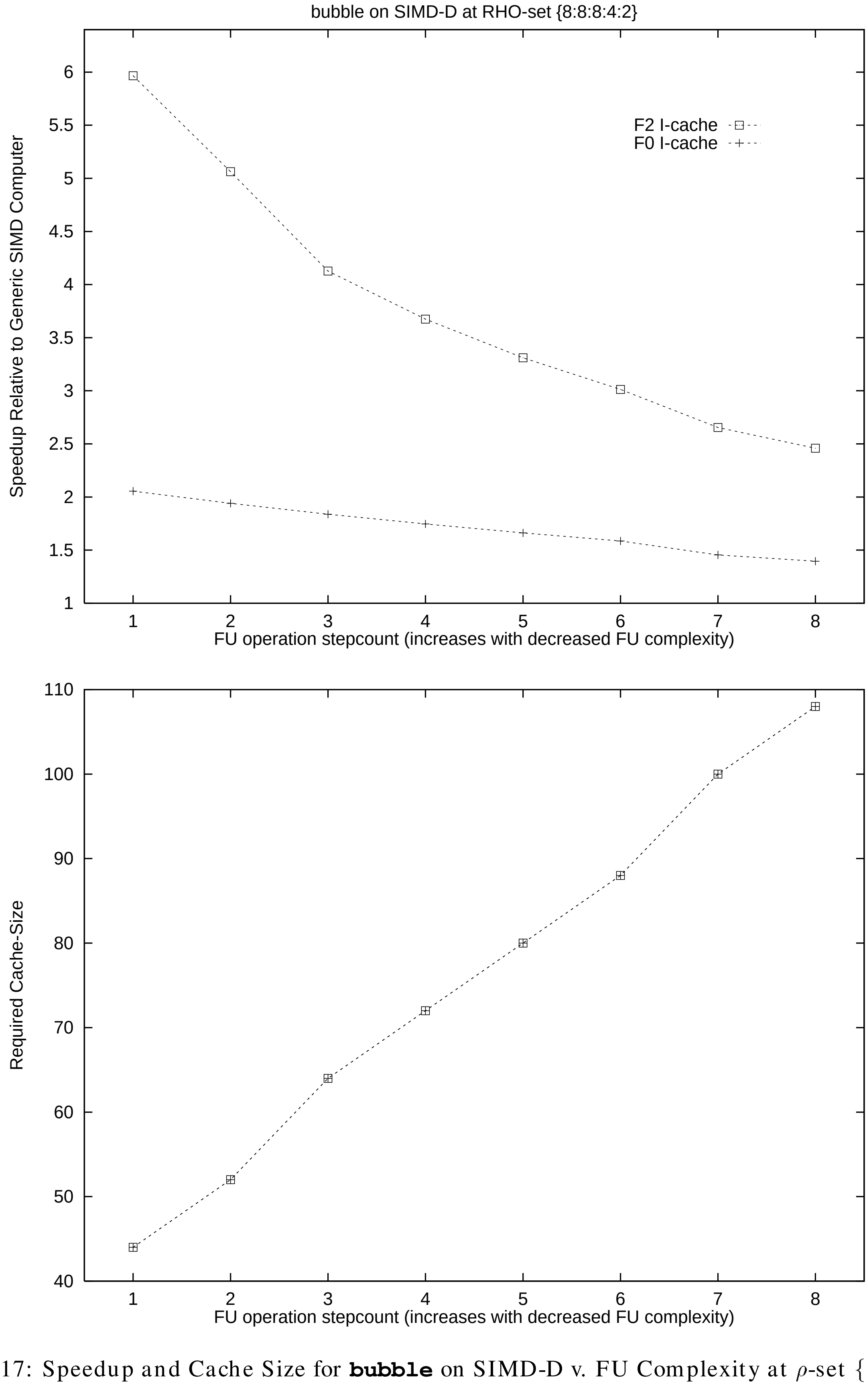


Figure 6.17: Speedup and Cache Size for **`bubble`** on SIMD-D v. FU Complexity at ρ-set $\{8,8,8,4,2\}$

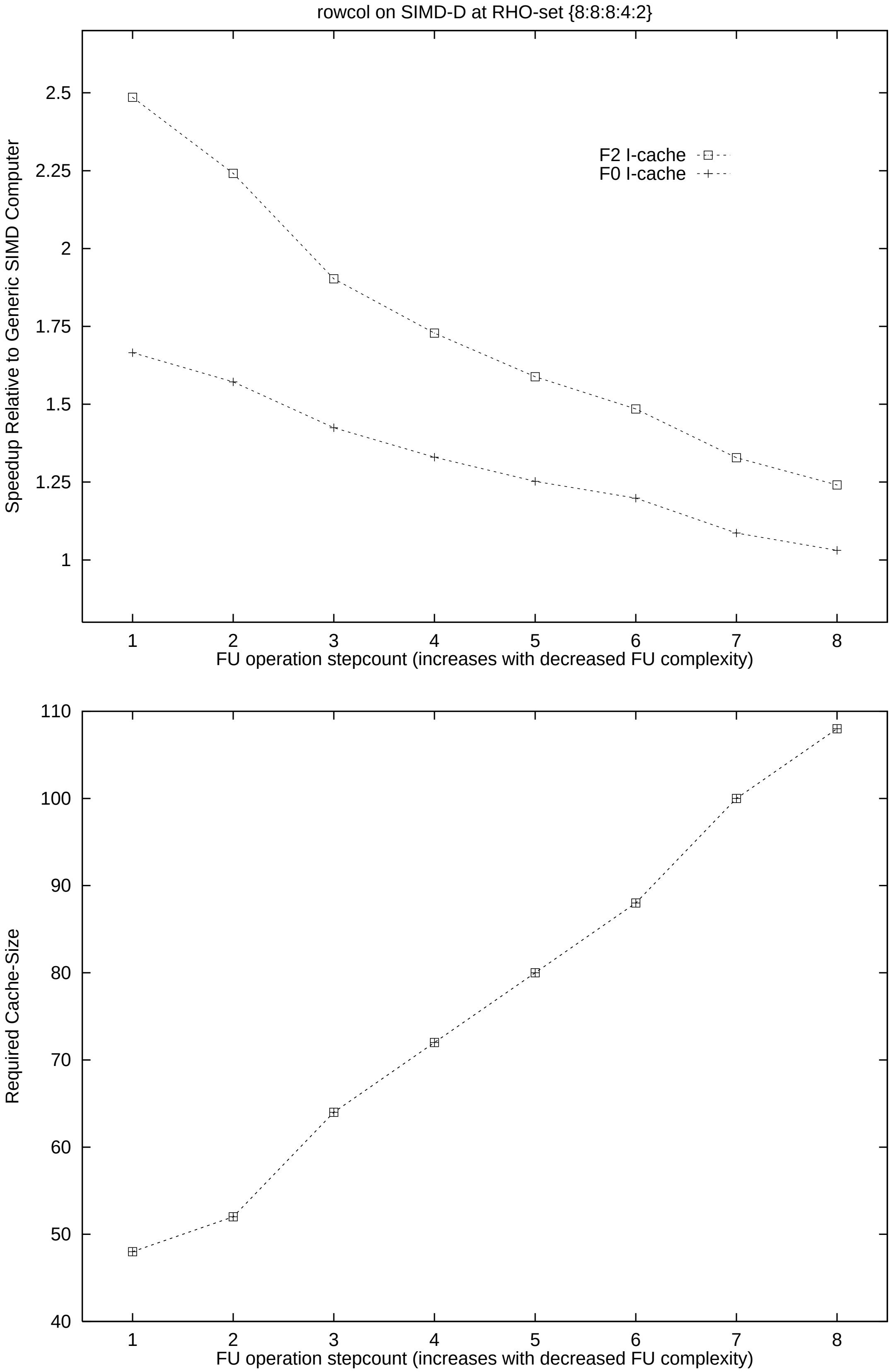


Figure 6.18: Speedup and Cache Size for **`rowcol`** on SIMD-D v. FU Complexity at ρ-set {8,8,8,4,2}

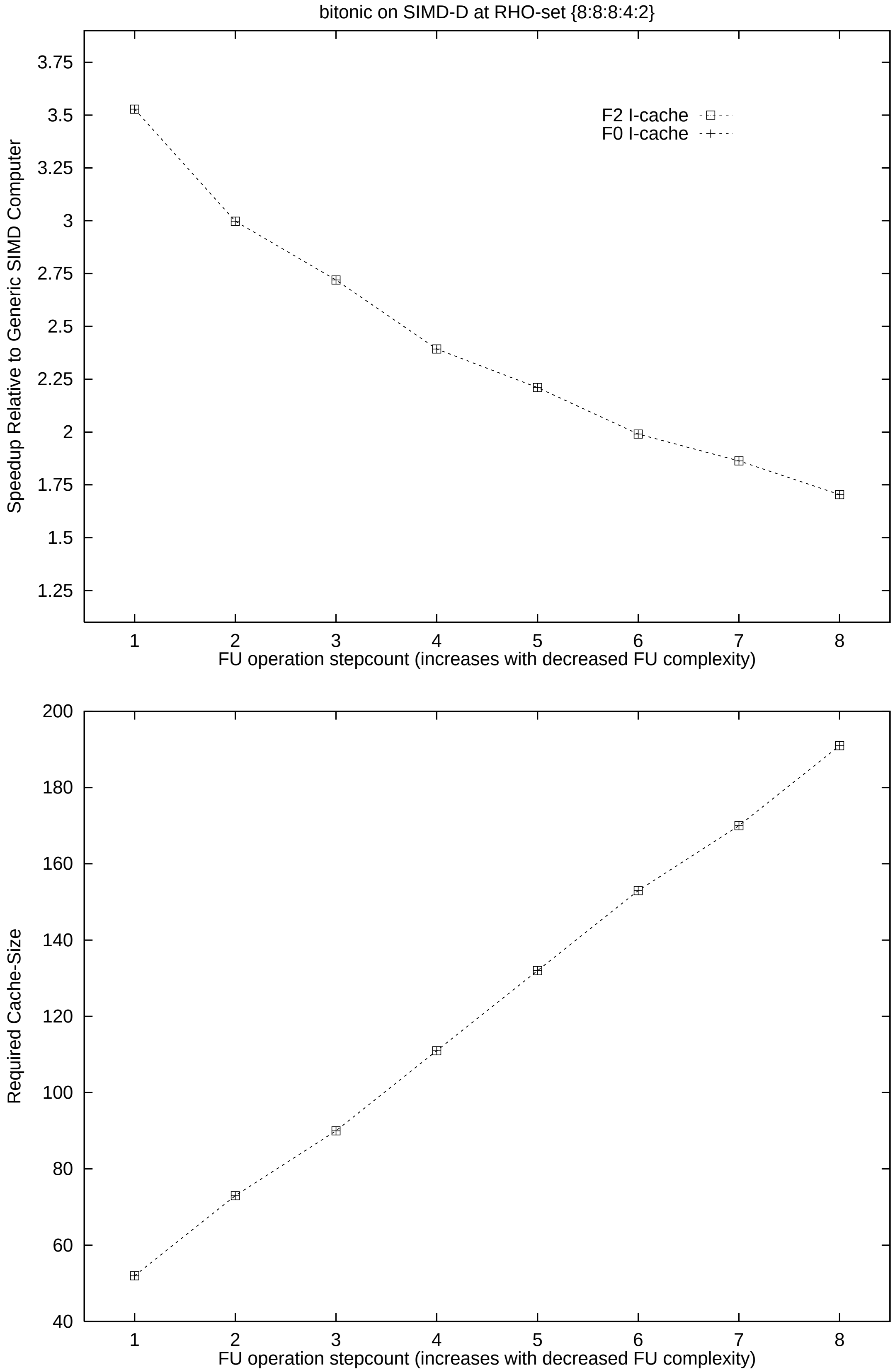


Figure 6.19: Speedup and Cache Size for **`bitonic`** on SIMD-D v. FU Complexity at ρ-set {8,8,8,4,2}

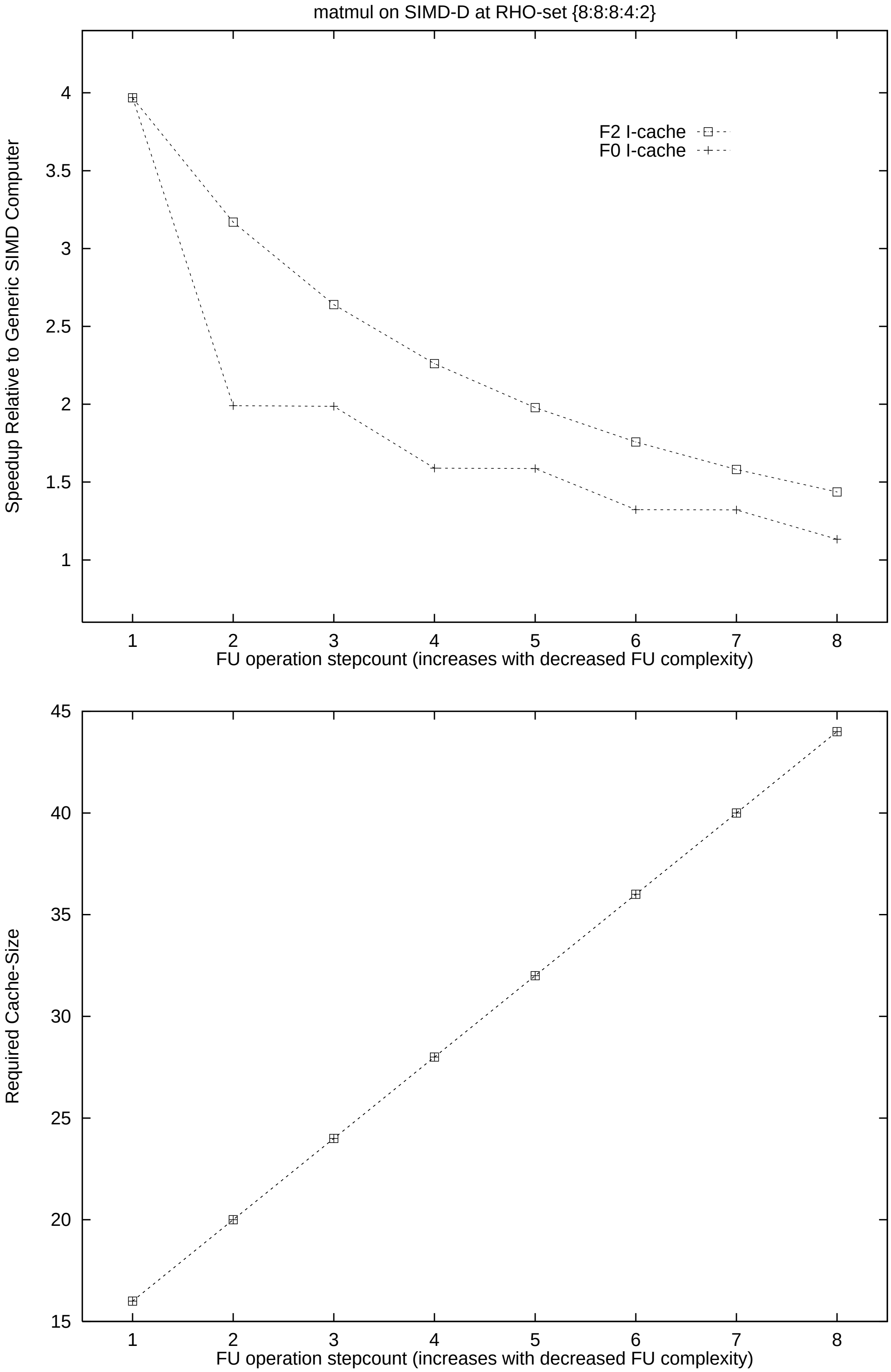


Figure 6.20: Speedup and Cache Size for **`matmul`** on SIMD-D v. FU Complexity at ρ-set {8,8,8,4,2}

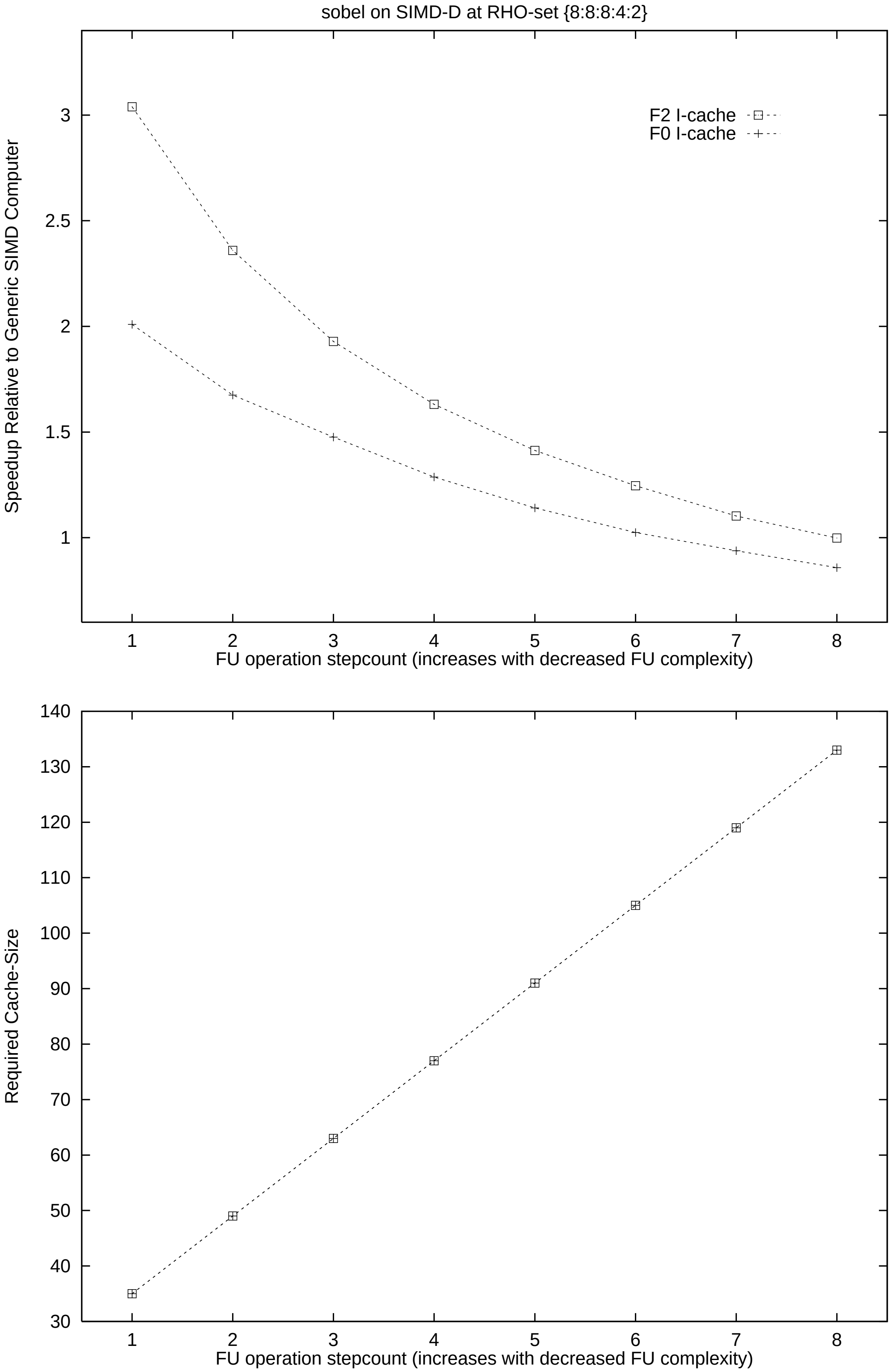


Figure 6.21: Speedup and Cache Size for **`sobel`** on SIMD-D v. FU Complexity at ρ-set {8,8,8,4,2}

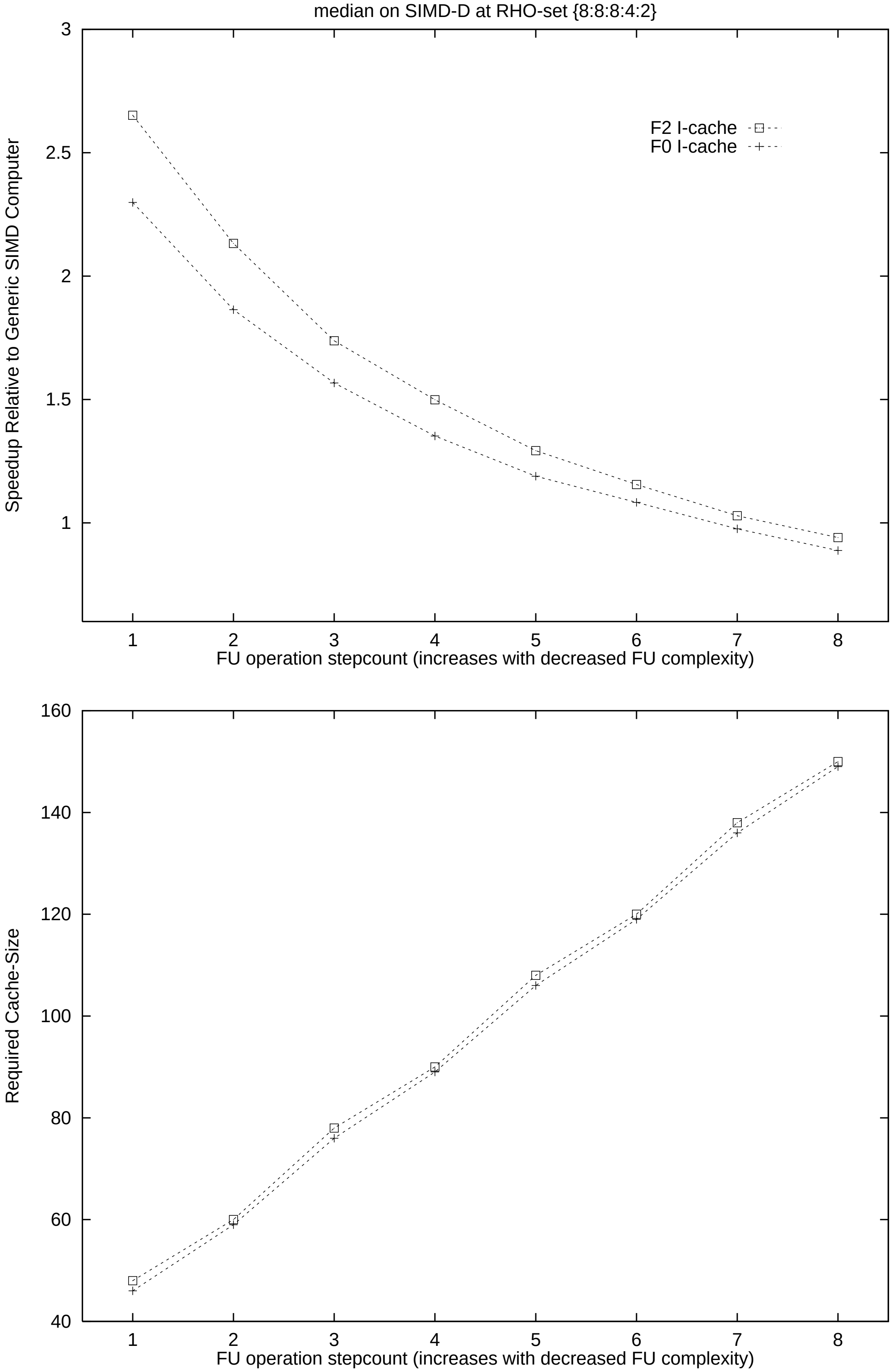


Figure 6.22: Speedup and Cache Size for **`median`** on SIMD-D v. FU Complexity at ρ-set {8,8,8,4,2}

6.7 Which Strategy Is Best?

There is no one Strategy for providing chip area for I-cache that is universally superior. The best Strategy or combination of Strategies depends on the requirements of the problem, the structure of the original computation, and the resources available in the PE chip.

Strategy 0, to use spare area for I-cache in the PE chip, has no impact on the PE chip's payload. Strategy 0 would thus seem ideal. Unfortunately, the high stakes in SIMD computer architecture for making good use of chip area lead to PE chips that typically have very little area left spare.

I-cache competes with PEs for chip area. Limited chip area introduces a limit to cache size, which in turn limits the number of repeat instructions that can be stored for subsequent retrieval at the high on-chip rate. Limited cache size also increases the quantization effect on I-cache speedup. A cache block can be iterated in cache only if it is wholly stored there. Some iterated instruction sequences are too long to fit entirely in a cache of limited cache size. Being able to store only a part of an iterated instruction sequence means that it cannot be iterated in cache. When the cache size is too small, the cache block iteration capability of I-cache variants including F_2 cannot be exploited, so quantization effects become apparent as they do for non-iterating I-cache variants including F_0.

Strategy 1, to displace PEs from the PE chip to make room for I-cache, increases the amount of work that is performed by the remaining PEs. Increasing the per-PE workload tends to increase iteration counts, which acts to increase the benefit of I-cache. However, a number of programming problems arise as a consequence of Strategy 1. For example, problem data may no longer fit into PE registers, thereby increasing the reliance on local external memory. Increased local memory usage tends to decrease I-cache speedup through increasing the local memory-boundedness of a computation. Strategy 1 has a surprising throughput-increasing effect: reducing the number of PEs in the PE chip reduces time-sharing of PE chip pins. Reduced pin time-sharing decreases the time taken to perform MCS operations, which tends to decrease subsystem-boundedness of computations. Less time spent waiting for MCS operations to complete means greater throughput, which in turn increases the apparent speedup due to I-cache.

Strategy 2, to remove PE registers to make room for I-cache, may have no effect or it may have an enormous effect. If a computation uses only a small number of registers, then the registers removed to make room for I-cache were redundant and are removed at no cost. However, a computation that uses all of the available PE registers makes greater use of local external memory under Strategy 2. Increased local memory-intensiveness means that the computation is more likely to be memory-bound, thus exhibiting reduced I-cache speedup.

Strategy 3, to remove some PE FU circuits to make room for I-cache, increases the number of PE clock cycles needed to perform some arithmetic operations. Strategy 3 makes computations more likely to be calculation-bound. Although I-cache is most useful for calculation-bound computations on the simulation model of SIMD computation, Strategy 3 yet degrades I-cache speedup. Strategy 3 has the additional negative consequence of drastically increasing the cache size needed to obtain maximum I-cache speedup by increasing the lengths of machine code instruction sequences used to control arithmetic operations.

In summary, the objectives of SIMD computer architecture lead to PE chips that are simplified to the point allowed by the calculation requirements of the problem mix. Unless the PE is overly complicated for a given problem, for example containing too many registers or unused single-cycle FU operations, further simplification of the PE as per Strategies 1, 2, or 3, is bound to reduce throughput.

On the other hand, the estimates of Section 4.3 indicate that I-cache with small cache size occupies negligible chip area in modern chips. Furthermore, the results of Chapter 5 show that, while small, simple I-cache variants indeed provide substantial speedups for the sample problems. Therefore, it is unlikely that a large proportion of the PE chip payload would need to be displaced to make room for I-cache that is useful for simple problems. However, there are problems that demand more complicated I-cache variants or larger caches. If the local controller with I-cache were to become very large, for example, displacing all but 1 of the PEs in the chip, then the resulting I-cached SIMD computer would become a tightly synchronized MIMD computer.

Chapter 7

Conclusion

The addition of I-cache to SIMD computers makes them faster. The speedup depends on program properties, PE architecture, chip characteristics, and the electrical characteristics of multi-chip subsystems. Detailed simulations of SIMD computations for a diverse collection of sample problems on a variety of hardware configurations show substantial speedups, even for simple I-cache variants.

Ideally, an I-cache is large enough to store entire iterated sequences, thereby attaining the speedup possible from controlling iterations within the PE chip. Also, the larger the cache memory, the less pronounced the thrashing due to conflict misses. Unfortunately, I-cache cannot be arbitrarily large because it occupies chip area that could otherwise have been used for PEs. The simulations show that small I-caches are useful for the subject problems. On this basis, it is reasonable to expect even small I-cache to be somewhat useful for larger problems. Detailed estimates indicate that I-cache containing 1000 32-bit instructions would occupy less than 5% of the chip area of a modern PE chip.

The appropriate complexity of the I-cache design clearly depends on program structure. For a simple loop with data-independent iteration count, a simple, statically managed I-cache variant exploits all of the potential speedup. A program containing multiple alternating repeated instruction sequences demands a slightly more complex I-cache variant. When inner loop iteration counts depend on outer-loop iteration index values, it is desirable to have a yet more complex I-cache variant that calculates iteration counts on its own. Finally, exploiting the maximum I-cache speedup for a program with arbitrary, data-dependent control flow may require a dynamically managed I-cache whose program-control complexity approaches that of the system controller itself. The best I-cache is determined by the requirements of the computation.

Predicting the I-cache speedup for an arbitrary high-level language program is difficult, due to the complex interactions among PE chip characteristics, system characteristics, I-cache capabilities, and program properties. Calculation-intensive loop bodies iterated large numbers of times make for large I-cache speedups. While programs that use MCSs intensively tend to exhibit lower I-cache speedups than those that don't, I-cache should yield some speedup. For example, the sample problems include some that are ordinarily considered to be inter-PE communication-bound. And yet the simulations show considerable I-cache speedups for all problems. This phenomenon occurs because a high-level programming language's "communication" operation is realized using machine code instructions that perform address calculations or context management operations. I-cache makes such calculations faster.

This analysis of I-cached SIMD computer architecture emphasizes the throughput performance metric and the chip-area hardware cost metric. For scalable data-parallel computations, wherein throughput is proportional to the number of PEs that are brought to bear, minimizing the chip area per PE maximizes the number of PEs in a given total chip area, thus maximizing throughput, all else being equal. Unfortunately, not all else is equal, because to drastically reduce the chip area of a PE by removing its program control is to introduce a limitation in the rate at which instructions are supplied. The measured I-cache speedups confirm the principle that to make the best use of chip area, a multiprocessor's PE chips should contain the right amount of redundantly replicated program control: Too much and there are fewer PEs than there could have been, too few and the PEs execute instructions too slowly.

This principle implies that a compromise between the MIMD and SIMD architectural extremes is valuable for data-parallel programs. In general, the electrical propagation characteristics of the technology used to fabricate the computer dictate the ideal way of distributing program control throughout the computer. The program control provided within each layer of the integration hierarchy (including chips, multi-chip modules, printed-circuit boards, racks, and so forth) should be simple so that the greatest proportion of resources are used for PE calculations and yet sufficiently powerful to be able to provide control within its own layer of the abstraction hierarchy at the highest rate attainable therein. In this light, the generally accepted taxonomic distinction between MIMD computers and SIMD computers as "equal" design alternatives appears to be misleading. Rather, SIMD computer architecture is the *specialization* of MIMD computers as appropriate for specific data-parallel computations.

7.1 Future Directions

This analysis introduces the issues pertaining in I-cache design and explores how properties of programs, systems, and chips interact in determining I-cache speedup. SIMD instruction cache is a new idea, and an I-cached SIMD computer has yet to be built. In establishing that even simple I-cache variants yield significant speedups over a collection of sample programs, large portions of an apparently vast design space have been left unexplored. Only a relatively small number of the many alternative physically structured variants of a subject computation have been measured, only a relatively small number of the many alternative transformations of the subject computation to reflect I-cache have been measured, and only a relatively small number of the many alternative physically structured variants of the I-cached computation have been measured. This section enumerates some of the areas in which to extend and improve the analysis.

7.1.1 Problem Characteristics

A natural and important extension is to analyze computations that are larger in scope than the sample problems used here. For example, programs whose loop structures are not statically analyzable due to data-dependence would provide a basis for evaluating dynamic cache management techniques.

Extending the study to more complicated problems would be facilitated by a high-level data-parallel language. High-level language programming introduces an array of compiler issues. It might prove valuable to quantitatively assess the dependence of I-cache speedup on the mapping between problem input data and PEs, or the interactions between I-cache speedup and compiler optimizations including register allocation and scheduling.

7.1.2 System Characteristics

For programs with complex loop structures, it may not be possible for a compiler to determine best which blocks to place in I-cache and when to store them there. A system controller component to perform these functions dynamically presents an interesting design challenge.

The simulation model used in the evaluation contains a single PE FU and a single instance of each type of MCS. Relaxing this assumption, for example so that the PE could contain multiple FUs perhaps operating at different rates, would allow the model to represent a wider range of SIMD computers.

The simulation model used in the evaluation incorporates synchronous elements throughout the computer. A multi-clock generator provides the clocks within the PE chip that coordinate the variously timed subsystems. The multi-clock generator design used in the basis computer is fairly clumsy. A good multi-clock generator is an interesting design problem.

7.1.3 I-Cache Characteristics

Although the simplest I-cache variants that have been studied in detail yield considerable speedups for the sample problems, more complicated programs require more complicated I-cache variants.

Beyond examining the properties of the members of the F-family beyond F_0 and F_2, it would also be interesting to evaluate I-caches with multiple ports. With multiple ports, one cache block can be stored while another is active, a form of prefetching that reduces the time spent waiting for cache blocks to be stored. Multi-port I-caching introduces the complexities of concurrent accesses to cache memory. The management problem here includes handling partial block stores, as will occur when a cache block finishes executing before another has been completely stored through the second port.

The identical I-cache speedups measured for F_0 and for F_2 on the **`bitonic`** sorting problem show that there is no advantage from F_2 in that case. The reason is that the iteration counts of the inner loop in **`bitonic`** depend on the value of the outer loop index. One way to overcome this limitation is to make it possible to use the system controller's index register subsystem to evaluate loop-index-dependent expressions to use in specifying iteration counts in F_2 cache block activations.

The low I-cache speedups shown for F_2 on the **`rowcol`** sorting problem arise from the inflexibility of iteration in F_2. The number of iterations of the inner loop in **`rowcol`** depends on the initial permutation of the data to be sorted, and this number of iterations differs each time around the outer loop. The iteration count for an F_2 cache block is specified when the block is activated. If a local counterpart of the response network were incorporated in the PE chip, then the local controller could be made to sense the completion condition for its chip's complement of PEs.

The possibility of the local controller sensing "global" data-dependent conditions among its group of PEs is one example of how conditional-control-flow programs would exploit I-cache. Execution in this case resembles pseudo-MIMD computation [10], wherein each PE chip operates independently as a SIMD computer in its own right during parts of a computation.

If the sequencing performed by the cache controller is able to depend on PE data conditions, then it becomes possible to incorporate data caching in SIMD computers. With D-cache, the time for a PE's local external memory access varies according to that PE's addressing pattern. If the cache controller is able to detect cache hits, it may sequence cache blocks appropriately for its complement of PEs. Inter-PE communication requires re-synchronization of the PE chips, which limits the benefit of D-caching. A characterization of the benefits of D-cache and the circumstances under which those benefits are realized might be valuable.

7.1.4 Evaluation Mechanism

Figure 5.1 illustrates the method used to evaluate I-cache speedup for sample problems. The translation step from assembly language programs to machine code programs uses timing information about operations to schedule machine code instructions. The scheduling algorithm attempts to overlap the execution of mutually flow-independent MCS and FU instructions where possible through reordering the instructions. As is apparent in Figures 6.3 through 6.10 showing the effects of limited cache size on speedup, the scheduler's attempt at optimization interacts in surprising ways with the cache size. Basic blocks reordering takes place prior to the assignment of instructions to instruction memory locations. When an instruction sequence turns out not to fit in cache memory, the reordering is undone in a conservative manner. The effect of the scheduler's simple algorithm for handling limited cache size shows up as the occasional non-monotonicity of I-cache speedup versus cache size apparent in Figures 6.3 through 6.10.

There is also room for improvement in the clocking assumptions. Each subsystem in the simulation model is regulated by a clock with a (potentially) unique rate. The multi-clock generator generates these multiple clocks at the requisite rates, subject to the following restrictions:

1. The PE clock, regulating the PEs and the local controller, is the fastest clock in the computer.
2. The system clock is the slowest clock in the computer.
3. All clocks are free-running.
4. All clocks are phase-locked to the system clock.
5. All clock rates are integer multiples of the system clock rate.
6. All clock rates are integer sub-multiples of the PE clock rate.

Each subsystem clock's interval can be no less than the duration of the subsystem's longest operation step. However, the restrictions listed above force the rate of a clock to be lower than necessary in some cases. Relaxing the arbitrary restrictions would allow a more complete exploration of I-cache speedup sensitivities to ρ-set values.

The restriction that the clocks are free-running appears to be an unfortunate design mistake on my part. For example, if the clocks were instead re-startable on an arbitrary cycle of the PE clock, then an otherwise-idle MCS could begin an operation on the earliest possible PE instruction. A free-running MCS clock introduces unnecessary delays in starting some MCS operations, because the instruction specifying the operation's commencement cannot be applied until the MCS clock and the PE clock are in-phase. The two clocks are in-phase only once every MCS clock cycle.

It would be interesting to extend the timing model to include asynchronous implementations of components. In principle, an asynchronous system need never operate at lower than the inherent maximum rate, subject to flow-dependencies. The impact of asynchrony would be particularly profound where operation step durations for a given subsystem vary widely.

7.2 How Important is SIMD Instruction Cache, Really?

For real problems and for realistic assumptions regarding the electrical properties of a high-PE-count SIMD computer, I-cache yields speedups of 30% to more than 700% over generic SIMD computation. From the high-level language programmer's point of view, the effect of I-cache is similar to that of increasing chip clock rates by many times, with concomitant speedups up to limits imposed by the requirements for inter-chip communication inherent in a given program.

As VLSI implementation technique continues to improve, the time to drive a lumped capacitance from the gate of a minimum inverter across a chip increases. It becomes less reasonable to view a chip as a single fast circuit domain and more reasonable to view a chip itself as a collection of fast circuit domains among which communication is slow or expensive. At some point in this scaling path, the time required to distribute instructions locally within the ever-larger PE chip itself becomes throughput-limiting. At that point, a single local controller in the PE chip can no longer provide instructions to the PEs at the maximum rate of PE operation. To counter this limitation, it will eventually become advantageous to re-apply I-caching *within the PE chip itself*, replicating enough local control within the chip to keep the PEs supplied with instructions at the maximum attainable operation rate.

The speedups possible from I-cache are limited to constant factors of perhaps 2 to 7 or so, far less than the asymptotic improvements available for some problems using parallelism. At first glance, I-cache appears to facilitate flexible encodings that increase the amount of control information provided to the PEs through an inherently slow channel. Another way to think of I-cache is as a means to exploit the throughput benefits of using large numbers of parts in parallel, while retaining the operation-rate advantages inherent in keeping the parts themselves small.

The sample problems were chosen for their expected dissimilarity. For example, sorting and tree-reduction are commonly thought to be inter-PE communication-bound, and thus might not be expected to yield much I-cache speedup, unlike matrix multiply, which is known to be calculation-intensive. For all of the sample problems, the benefits of I-cache more than compensate for the chip-area cost of enhancement. The consistency of the results across the range of problems points strongly to the conclusion that throughput is significantly higher in I-cached SIMD computers than in their generic counterparts, even for problems commonly thought to be subsystem-bound. The I-cache speedups for complete, practical applications could possibly be greater or less than those characterized here, depending on specific communication and calculation requirements relative to the FU and MCS characteristics of the underlying SIMD computer. Aside from characteristically simple loop structures, the sample programs are not extraordinary in their operation mixes.

The measured results reflect the assumption in the simulation model that there is no local control in the PE chip for the FU, although there is local control in the PE chip for the MCSs. A PE chip might contain local control for the FU, thus shortening instruction sequences and lessening the apparent I-cache speedup. Alternatively, a PE chip might contain no local control for MCSs, thus lengthening instruction sequences and increasing the payoff from I-cache. In any case, the measurements presented for SIMD-D, whose PEs perform 32-bit multiply in one clock cycle, are not affected by the lengthening of FU instruction sequences arising from the assumptions regarding local control.

Clearly, the coverage of this work is not exhaustive, and there exist many further avenues of research that follow from it. The results presented here have consequences for the analysis of problems solved by SIMD computers, and for languages and compilers used in describing those solutions. The measurement method provides a means for studying in detail with respect to specific problems the interactions among system controller, PE, MCS, and I-cache designs. These research avenues become compelling in light of the high stakes for providing instructions to PEs at the highest rates.

7.3 Final Comments

The analysis has been performed for VLSI-based computers. However, the reader may see that these results should apply with equal validity given any computer implementation technology wherein information is represented as energy that is spatially distributed in three dimensions. The simple underlying principle exploited in I-cached SIMD computer architecture is, "the more energy to be re-distributed per computation step, the larger the radius over which it is re-distributed, the slower or more expensive the computation." I-cache makes it possible to control large numbers of PEs that are packed as densely as possible within chips without having to re-broadcast repeated sequences of instructions through a relatively slow channel.

I-cache speedup is bounded above by ρ_b. ρ_b may not be much higher than 8 in a practical computer. This number is far less than the largest number of useful PEs in a multiprocessor. Therefore, the possible gain from adding I-cache to SIMD computers is not nearly so compelling as the possible gain from parallelism itself. This point is underscored by the observation that I-cached SIMD computers are generally useful only for data-parallel problems, a subset of the set of problems for which parallelism is advantageous.

In a generic SIMD PE chip, K times more chip area is allocated to FU, context manager, and data registers than in its same-technology MIMD counterpart. And yet, a generic SIMD computer's PEs operate at a rate ρ_{b} times lower than that of the corresponding MIMD computer. For a scalable data-parallel computation, the following relationship obtains, given a limited implementation budget with respect to total chip area:

$$\text{generic SIMD computer throughput} \quad \leq \quad \frac{K}{\rho_{\mathrm{b}}} * \text{MIMD computer throughput} \tag{7.1}$$

The "jury has been out" with respect to the relative merits of SIMD computers, and in fact it has recently been trickling in with a negative verdict. The reason might be that when $K \approx \rho_{\mathrm{b}}$, generic SIMD computer throughput is roughly equivalent to that of the MIMD counterpart, and SIMD computers are inherently more difficult to program. The main reason that SIMD computers have been attractive to some manufacturers would be that it is possible to produce a given-PE-count SIMD computer using much lower total chip area than for a MIMD counterpart.

I-cache overcomes the instruction delivery limitation that contributes the factor-of-ρ_{b} denominator in Equation 7.1. If the PE chip is allowed to expand slightly to accommodate I-cache sufficiently large to obtain a speedup of nearly ρ_{b} for a given computation, then the following relationship obtains:

$$\text{I-cached SIMD computer throughput} \quad \leq \quad K * \text{MIMD computer throughput} \tag{7.2}$$

This comparison on the basis of throughput and chip area notably neglects factors such as market size that impact monetary cost of chip design and fabrication. Even if I-cached SIMD computers exhibit the highest throughput-to-area ratios, they are not necessarily preferable, even for scalable data-parallel problems. The SIMD PE chip is a low-volume part, whereas MIMD computers are often made using PEs that are microprocessors fabricated in medium-to-high volumes. Economies of scale lead to a unit cost for the SIMD PE chip which is a factor of L times higher than that of the MIMD PE chip. Under the simple assumption that computer cost scales linearly with L, the throughput per cost ratio of the SIMD computer is no more than $\frac{K}{L}$ that of the MIMD computer. If L is as large as or larger than K, then SIMD computer architecture can be justified only for applications for which making the best use of total chip area is paramount. It is for these area-critical applications that I-cached SIMD computer architecture is a compelling choice.

Appendix A

The Basis Computer

Rather than having been evaluated for a single specific SIMD computer, I-cache variants have been evaluated for a range of SIMD computers. A parameterized SIMD computer was designed and used as a basis for the evaluations. This appendix describes the machine code programming of the basis computer and highlights the "machine-dependent" aspects of its design that affect I-cache evaluation.

Given the large number and variety of SIMD computers that have been published since the Solomon computer was described in 1962 [74], such generality would seem intractable. Fortunately, the task is simplified by the observation that only a relatively small number of VLSI-based SIMD computers have been reported to date, including Vastor [87], CAAPP [81, 82], SLAP [26], Blitzen [37], and MP-1 [33, 8]. In a VLSI-based SIMD computer, the PE is an integrated circuit capable of performing calculations within the confines of a PE chip. This definition of a VLSI-based SIMD computer requires that the PE's FU component be packaged in a chip along with some amount of register memory. This restriction rules out candidates such as CM-2, whose bit-serial PEs require 3 off-chip memory references to perform a single full adder step [18].

The simulated computer's generality causes some of the details to be more intricate than would be required in an actual design. The generalization is less than perfect, and the design reflects some assumptions about specific subsystems. While the details of actual and foreseeable SIMD computers vary over a large space, the design of the basis computer provides a general understanding of the mechanisms that are involved in I-cache.

A.1 PE

The PE contains an FU, register memory, a context manager, and interfaces to the MCSs. The PE components are inter-connected by busses. An actual PE might depart from this basic design, for example by having multiple specialized function units or by having point-to-point internal interconnections rather than busses. The machine-code language programmer's view of the PE is sketched in Figure A.1, showing the busses inter-connecting the register file, the FU, and the MCS interface registers.

An important characteristic of the generic computer used in the experiments is that local control is not provided in the PE chip for the FU, whereas local control *is* provided for the MCSs. This aspect of the design of an ostensibly generic computer is justified by the observation that the local control of an MCS is potentially simple, whereas the local control of the FU is potentially complex. For example, compare the local external memory subsystem local control illustrated in Figure A.2 against one that generates the sequence of control signals for multiplying 32-bit floating-point numbers on a 4-bit PE. This assertion regarding PE chip local control appears liberal from the point of view of the goals of the experiment, because it causes FU calculations to appear instruction delivery-rate bound. However, recall that the SIMD idea is to remove as much redundant local control from the PE chip

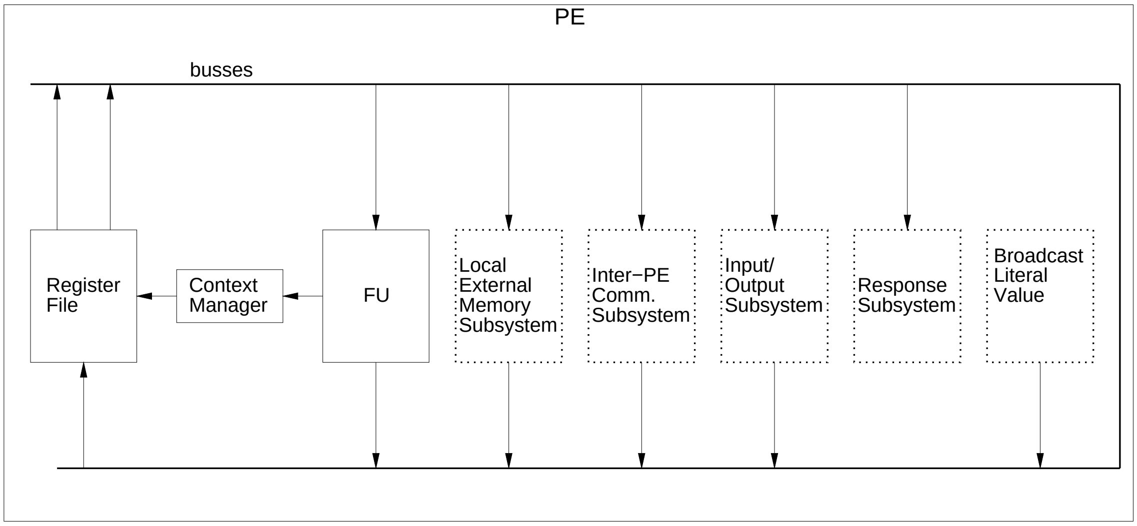


Figure A.1: PE Architectural Components. Dotted lines indicate MCSs to which the PE is connected through interface registers.

as practicable. In the sense that redundant local control is provided in the PE chip so that the MCSs tend *not* to be instruction delivery-rate limited, this assertion is actually conservative.

A.2 Local External Memory Subsystem

The PEs within a PE chip share a single access port to local external memory. While this assumption is reasonable for wide-word PEs and for PEs that provide their own addresses to memory, it does not accurately model local external memory access in SIMD computers whose PEs cannot provide their own addresses to memory. (Use of a single memory address for all PEs, as in CM-2 [22], reduces the PE chip pin cost of local external memory.) The assumption that the PEs generate local external memory addresses restricts the applicability of the resulting measurements to systems whose PEs have that capability.

Figure A.2 shows how the local external memory subsystem is organized. The small control circuit inside the PE chip illustrates the potential simplicity of MCS control.

A.3 System Controller

The primary components of the system controller are a microcoded sequencer and a mechanism for evaluating loop-index-dependent expressions. The system controller is intended to provide the basic required control functions required using a small set of single-cycle operations. An actual system controller may be optimized for specific computations being performed, as for example is the case in SLAP [28]. The potentially crucial topic of SIMD system controller design is beyond the scope of the thesis.

The system controller is shown in Figure A.3. The system controller is partitioned into subsystems that perform the functions enumerated in the following list:

1. Generate the system clock (system clock generator),
2. Store the program controlling the computation (instruction memory),
3. Sequence the control program (sequencer),

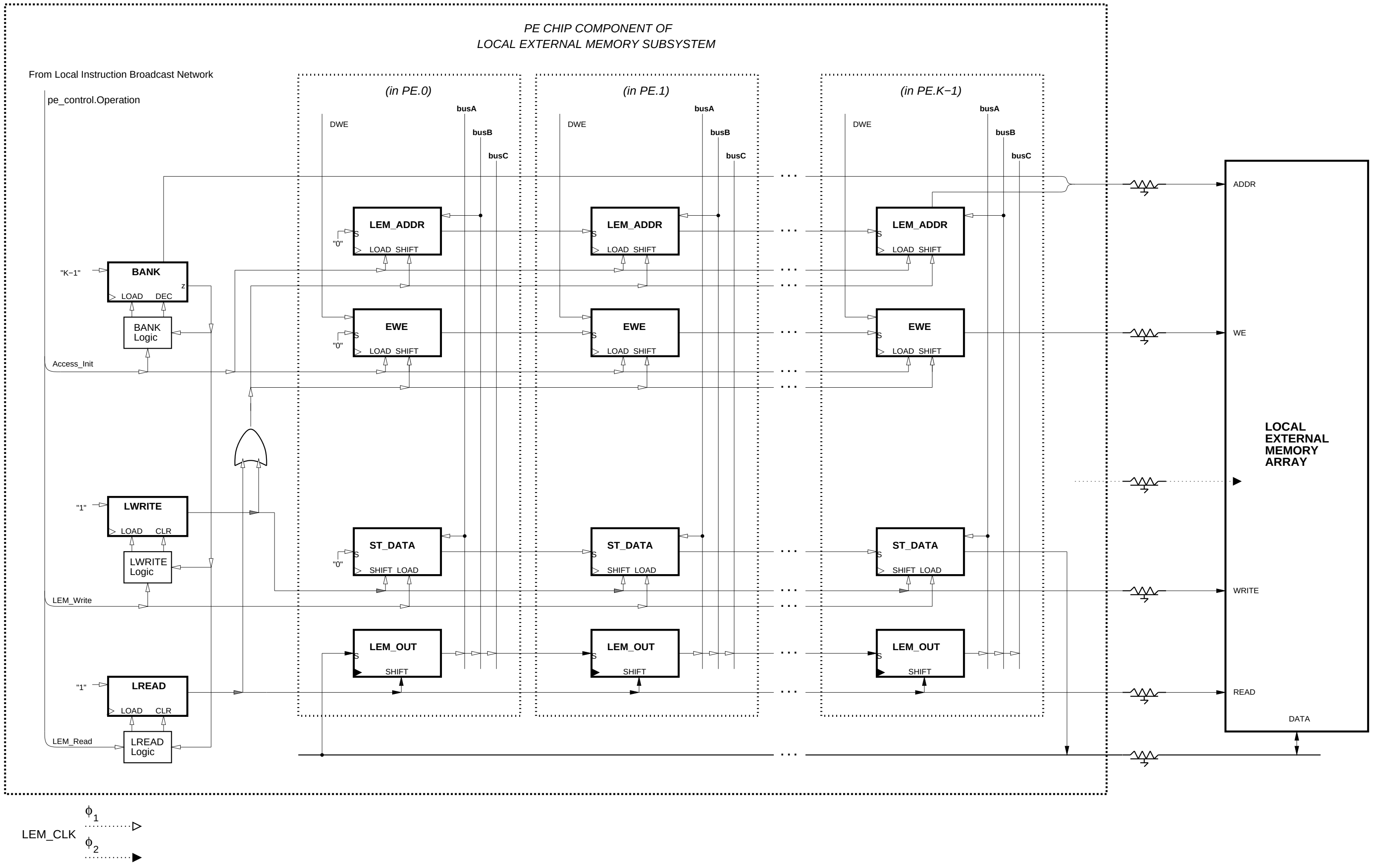


Figure A.2: Local External Memory Subsystem

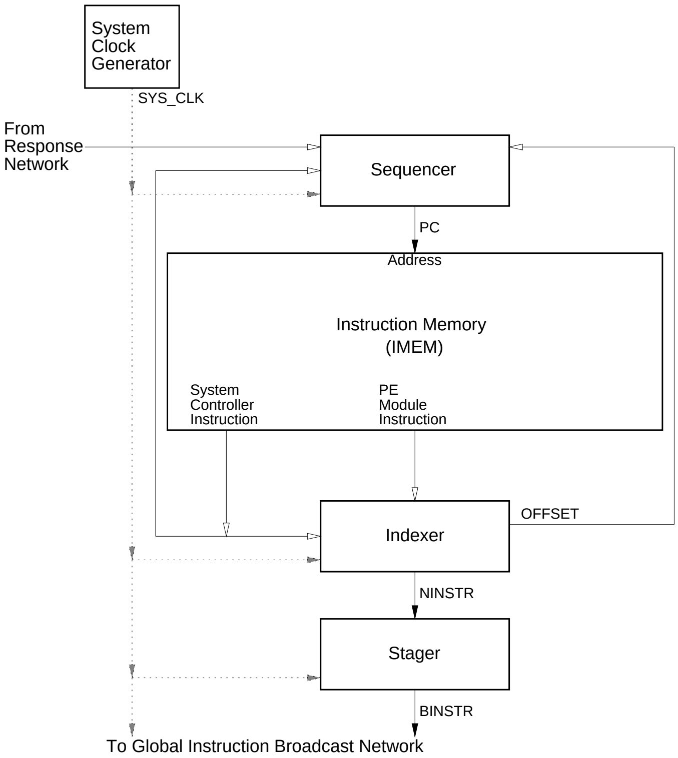


Figure A.3: System Controller

4. Evaluate loop-index-dependent expressions and inject the calculated values into PE chip machine code instructions as required (indexer),

5. Gather components from successive machine code instructions to form a broadcast instruction that controls a single cycle of PE activity (stager).

A.4 Machine Code Programming Language

A machine code program for the basis computer is a table of instructions. Each row of the table has an index (increasing from 0) and contains a machine code instruction. Figure C.4 contains an example of a machine code program. The machine code instruction fields are shown in Figure A.4. Each field of a machine code instruction takes a numeric value, or equivalently the mnemonic representation of a numeric value. A field whose value is not specified takes a standard "null" value. As indicated in Figure A.4, each machine code instruction has two parts: a system controller part and a PE part.

System Controller Instruction				(proto-) PE Instruction					
SC_Operation	f1	f2	f3	C	Dest	Operation	SrcA	SrcB	Literal_Base_Value

Figure A.4: machine code Instruction Word Components

A.4.1 System Controller Instructions

There are two classes of system controller instruction, as selected by the **SC_Operation** field: sequencer instructions and indexer instructions. These two types of instructions differ as follows:

- Sequencer instructions resemble those of a typical microprogram controller (such as, for example, the 49C410 [45]). Sequencer instructions may specify conditional branches or subroutine calls and returns. The standard "null" sequencer instruction specifies that the next instruction is the one following the present instruction in linear sequence in the program. An indexer instruction implies the standard "null" sequencer instruction.
- Indexer instructions provide for initializing, copying, and incrementing members of a set of index registers. A value produced by an indexer instruction is added to the PE instruction's **Literal_Base_Value** to form a literal for broadcast to the PEs. The standard "null" indexer instruction does not alter any index registers and produces the value 0. A sequencer instruction implies the standard "null" indexer instruction.

The system controller part of the stored machine code instruction specifies an operation of either the sequencer or the indexer. These operations are horizontally microcoded. The encodings of the four fields of the system controller part of the stored instruction are shown in Figure A.5. When the **`SC_Operation`** field specifies an indexer operation, the implied sequencer operation is ***CONT***; when the **`SC_Operation`** field specifies a sequencer operation, no indexer operation is performed.

	SC_Operation	f1	f2	f3
Sequencer Operation	CJSR	Condition Select	Branch Target	Initial Iteration Count
	LTST	Condition Select	Branch Target	
	CBR	Condition Select	Branch Target	
	CONT			
	HALT			
Indexer Operation	LDX	Write Address	Index Literal	
	SPX	Write Address	Index Literal	
	CPX	Write Address	Read Address	

Figure A.5: System Controller Instruction Word Components

There are a small number of sequencer operations, any one of which can be specified in the **SC_Operation** field of the system controller instruction word shown in Figure A.5. The mnemonics for the sequencer operations are given as follows, along with a description of how each operation affects program control flow:

1. **CJSR** (Conditional Jump to SubRoutine). If the condition selected in field **f1** is true, then push (**PC**)+ 1 onto the **PC** stack, write the branch target address from field **f2** into **PC**, push (**IC**) onto the **IC** stack, and write the initial iteration count value from field **f3** into **IC**. If the condition selected in field **f1** is false, then write (**PC**)+ 1 into **PC**.
2. **LTST** (Loop TeST). If the condition selected in field **f1** is true, then write the top of the **PC** stack into **PC**, pop the **PC** stack, write the top of the **IC** stack into **IC**, and pop the **IC** stack. If the condition selected in field **f1** is false, then decrement **IC** and write the branch target from field **f2** into **PC**.
3. **CBR** (Conditional BRanch). If the condition selected in field **f1** is true, store the branch target from field **f2** into **PC**. If the condition selected in field **f1** is false, store (**PC**)+ 1 into **PC**.

4. **CONT** (CONTinue). Store (**PC**)**+** 1 into **PC**.

5. **HALT**. Terminate the computation.

If the system controller operation specified in the **SC_Operation** field is not among those listed above, the sequencer operation is taken implicitly to be ***CONT***.

For sequencer operations, system controller instruction word field **f1** controls the condition selector **ccmux**. The mnemonic values for this field are as follows, along with their interpretations:

- **RSP0**. Select input 0, the **RESPONSE ==**0 condition.

- **FORC**. Select input 1, always true.

- **ICT0**. Select input 2, the **IC ==**0 condition.

Of the four fields in the system controller instruction word shown in Figure A.5, only the first three control indexer operations. There are only three system controller index register subsystem operations that can be selected in the **SC_Operation** field. The mnemonics for these operations are given as follows, along with descriptions of how they work:

1. **LDX** (LoaD indeX register). Write the value contained in field **f2** into the index register addressed in field **f1**.

2. **SPX** (SteP indeX register). Increment the index register addressed in field **f1** by the value in field **f2**.

3. **CPX** (CoPy indeX register). Copy the contents of the index register addressed in field **f2** into the index register addressed in field **f1**.

Any other value of the **SC_Operation** field leaves all index register file locations unaltered.

A.4.2 PE Machine Code Instruction

The PE has a register-to-register instruction set with explicit operations used to access off-PE-chip data via the MCSs. Each PE instruction specifies an operation, the register addresses of two sources and of a result destination, a context operation, and a literal base value. The standard "null" operation is ***NO_OP***, the standard "null" register address is ***NO_LOC***, and the standard "null" context operation is ***NO_CO***. In other words, a machine code instruction in which every field has the standard "null" value is a *null instruction* that leaves the PE state unchanged.

The PE machine code instruction has the following general form:

```
Dest = Operation(SrcA,SrcB)
```

The sources and destination of a PE machine code instruction are PE registers, so the machine code instruction specifies a register-to-register operation. The PE machine code instruction includes the fields specified in Figure A.6.

The set of PE operations is large, including the ordinary two-operand arithmetic and logical operations as well as operations using the MCSs. Following standard uniprocessor design practice, instruction execution is pipelined. The PE executes instructions in the three-stage pipeline illustrated in Figure A.7.

In a pipelined PE, some instructions begun on successive PE clock cycles exhibit *pipeline hazards*. A pipeline hazard arises, for example, when a second instruction is flow-dependent on the immediately preceding instruction. (Such a pipeline hazard is called a *destination-source pipeline*

C	Dest	Operation			SrcA	SrcB	Literal_Base_Value
		Op_Code	Op_Cycle	new_op			

Figure A.6: PE Machine Code Instruction Word Components

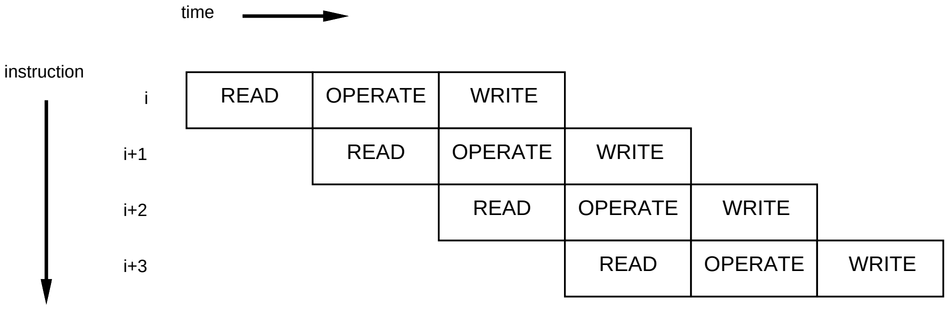


Figure A.7: PE Execution Pipeline

conflict [34]). Some pipeline hazards necessitate the second instruction being delayed, so as to allow time for its operand to be produced by the first instruction. Failure to do so yields an incorrect result. In the basis computer, the PE contains no pipeline interlocks, so machine code programs containing pipeline hazards are not allowed.

An MCS operation which requires more than one subsystem clock cycle to be performed may be *phase-split*: An initiating instruction starts the operation, and a terminating instruction completes the operation. The initiating instruction specifies the operands and the operation itself, while the terminating instruction specifies the destination into which to write the result.

Phase-splitting in machine code programs expresses the overlap of high-latency MCS operation with operations on other MCSs or on the FU. For machine code programs describing generic SIMD computations, the number of instructions intervening between the pair specifying a phase-split MCS operation is exactly 2 less than the number of clock cycles required for the operation. Instructions intervening between the pair may specify FU operations or operations on MCSs other than the one in use by the pair. An operation specified in one of these intervening instructions overlaps with the outstanding operation. Where overlap cannot occur, for example because no flow-independent instruction is available, the time between the initiating and terminating instructions is spent waiting for the long-duration MCS operation to complete.

Some MCS operations return no value to the PE. These operations include local external memory store, system data memory store, and response. These operations are phase-split using only a single initiating instruction. Although no instructions after the initiating instruction are needed to perform the operation, no subsequent instruction may specify another operation on the busy MCS until the current operation completes.

The only operation using the instruction broadcast subsystem itself is the ***LITERAL*** operation that delivers an indexer-calculated value to the PEs. This operation takes no source operands. Because literals are delivered via the global instruction broadcast network, they are always instruction delivery-rate limited. Therefore even if its latency is high, a ***LITERAL*** operation cannot be phase-split.

The fields of the PE instruction shown in Figure A.6 are interpreted as follows:

- **C** (Context). This field specifies a context operation from the following set (creation of a new context uses an operation specified in the ***Operation*** field):
  - **NO_CO** No context operation.

  - **FRC** Force modification of this instruction's **Dest** irrespective of the current context.
  - **POP** Revert to the previous context.
  - **INV** Invert the sense of the current context.
  - **CLR** Reset context to its power-up state.

- **Dest** (Destination). This field specifies the PE location to be written with the result of the instruction's operation via busC, if permitted by the current context. Possible destinations are among the following:
  - **NO_LOC**. Nowhere.
  - **REG.i**. Location i in the register file.
  - **PTR** (Pointer). The register file's **POINTER** register.
  - **IND** (Indirect). The register whose address is contained in **POINTER**.

- **SrcA** and **SrcB** (Source). These fields specify the locations providing operands via busA and busB, respectively. Possible sources are as follows:
  - **NO_LOC**. No operand is required on this bus for this instruction.
  - **REG.i**. The operand is read from location i in the register file.
  - **PTR** (Pointer). The operand is read directly from **POINTER**, the index register available for addressing the register file.
  - **IND** (Indirect). The operand is read from the register whose address is contained in **POINTER**.
  - **LIT** (Short literal). The operand is supplied as a literal contained in this instruction field. (For implementation economy, only **SrcA** can be used for short literals.)
  - **FU_OUT** The operand is supplied from the FU output register.
  - **LIT_OUT** The operand is supplied from the **LIT_OUT** register, containing the most recently received broadcast literal value.
  - **LEM_OUT** The operand is supplied from the local external memory circuit output register.
  - **COM_OUT** The operand is supplied from the inter-PE communication circuit output register.
  - **IO_OUT** The operand is supplied from the system data memory circuit output register.

- **Lit_Value**. This field carries a constant to be stored or operated upon within the PE.

- **Operation**. This field specifies activity in the FU or in one of the MCSs. This field designates a unit to perform an operation and controls the latching of that unit's input registers. There are three sub-fields of the **Operation** field:
  - **Op_Code** This sub-field names the operation to be performed, and by implication the unit (FU or MCS) that performs it. When an operation completes, the result remains in the unit's output register until a subsequent operation is performed on that unit.

    A list of possible operation codes follows, grouped according to the unit that performs the operation:
    - FU: ***PASS, NOT, AND, OR, NAND, NOR, XOR, ADD, SUB, MULT, DIV, MOD, LSHIFT,*** and ***RSHIFT***.

  - FU and context manager: ***LC_PUSH_LT, LC_PUSH_LE, LC_PUSH_EQ, LC_PUSH_NE, LC_PUSH_GE,*** and ***LC_PUSH_GT***. The FU subtracts the two operands, and the context manager generates a new context on the basis of the condition codes set by the subtraction.
  - Local external memory subsystem: ***LOAD, LOAD_TX, LOAD_RX,*** and ***STORE***.
  - Inter-PE communication subsystem:
    - Linear array: ***LDN0, LDN0_TX, LDN0_RX, LUP0, LUP0_TX,*** and ***LUP0_RX***,
    - Square mesh: ***SDN0, SDN0_TX, SDN0_RX, SUP0, SUP0_TX, SUP0_RX, SDN1, SDN1_TX, SDN1_RX, SUP1, SUP1_TX,*** and ***SUP1_RX***,
    - Cubic mesh: ***CDN0, CDN0_TX, CDN0_RX, CUP0, CUP0_TX, CUP0_RX, CDN1, CDN1_TX, CDN1_RX, CUP1, CUP1_TX, CUP1_RX, CDN2, CDN2_TX, CDN2_RX, CUP2, CUP2_TX,*** and ***CUP2_RX***,
    - Router-based network: ***ROUTE, ROUTE_TX,*** and ***ROUTE_RX***
  - System data memory subsystem: ***IO_LD, IO_LD_TX, IO_LD_RX, IO_ST***
  - Response subsystem: ***RESPOND***
  - Literal: ***LITERAL***

- **Op_Cycle** This field tracks the step index in a multi-step FU operation. Given a value $S-1$ on the first step of an S-step sequence, this field is decremented by one on each successive instruction. The steps intervening between the first and the last are place holders for the specific operations that would be performed on an actual PE.
- **new_op** This one-bit field indicates whether the input registers of the unit designated by the ***Op_Code*** should latch the operands. This field is un-asserted for all but the first step of a multi-cycle FU operation sequence.

A.5 PE Chip Local Controller

Adding I-cache to a SIMD computer really means changing the local controller design. The local controller in the PE chip of a generic SIMD computer, shown in Figure A.8, is very simple.

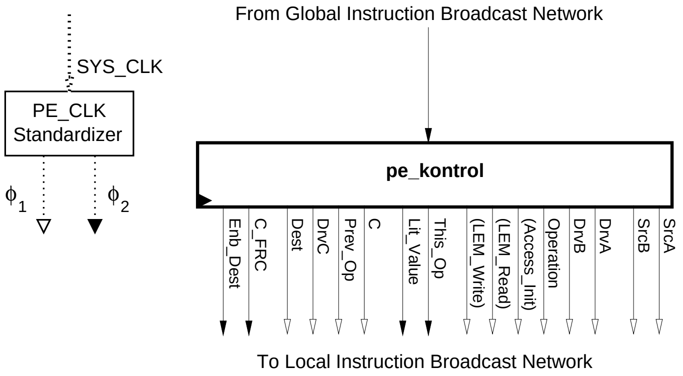


Figure A.8: Local Controller for Generic SIMD Computer

The local controller of a generic SIMD computer standardizes the system clock and latches broadcast instructions for use within the PE chip.

Adding I-cache to the PE chip involves a multi-clock generator and a cache controller. The multi-clock generator provides all clocks needed in the PE chip, including the PE clock. The PE clock rate exceeds the system clock rate, so there are multiple cycles of the PE clock per cycle of the system clock. In addition to controlling cache memory, the cache controller provides a new instruction for

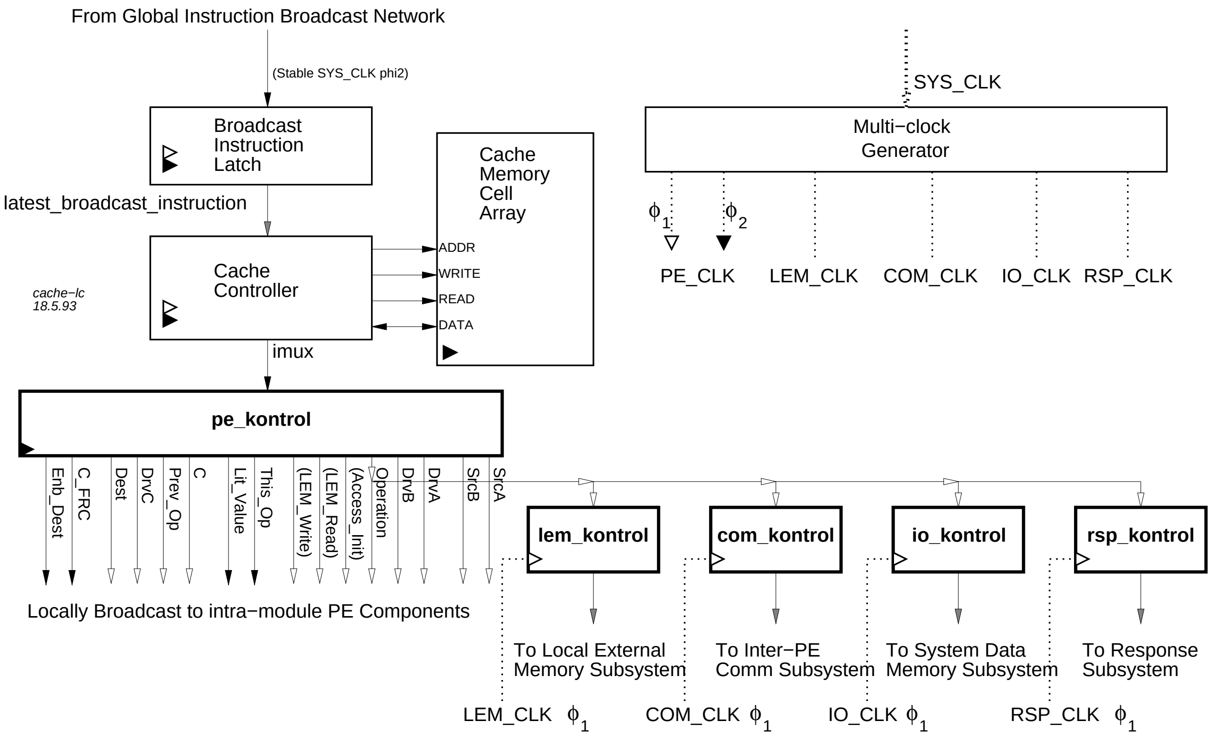


Figure A.9: Local Controller with I-Cache

local broadcast within the PE chip on each cycle of the PE clock. A local controller with I-cache is shown in Figure A.9.

A consequence of the variety of time bases within the PE chip is that correspondingly clocked control words need to be provided to each of the separately clocked subsystems. Figure A.9 shows a separate control latch (_**kontrol**) for each subsystem. Note however that providing the PEs and each of the MCSs a unique clock is a change only in the routing of clock signals to those subsystems, rather than a change in the logic of the clocked elements themselves.

The model allows that each MCS may have its own clock rate. It is possible, however, that subsets of the MCS clocks are unified. For example, where the I/O subsystem and the response subsystem both operate at the global instruction broadcast rate, both those MCSs are regulated by the system clock, just as they are in the generic SIMD computer. Another possibility is that local external memory and inter-PE communication operate at the same rate as the PE's intra-chip components, in which case these MCSs are regulated by the PE clock. The presentation here assumes the most general case, where each subsystem's clock rate is unique.

The PE clock, **PE_CLK**, is the fastest clock in the PE chip and in the computer. **PE_CLK** regulates the components that are integrated entirely within the PE chip, including the local controller, the local instruction broadcast network, and the PEs. This assertion about **PE_CLK** reflects an assumption that the PE chip is an equipotential region (as defined in [69]) within which signaling delays are negligible.

There are 5 clocks other than **PE_CLK** in the computer:

- **SYS_CLK**, the clock regulating the global instruction broadcast network and, for simplicity, the system controller. **PE_CLK** runs ρ_b times faster than **SYS_CLK**.

- **LEM_CLK**, the clock regulating the local external memory subsystem. **PE_CLK** runs ρ_l times faster than **LEM_CLK**.

- **COM_CLK**, the clock regulating the inter-PE communication subsystem. **PE_CLK** runs ρ_c times faster than **COM_CLK**.

- **IO_CLK**, the clock regulating the system data memory subsystem. **PE_CLK** runs ρ_i times faster than **IO_CLK**.

- **RSP_CLK**, the clock regulating the response subsystem. **PE_CLK** runs ρ_r times faster than **RSP_CLK**.

One way to generate high-rate PE clocks is to use phase-locked loop (PLL) techniques, such as those described in [88]. PLLs for generating high-rate on-chip clocks are increasingly frequently used in microprocessors due to the increasing disparity between typical intra-chip and inter-chip signaling rates [5, 88].

The multi-clock generator used in the basis computer outputs a set of clocks that are free-running and in-phase with **SYS_CLK**. The ρ value associated with each subsystem expresses the factor by which **PE_CLK** is faster than that subsystem's clock. Section 3.9 introduces a ρ-set that characterizes a SIMD computer's relative rates of MCS operation. The design of the multi-clock generator imposes the following constraints on the ρ-sets that characterize the SIMD computers for which I-cache is evaluated:

1. Each ρ-set value is a positive integer ≥ 1, and
2. ρ_b is an integer multiple of every ρ-set value.

These constraints mean that each MCS clock rate is an integer sub-multiple of the PE clock rate and that each MCS clock rate is an integer multiple of the system clock rate. While simplifying the systematic variation of the ρ-sets in evaluating I-cache variants, these constraints unfortunately also quantize the space of possible ρ-sets. An example of this quantization is that if the PE clock rate is A times higher than the top operation rate of the inter-PE communication subsystem (ρ_c=A), while the PE clock rate is B times higher than the top operation rate of the system data memory I/O subsystem (ρ_i=B), and A and B happen to be mutually prime, then the PE clock rate must be some multiple of $A * B$ times faster than global instruction broadcast (ρ_b=nAB for some n). In other words, the ρ values should be independent variables, but they are made inter-dependent by the multi-clock generator design.

A.6 Changed Globally Broadcast Instruction Format

The multi-clock generator provides a time base inside the PE chip that is higher than the time base of the global instruction broadcast subsystem that ultimately controls the PE chip. So that the globally broadcast instructions may specify activity within the PE chip at the higher temporal resolution, a new field is added to the broadcast instruction. This field, called **delayed_instruction_delay_count** (or **didc** for short), specifies a number. **didc** is the number of PE clock cycles for which the local controller is to wait before applying the destination write-control information conveyed in a broadcast instruction. The **didc** field is necessary because it is possible, depending on operation stepcounts and the ρ-set, for an MCS operation to conclude on an arbitrary cycle of the fast PE chip clock. The **didc** field essentially provides the index of this cycle, such that a result returned by an MCS operation is stored in the PE's register memory on the cycle it becomes available. Note that the addition of the **didc** field to the global broadcast instruction word does not affect the instruction broadcast locally within the PE chip, nor does it affect the PE itself.

Note that the **didc** field is needed because **PE_CLK** is faster than **SYS_CLK**, not because of I-cache itself. However, the cache controller, which sequences instructions to the PEs, also interprets the new **didc** field appropriately.

A.7 Two I-Cache Variants: F_0 and F_2

The F-family of single-port caches is introduced in Section 4.2. An F_0 I-cache is the simplest family member, able to store only a single cache block at a time, and able to execute only single iterations of the stored cache block. F_2 is a slightly more powerful I-cache variant, still able to store only a single cache block at a time, but able to execute multiple iterations of the block without assistance from globally broadcast instructions.

The F_0 and F_2 cache-control protocols are almost identical, each including the following four cache-control instructions:

- ***CC_NOOP***. No cache control operation.
- ***CC_BSTO*** (Begin STOring). The globally broadcast instruction following this one is the first in a cache block about to be stored. That next broadcast instruction will be placed at address 0 in cache memory, with subsequent broadcast instructions stored to subsequent locations in cache memory.
- ***CC_ESTO*** (End STOring). The present globally broadcast instruction is the last in the cache block currently being stored. The instruction specifying ***CC_ESTO*** serves as a sentinel delimiting the end of a cache block, and is itself placed in the cache.
- ***CC_FORK***. Activate the previously stored cache block.

An F_2 I-cache is identical to an F_0 I-cache, except that an F_2 I-cache's ***CC_FORK*** instruction also specifies the number of times that the routine is to be iterated. The cache-control instruction associated with any globally broadcast instruction not in the above list is interpreted as ***CC_NOOP***.

The cache controller generates the control signals necessary for accessing the cache memory and selects locally broadcast instructions through **imux**. On each **PE_CLK** cycle, the cache controller is in one of six states. The intended meanings of these cache control states are described below:

- **LOCK** No cache block is active, and globally broadcast instructions are being executed. The cache controller supplies $\rho_b - 1$ "null" instructions after every globally broadcast instruction received while in the LOCK state. In this state, instructions are delivered to the PEs at the same rate as in a generic SIMD computer, although the presence of fast subsystem clocks allows some subsystems to run faster than in the generic SIMD computer.
- **BSTO** The next broadcast instruction will be the first of the cache block to be stored in cache memory.
- **STOR** Globally broadcast instructions are being stored in consecutive cache memory locations. The cache controller supplies ρ_b "null" instructions on every PE clock cycle while in the STOR state. No useful instructions are executed by the PEs in this state.
- **ESTO** The sentinel ***CC_ESTO*** instruction has been received, indicating that the entire cache block has been stored in cache memory.
- **EXEC** A cache block is active. The cache controller supplies an instruction from cache memory on every cycle of **PE_CLK** in this state. Up to ρ_b instructions are executed by the PEs during every cycle of **SYS_CLK** while in the EXEC state.
- **JOIN** Execution of a cache block has completed, but the subsequent global broadcast instruction has not yet arrived at the PE chip. The cache controller supplies "null" instructions while in the JOIN state. Cycles spent in the JOIN state are those wasted due to quantization; at most $\rho_b - 1$ cycles are spent in the JOIN state per cache block activation.

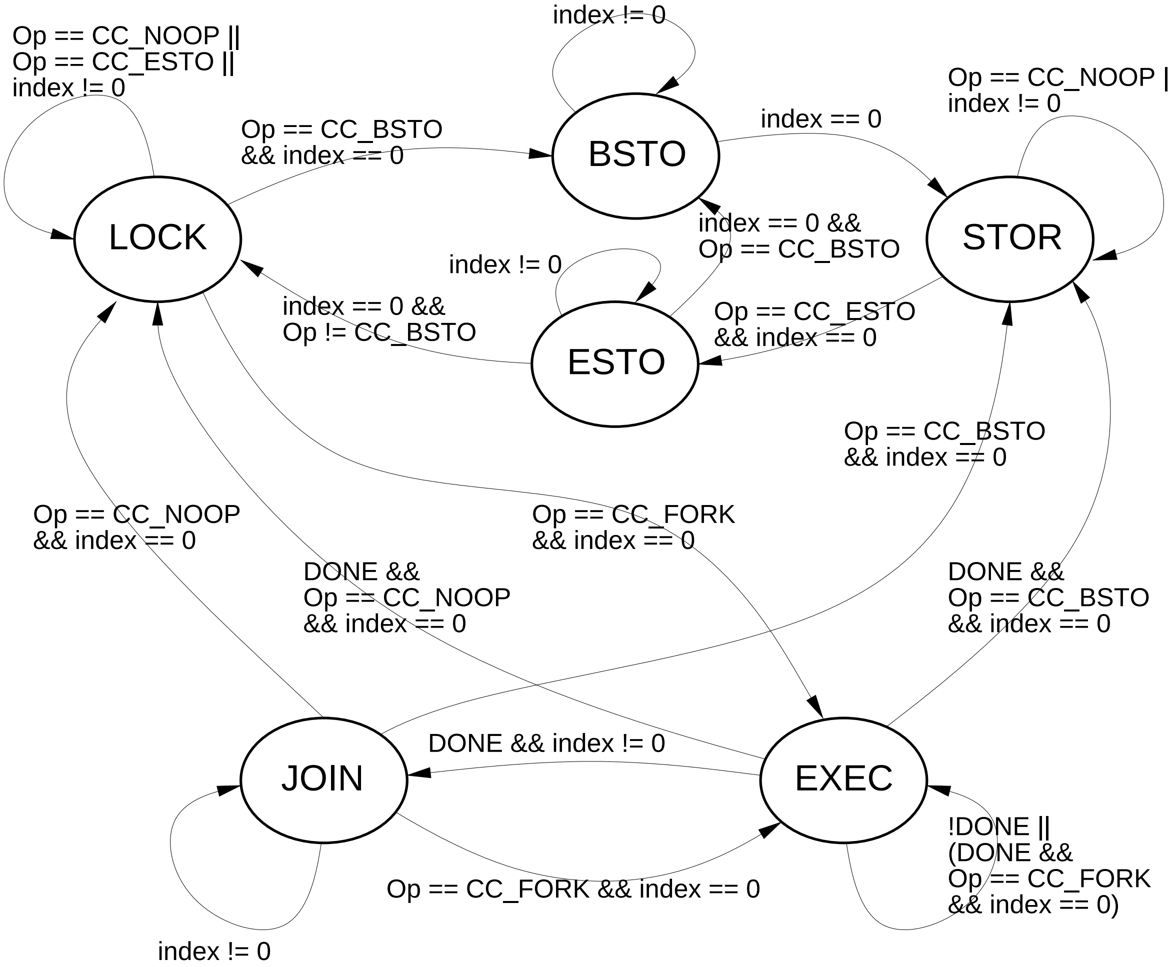


Figure A.10: F_0 State Transition Diagram. F_2 state transitions are slightly more complicated, because the iteration counter value is used in determining completion of a cache block's execution.

The power-up state is **LOCK**.

The diagram in Figure A.10 shows the allowed state transitions of the F_0 cache controller.

The labels on the arcs in Figure A.10 refer to the following values:

- **Op** is the cache operation contained in the current instruction's **Operation** field **Op_Code** subfield.
- **index** is the current phase of **`PE_CLK`** with respect to **`SYS_CLK`**, which is the value in the **PE_CLK_index** register.
- **DONE** is a Boolean value indicating completion of the execution of a cached instruction sequence.

Figure A.10 shows that cache controller state changes usually occur when **index** == 0.

Timing delays are represented as sequences of "null" instructions. These sequences can be expensive both in terms of time to place them in cache as well as in terms of the space they occupy in cache. Sequences of "null" instructions are compactly encoded using a single instruction that causes the cache controller's program counter **CPC** to stall for a number of cycles equal to the number of encoded "null" instructions. In principle, these sequences representing timing delays are unnecessary, although some equivalent means of representing delays is needed in any case.

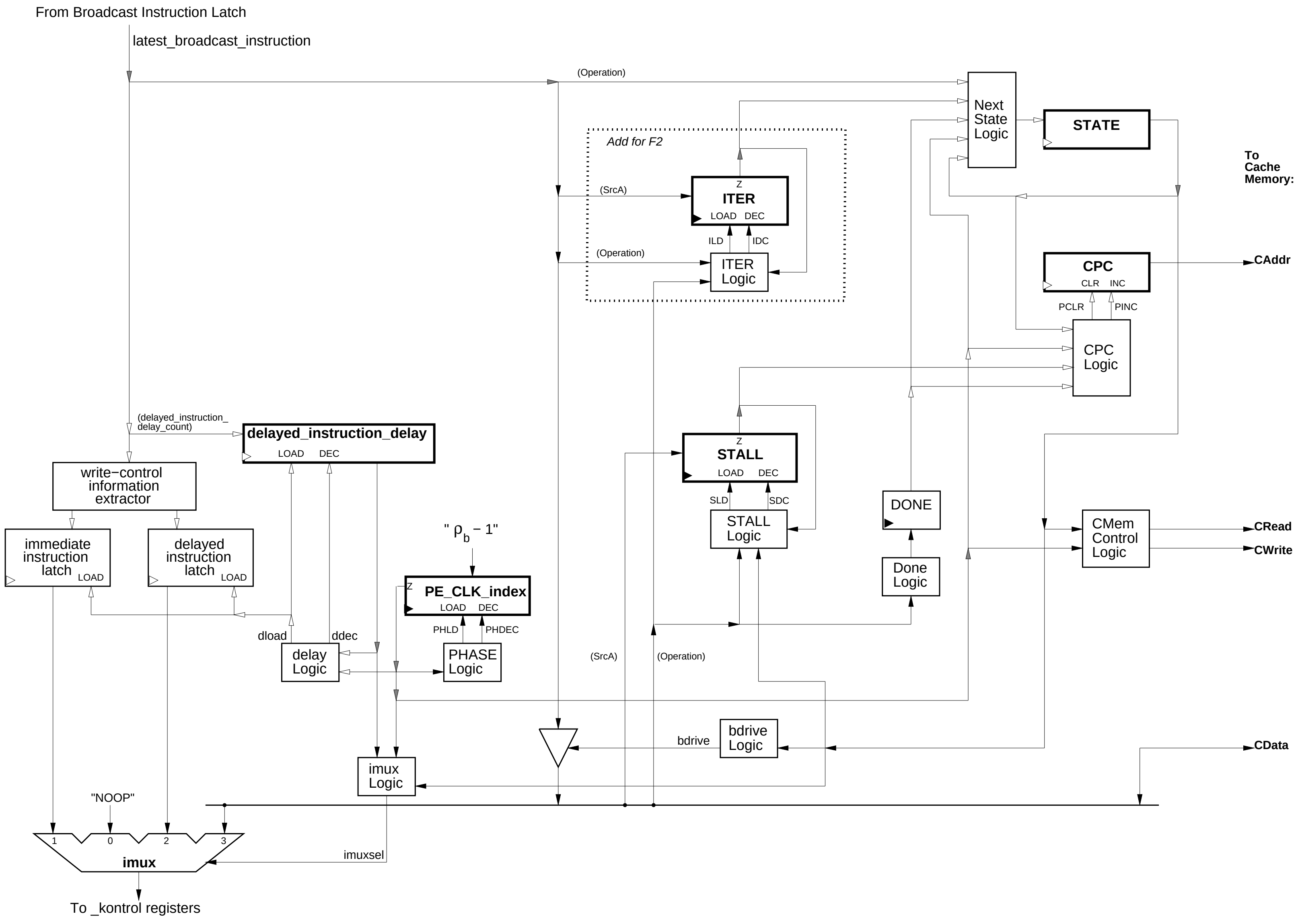


Figure A.11: F_0 and F_2 Cache Controllers

Figure A.11 shows the F_0 and F_2 cache controllers. It is interesting to note the simplicity of the cache controllers, and the very slight change needed to convert an F_0 I-cache variant into an F_2 I-cache variant.

Appendix B

Assembly Language Programming and Translation

As indicated in Figure 5.1 illustrating the method used to evaluate I-cache, the computations solving the sample problems were described using assembly language programs. The assembly language is closely related to the machine code described in Appendix A. The following important abstractions achieved in the assembly language facilitated the programming of the sample problem solutions:

1. The assembly language programming model is sequential, abstracting details of pipelined instruction execution.
2. The assembly language programming model abstracts the details of instruction timing. An assembly language program specifies a sequence of operations, without regard to the latencies of individual operations.
3. Assembly language programs define parameters which are used in expressions that provide compile-time literal values.
4. Assembly language programs define labels which are used as symbolic branch target addresses in system controller sequencer instructions.

These abstractions provide convenience in describing SIMD computations. For example, assembly language program parameters allow a single program to describe a set of operationally structured SIMD computations. Typical parameters are functions of problem size (N), the number of PEs in the system (P), or the number of PEs per PE module (K).

This appendix motivates the choice of assembly language programming, describes the language itself, explains details relating to re-programming for I-cache, and presents some of the interesting details of the language implementation.

B.1 Assembly Language v. High-level Languages

High-level languages used to describe data-parallel algorithms typically suppress such details as the target system's inter-PE communication network topology or the number of PE registers [39, 41, 84]. However, the details do affect the operational structure of a computation. For example, the inter-PE communication network topology determines the set of available inter-PE communication subsystem operations. As another example, maintaining a high-level language program variable in local external memory requires local external memory access operations that would not be needed were the variable maintained in a register. It is the properties of the operational structure of a computation that determine I-cache speedup. In describing a sequence of FU and MCS operations performed by the PE, an assembly language program makes the details of computation's operational structure explicit. This characteristic distinguishes assembly language from the high-level languages.

The assembly language is not notable for the conciseness of algorithmic expression that it engenders. The emphasis is on explicitness of operation sequences corresponding to physical activity in a SIMD computation. There are no virtual processors and no virtual memory; there are only representations of real PEs, real networks, and real memory required for detailed I-cache speedup measurements.

Of course, it is possible to compile into assembly language from a high-level language. It seemed at the outset of this work that the number of examples would be small enough, and the details important enough, to merit writing assembly language programs by hand. In fact, coding efforts often began with a representation of the PE operation sequence in a C-like pseudo-code, subsequently hand-translated into assembly language. While assembly language facilitated producing the ultimate machine code programs needed for the simulations, its small number of simple abstractions were easier to implement efficiently than the high-level programming abstractions that are themselves the subject of many a dissertation would have been.

B.2 Assembly Language Syntax

This section provides an overview of the assembly language. Appendix C shows the derivation of an example, and the assembly language program in Figure C.3 may be a helpful illustration of the syntax described here.

The first line of an assembly language program declares the program's parameter names. Each remaining line specifies a label or one statement. Each statement associates a system controller instruction with a PE instruction, although either part of a statement may be left blank. A "!" character denotes the beginning of a comment that runs to the end of the current line.

Each system controller operations associated with register-to-register PE operations. The assembly language syntax for baseline computations is simple: each line contains a statement or a label. A statement has a system controller part and/or a PE part, separated by a semicolon, as follows:

`system_controller_part ; PE_part`

B.2.1 System Controller Instruction

The system controller part of a line specifies a system controller instruction along with parameters for that instruction. The format of the system controller instruction closely follows that of the corresponding machine code instruction shown in Figure A.5, with the exception that conditions are specified mnemonically, branch targets are labels instead of absolute addresses, and iteration counts are specified as expressions contained within matched single-quote characters.

B.2.2 PE Instruction

The PE part of a statement specifies a PE instruction of the form:

`[Context] Dest = Operation(SrcA,SrcB)`

The **`Context`** field specifies a context operation, **`Dest,`** **`SrcA,`** and **`SrcB`** specify register addresses, and **`Operation`** names an FU or MCS operation. The operations are listed in Section A.4.2. Any field omitted in the PE part of a line takes its standard "null" value.

B.3 Assembly Language Re-Programming for I-Cache

I-cache requires that a small number of *cache-control* instructions be added to the set of globally broadcastable machine code instructions. Cache-control instructions direct the storing of cache blocks and their subsequent retrieval from cache.

I-cache uses two additions to the assembly language. One addition is the inclusion of cache-control instructions among the set specified in the PE part of an assembly language statement. The three cache-control instructions for F_0 and F_2 I-cache variants are ***CC_BSTO*** and ***CC_ESTO***, delimiting cache block preambles, and ***CC_FORK***, activating a previously stored cache block.

Another addition to the assembly language is a special instruction form called a "***FORK*** construct". The ***FORK*** construct allows the programmer to associate a ***CC_FORK*** operation with the cache preamble that will have been executed prior to the ***CC_FORK*** itself. This construct frees the programmer from having to determine the duration of cache block execution. The assembler/scheduler transforms each ***FORK*** construct into a ***CC_FORK*** operation in the resulting machine code program and causes the system controller to execute a wait loop for the duration of the cache block's execution.

Figure C.5 shows the program from Figure C.3 adapted to use an F_0 I-cache variant. Figure C.6 shows the program adapted to use an F_2 I-cache variant. The difference between the two programs is that the F_2 program associates an iteration count with the ***FORK*** construct, which is used in the ***CC_FORK*** instruction activating the F_2 cache block.

B.4 Scheduler

B.4.1 Basic Block Definition

To simplify its implementation, the scheduler performs code-motion optimizations only within basic blocks. In ordinary programming languages, basic blocks are defined as sequences of necessarily sequentially executed instructions, or "straight-line" code; only the first statement in a basic block can be a branch target in the program's execution, while only the last statement of the basic block can result in a branch being taken [29].

Compiler optimizations that overlap operations along multiple independent pathways are easiest to perform within basic blocks [49]. Flow graph analysis complexity increases exponentially with the number of possible outstanding branch operations one is willing to consider simultaneously. While it is certainly possible to optimize across basic blocks in some cases [29], general solutions can be prohibitively difficult.

The scheduler's definition of a basic block is augmented from the conventional definition: Not only do labels and conditional branches delimit basic block boundaries, but PE context management instructions delimit basic block boundaries as well.

B.4.2 Pipeline Optimization

A PE instruction is at risk of a pipeline hazard if either of its operands is a PE register. A pipeline hazard arises when one of an instruction's source registers is written by a dynamically preceding instruction. In pipelined execution, the preceding instruction will not have written the register by the time it is read for this instruction. A straightforward transcription of the assembly language program would thus yield incorrect computation results.

A conservative solution to a pipeline hazard is to delay the second instruction, stalling the pipeline by inserting a ***NOOP*** into the machine code program. This delay ensures that the second instruction obtains the correct register value. However, in some cases pipeline hazards can be repaired so that no cycles are wasted on pipeline stalls.

The scheduler detects pipeline hazards by examining each instruction's sources and comparing them against each potentially preceding instruction's destination. If either of this instruction's source registers is the same as any of the preceding instructions' destination registers, a pipeline hazard exists. If *all* preceding instructions' PE operations write the same register and *all* of those operations use the FU or the same MCS, then a pipeline stall is avoided by replacing this instruction's source reference with a reference to the output register of the subsystem used by those preceding instructions. Otherwise, a pipeline stall needs to be inserted so that this instruction can execute correctly.

B.4.3 Phase-Splitting

Phase-splitting is a widely used technique for overlapping PE operations along high-latency PE-external pathways with PE-internal operations. The basic idea of phase-splitting is to re-cast the high-latency operation as a pair of low-latency operations, one initiating and one terminating activity on the high-latency pathway. The initiating and terminating operations must be separated by a fixed number of instructions in the program executed by the PE. Phase-splitting makes it possible for operations on other data pathways to overlap with the high-latency operation. Where such overlap cannot occur, for example due to flow-dependencies in a program, time is spent idle waiting for the result from the high-latency operation.

A detailed discussion of this concept in the context of multiprocessor PE memory reads through high-latency networks is found in [54]: Where possible, reads are initiated well in advance of the instruction requiring the referenced value.

The introduction of I-cache and its concomitant fast PE clock into the PE chip means that some MCS operations which have single-cycle latency in generic SIMD computers (such as neighbor communications in SLAP [27] and in MPP [4]), become high-latency operations in I-cached SIMD computers. High-latency MCS operations should be overlapped where possible with each other and with FU calculation. For example, a regular neighbor communication instruction is phase-split into an initiating instruction transmitting a register value through the inter-PE communication subsystem and a terminating instruction storing into a register a corresponding value received from that subsystem. In SLAP, local external memory references always have latency greater than 1 and thus are phase-split [39].

PE FU instructions for the basis computer cannot be phase-split because they are require control information on every cycle. In machine code programs, this assumption is reflected by the presence of place holder instructions for all but the first and last instructions of a sequence; the initiating and terminating operations of a sequence carry meaningful source and destination information, while the intervening instructions carry out the steps of the FU operation.

Phase-split MCS instructions are not allowed in assembly language programs. The scheduler phase-splits high-latency MCS instructions, and it subsequently reorganizes basic blocks to overlap those high-latency instructions with other instructions where possible. An example of phase-splitting is shown in the left-hand side of Figure B.1.

When an assembly language statement's PE instruction is phase-split and the statement also specifies a system controller sequencer instruction, the sequencer instruction is associated with the terminating node of the phase-split instruction. If the statement specifies a system controller indexer instruction, the indexer instruction is associated with the initiating node of the phase-split instruction.

B.4.4 Code Reorganizer

The reorganizer attempts to overlap phase-split MCS instructions with each other and with FU instructions. The reorganizer operates on the flow graph within basic blocks, performing conservative

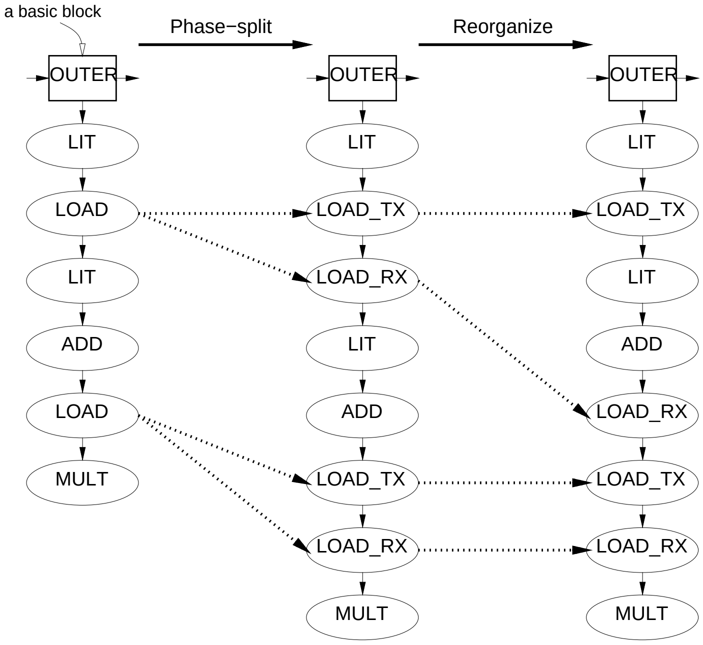


Figure B.1: Phase-Splitting and Operation Overlap

greedy local re-ordering.

An example of the result of the reorganizer result is shown on the right-hand side of Figure B.1. In the example, the ***LIT*** and ***ADD*** instructions overlap with the first ***LOAD*** instruction, while no overlap is possible for the second ***LOAD*** instruction.

Note that if the reorganizer were excluded, the scheduled code would be correct but possibly inefficient. The flow graph input to the reorganizer represents as phase-split all high-latency MCS instructions. The reorganizer's goal is to move the first phase of a phase-split operation up in the flow graph (and thus earlier in the schedule) and to move the second phase of a phase-split operation down in the flow graph (and thus later in the schedule). The instructions associated with the flow graph nodes that intervene the two nodes of the phase-split instruction overlap with that phase-split operation.

A constraint on the reordering of flow graph nodes is that results from MCS instructions must be retrieved from the PE's MCS interface registers on the cycle in which they become available. In other words, the executions of the initiating instruction and the terminating instruction of a phase-split pair must be separated by a fixed amount of time. This constraint is met by the flow graph input to the reorganizer, because no nodes intervene the two nodes of a phase-split instruction. The scheduler inserts explicit delays required when the instruction slots between the two phases of a phase-split instruction are not used for other instructions.

The initiating phase of a phase-split instruction has the same name as the original instruction with the suffix "***_TX***", while the terminating phase has the suffix "***_RX***". The reorganizer traverses a basic block until it finds a ***_TX*** instruction. When such an operation is found, the ***_TX*** phase instruction is swapped up as far as possible, and then the associated ***_RX*** phase instruction is swapped down as far as possible.

It is possible to swap the ***_TX*** phase statement (called T) with its preceding statement in the flow graph (called C) when the following conditions obtain (where T's ***_RX*** statement counterpart is called R):

1. C's PE instruction uses a different MCS from that used by T's PE instruction, or C's PE instruction uses the FU;

2. Neither source of T's PE instruction is the destination of C's PE instruction;
3. T and C do not specify system controller indexer instructions that exhibit index register flow dependencies;
4. There are enough time available between T and R to accommodate the duration of C's PE instruction;
5. If C is an *_RX* node, there is enough time available between C and its predecessor to accommodate the duration of the phase-split instruction represented by T and R.

It is possible to swap the *_RX* phase statement (R) with its succeeding statement in the flow graph (C) when the following conditions obtain (where R's *_TX* statement counterpart is T):

1. C's PE instruction uses a different MCS from that used by R's PE instruction, or C's PE instruction uses the FU;
2. Neither source of C's PE instruction is the destination of R's PE instruction;
3. R and C do not specify system controller indexer instructions that exhibit index register flow dependencies;
4. There is enough time available between T and R to accommodate the duration of C's PE instruction;
5. If C is a *_TX* node that has an *_RX* node associated with it, there is an instruction slot available between C and its successor to accommodate R.

B.4.5 Calculating I-cache Speedups

The determination of whether to use a cachable instruction sequence as a cache block rests on the balance between the extra time taken to store the cache block and the time saved by executing the instruction sequence from cache. If the former exceeds the latter, then the sequence should not be cached. If the latter exceeds the former, then the sequence might be cached, so long as so doing does not disadvantage other, more profitably cached sequences.

Cachable instruction sequences whose executions alternate during the computation compete for cache space. In single-block I-cache variants, including F_0 and F_2, caching both of a pair of such sequences necessitates re-storing them as they are needed. Therefore the determination in general of which instruction sequences to store can be arbitrarily complicated, depending as it does on the degree of conflict among candidate sequences. Such conflicts arise also for multi-block I-cache variants due to the limited capacity of cache memory.

In the basis computer, each MCS has its own local control within the PE chip. Instructions are needed only to initiate MCS operations and to terminate them by directing the storing of returned results in PE registers. The FU is different from the MCSs, in that it requires a new instruction on every clock cycle of its operation; the operation of the FU is instruction delivery rate-bound. Some actual SIMD computers diverge from this model. An example of such divergence is found in the SLAP local controller, which manages the steps of a multiplication autonomously after receiving an initiating instruction. Another example of divergence from the model is found in the CM-1 inter-PE communication subsystem, which requires broadcast control on each clock cycle of its operation [42].

The following are some general observations regarding cache speedups for the basis computer. These observations are used heuristically in the scheduling algorithm:

1. A cachable sequence of FU instructions that does not overlap with any MCS operation always executes ρ_b times faster from cache. The calculation-intensiveness of the problem, the circuit complexity of the PE relative to the widths of problem data words (in bits), and to a limited extent the number of PE registers determine the lengths of such sequences.

2. A cachable sequence of instructions using a single MCS, where no other instruction is available to overlap with those in the sequence, may or may not execute faster from cache.

   The I-cache speedup is subject to quantization and so depends on whether the durations of the instructions in the sequence happen to be multiples of ρ_b. When an instruction's latency is not a multiple of ρ_b, then in the generic SIMD computation there is some slack time between the completion of the instruction and the arrival of the next broadcast instruction. Delivering such an instruction sequence from cache allows instructions to be provided at the MCS' operation rate. The greater temporal resolution of MCS control within the PE chip gained by I-cache saves that otherwise wasted slack time.

   This time savings is significant in some instances. For example, consider a sequence of N inter-PE communication instructions on a linear array, such as might arise in the inner loop of an image motion-compensation program. Assume that the linear array communication instruction takes a single clock cycle. The sequence requires N globally broadcast instructions, and N system clock cycles, to complete in the generic SIMD computer. If shifting on a linear array could occur at twice the rate of global instruction broadcast, then the sequence can be executed from I-cache in just $\lceil \frac{N}{2} \rceil$ system clock cycles, resulting in I-cache speedup of approximately 2.

   In general, a sequence of single-cycle instructions on MCS X executes faster from I-cache by a factor of $\frac{\rho_\mathrm{b}}{\rho_\mathrm{X}}$. A sequence of S-cycle operations on MCS X executes faster from I-cache by the following factor:

$$\frac{\left\lceil \frac{S*\rho_\mathrm{X}}{\rho_\mathrm{b}} \right\rceil}{\frac{S*\rho_\mathrm{X}}{\rho_\mathrm{b}}}$$

   From this equation, it is clear that where $S * \rho_\mathrm{X}$ is a multiple of ρ_b, there is no I-cache speedup. The maximum I-cache speedup is $1 + \frac{\rho_\mathrm{b}}{S*\rho_\mathrm{X}}$.

3. A cachable sequence of overlappable instructions using disparate MCSs may or may not execute faster from cache.

   The I-cache speedup depends on the instruction latencies, as well as on the order in which they are executed. As a simplest example of this phenomenon, consider the execution of a pair of overlappable instructions, Ω and Φ, that use different MCSs. If one instruction's latency is significantly greater than the other's, then the longer instruction determines the time taken to execute the pair. A good compiler would schedule the longer operation first, so there would be no I-cache speedup. However, if Ω and Φ are of roughly similar duration, say within one broadcast instruction interval of one another, then there may be some I-cache speedup. Assume, for example, that both Ω and Φ have latency equal to one broadcast instruction interval. Then the generic SIMD computer uses exactly two broadcast instructions to execute Ω followed by Φ. When executed from cache, the instruction starting Φ can be issued as early as possible following the one starting Ω, and the time between those instructions may be less than that between globally broadcast instructions. Quantization causes a one-iteration pass through a sequence containing only these two instructions to be no faster from I-cache than from the broadcast network. However, an iterated loop containing just those two instructions may execute faster from I-cache, as might a single pass through a longer sequence containing these two instructions as a subsequence.

Appendix C

Illustrated Example of Speedup Measurement

I-cache speedup is obtained by comparing throughput of computations on SIMD computer variants. A generic SIMD computation is simulated and its output data examined to verify the correctness of the result. A computation with I-cache is then also simulated and verified. The ratios of system clock cycle counts yield the I-cache speedup. Another interesting statistic produced by the simulations is the cache size required to attain the reported speedup. Measurements are taken for varying problems, varying underlying PE chip designs, and varying system network characteristics, to yield a picture of the tradeoff stakes in portions of the large design space. This appendix presents a detailed example of how the basis computer is used to describe a computation and measure its I-cache speedup.

C.1 Operationally Structured Computation

This subsection discusses aspects of creating the assembly language program describing the generic SIMD computation that is the starting point for I-cache evaluations.

High-Level Algorithm

The derivation of a generic SIMD computation begins with a high-level algorithm solving a scalable data-parallel problem. A data-parallel problem is be defined as a set of roughly independent sub-problems such that the sub-problems can be solved concurrently. A high-level algorithm solving a data-parallel problem defines a sequence of operations to be performed by each PE that yields the answer to each sub-problem.

As an example, consider the following high-level algorithm for multiplying two square matrices of dimension P:

$$\forall \text{row=0} \ldots P-1 \left(\forall \text{col=0} \ldots P-1 \left(C_{\text{row,col}} = \sum_{\text{acol=0}}^{P-1} A_{\text{row,acol}} * B_{\text{acol,col}} \right) \right) \tag{C.1}$$

The solution of each independent sub-problem represented by each element of the result matrix C is given by an accumulated sum of products. The algorithm in Equation C.1 specifies the sequence mathematical operations required to solve the sub-problem represented by each element of the result matrix.

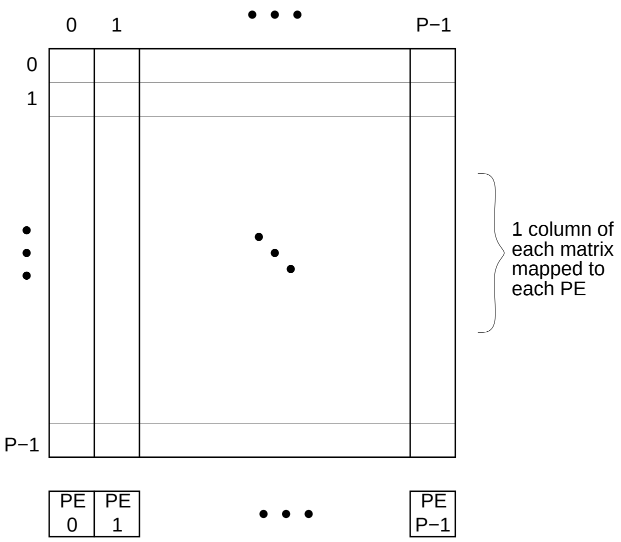


Figure C.1: Square Matrix Layout on a P-Element Linear Array

Topology-Specific Algorithm

The next step in the derivation of the generic SIMD computation transforms the high-level algorithm into a slightly less high-level description of its implementation on a computer having a specific inter-PE communication topology. This step involves identifying an inter-PE communication network topology and assigning the sub-problems to the PEs. Together, the high-level algorithm, the inter-PE communication topology, and the mapping of sub-problems to PEs define the set of mathematical operations performed within each PE as well as the inter-PE communication necessary for the PE to obtain operands.

The ideal mapping for a given high-level algorithm on a given inter-PE communication topology is not easy to ascertain in general. Automatic means for doing so is the subject of current research [84]. For the purposes of experimentation, it suffices to have reasonable mappings for the problem/topology combinations of subject computations. Experimentally adding I-cache to realistic computations does not require a general automatic solution to the mapping problem.

As an example of this mapping step, consider the square matrix multiplication on a linear array. A P-element linear array might use the mapping depicted in Figure C.1. This simple mapping assigns each PE a column of each matrix, A, B, and C, such that PE index π contains the sets of matrix elements $\{A[i,\pi] \mid i{=}0\ldots P-1\}$, $\{B[j,\pi] \mid j{=}0\ldots P-1\}$, and $\{C[i,\pi] \mid i{=}0\ldots P-1\}$.

The high-level algorithm in Equation C.1 and the mapping sketched in Figure C.1 together imply that each PE index π solves the set of sub-problems described in Equation C.2. The required inter-PE communication is evident in Equation C.2 in the references to $A_{\mathrm{row,acol}}$: for acol $\neq\pi$, PE index π needs to access $A_{\mathrm{row,acol}}$, which is mapped to PE index acol.

$$\forall \mathrm{row}{=}0\ldots P-1 \left(C_{\mathrm{row},\pi} = \sum_{\mathrm{acol}=0}^{P-1} A_{\mathrm{row,acol}} * B_{\mathrm{acol},\pi} \right) \tag{C.2}$$

The communication-explicit algorithm at this stage of the derivation is represented as a sequential program, for example that shown in Figure C.2. Explicit at this stage of the derivation are the set of variables resident in each PE, the set of operations applied within each PE to its resident variables, and the inter-PE communication required for operand access.

```
linear_array_square_matrix_multiply($A$, $B$, $C$, $P$)
  int $P$;
  element $A$[$P$][$P$], $B$[$P$][$P$], $C$[$P$][$P$];
{
  int $i$, $k$;
  PE index $\pi$;

  for ($i$ = 0 upto $P$)
  {
    forall ($\pi \in \{0$ upto $P\}$) in parallel
    {
      $C$[$i,\pi$] = $A$[$i,\pi$]$*B$[$\pi,\pi$];
      for ($k$ = 1 upto $P$)
        $C$[$i,\pi$] += $A$[$i,\pi + k$ (mod $P$)] $*$ $B$[$\pi + k$ (mod $P$),$\pi$];
    }
  }
}
```

Figure C.2: Linear Array Matrix Multiply

Assembly Language Program

Given the preceding algorithm with explicit inter-PE communication, the following is a list of some of the transformations applied to yield the assembly language program describing an operationally structured generic SIMD computation:

- Conditional control structures are transformed into PE context management operations.
- Loops are transformed into subroutine calls with explicit iteration counts.
- Loop-index-dependent expressions from the high-level language description are partitioned into two sets: some loop-index-dependent expressions are evaluated by the system controller and broadcast to the PEs, while the rest are evaluated in the PEs. Although evaluation of these expressions in the PEs is redundant (since a loop-index-dependent expression has the same value in all PEs), it is sometimes more efficient to evaluate these expressions on PEs than it is on the system controller.
- Variables are assigned locations in PE registers and local external memory, and variable references are converted into calculations of addresses followed by local external memory accesses.

As an example, Figure C.3 contains an assembly language program for a generic SIMD computation for square matrix multiplication on a linear array. The parameters **`&A_pi, &B_pi, &C_pi`** are the addresses of matrix column buffers in local external memory.

C.2 Summary of Generic SIMD Computer Parameters

The following parameters are used in deriving physical descriptions of SIMD computation from operationally structured descriptions:

```
program lsmm_ass(&A_pi, &B_pi, &C_pi, P):
        LDX IR0 `0' ; R2 = LITERAL(`&A_pi - 1')
        LDX IR2 `-1' ;
        CJSR FORC INPUT `P-1' ;
        LDX IR0 `0' ; R2 = LITERAL(`&B_pi - 1')
        LDX IR2 `P-1' ;
        CJSR FORC INPUT `P-1' ;
        LDX IR0 `0' ; R1 = LITERAL(`&A_pi')
        LDX IR0 `0' ; R7 = LITERAL(`&B_pi')
        LDX IR0 `0' ; R8 = LITERAL(`&B_pi + P')
        LDX IR0 `0' ; R3 = LITERAL(`&C_pi')
        CJSR FORC OUTER `P-1' ;
        LDX IR0 `0' ; R2 = LITERAL(`&C_pi - 1')
        LDX IR2 `2 * P - 1' ;
        CJSR FORC OUTPUT `P-1' ;
        HALT ;
INPUT:
        SPX IR2 `1' ; R1 = IO_LD(`0')
        ; R2 = ADD(`1',R2)
        LTST ICT0 INPUT ; STORE(R1, R2)
OUTER:
        ; R4 = LOAD(R1)
        ; R1 = ADD(`1', R1)
        ; R2 = ADD(R7, R0)
        ; R5 = LOAD(R2)
        ; R2 = ADD(`1',R2)
        ; LC_PUSH_EQ(R2,R8)
        ; R2 = PASS(R7)
        CJSR FORC INNER `P-2' ; [POP] R6 = MULT(R4, R5)
        ; STORE(R6,R3)
        LTST ICT0 OUTER ; R3 = ADD(`1', R3)
INNER:
        ; R4 = LDN0(R4)
        ; R5 = LOAD(R2)
        ; R2 = ADD(`1',R2)
        ; LC_PUSH_EQ(R2, R8)
        ; R2 = PASS(R7)
        ; [POP] R9 = MULT(R4, R5)
        LTST ICT0 INNER ; R6 = ADD(R6, R9)
OUTPUT:
        ; R2 = ADD(`1',R2)
        ; R1 = LOAD(R2)
        SPX IR2 `1' ; IO_ST(R1,`0')
        LTST ICT0 OUTPUT ;
```

Figure C.3: Assembly Language Program for Linear Array Square Matrix Multiply

1. Number of PEs in the computer $\{P\}$
2. Number of PE building blocks in the computer $\{M\}$
3. Number of PEs per PE chip $\{K\}$ (=$\frac{P}{M}$)
4. Per-PE local external memory size {LEMSize}
5. PE register file size {PERFSize}
6. The set of possible inter-PE communication operations. This set is determined by the topology of the computer's inter-PE communication network.
7. FU and MCS operation stepcounts, determined by such factors as
    - PE datapath width (in bits) relative to problem data word-width,
    - FU circuit complexity (For example, multipliers and barrel shifters require more complex circuits than do adders and distance-1 shifters.), and
    - PE chip pin time-sharing for MCSs.

C.3 Physically Structured Generic SIMD Computation

At this final stage of the derivation, the latencies of all FU and MCS operations are explicit, and high-latency operations are explicitly overlapped where possible with other operations that are flow-independent.

Figure C.4 shows a program produced from the program in Figure C.3 that provides a 1024-PE SIMD-D throughput baseline.

C.4 Derivation of Multi-Clock Computation

Multi-clock SIMD computation is I-cached SIMD computation without the I-cache itself. The local controller of a multi-clock SIMD computer contains the multi-clock generator that supplies the MCSs with clocks, each at its highest rate. The throughput on a multi-clock SIMD computer is compared with the throughput baseline to indicate how much speedup is attained from multi-clocking alone.

The assembly language program describing an operationally structured generic SIMD computation also describes an operationally structured multi-clock SIMD computation. The difference between the corresponding physically structured computations is that the ρ-set describing the relative MCS clock rates is an additional set of parameters in the translation to the physically structured multi-clock computation.

C.5 Derivation of I-Cached Computations

An operationally structured I-cached computation is obtained by changing the assembly language program for the generic SIMD computation. The changes involve re-ordering some of the instructions and adding cache-control instructions as required to store and subsequently activate routines in cache.

The physically structured I-cached computation is obtained by assembling and scheduling the program using operation stepcounts, a ρ-set, and cache size as parameters.

	System Controller Instruction				PE Module Instruction					
#	SC_Op'n	f1	f2	f3	C	Dest	Operation	SrcA	SrcB	L_B_Val
0	LDX	0	0			REG.2	LITERAL.0			-1
1	LDX	2	-1							
2	CJSR	FORC	23	1023						
3	LDX	0	0			REG.2	LITERAL.0			1023
4	LDX	2	1023							
5	CJSR	FORC	23	1023						
6	LDX	0	0			REG.1	LITERAL.0			0
7	LDX	0	0			REG.7	LITERAL.0			1024
8	LDX	0	0			REG.8	LITERAL.0			2048
9	LDX	0	0			REG.3	LITERAL.0			2048
10	CJSR	FORC	27	1023						
11	LDX	0	0			REG.2	LITERAL.0			2047
12	LDX	2	2047							
13	CJSR	FORC	48	1023						
14										
15										
16										
17										
18										
19										
20										
21										
22	HALT									
23	SPX	2	1			REG.1	IO_LD.0			0
24						REG.2	ADD.0	1	REG.2	
25							STORE.0	REG.1	FU_OUT	
26	LTST	ICT0	23							
27							LOAD_TX.0		REG.1	
28						REG.4	LOAD_RX.0			
29						REG.1	ADD.0	1	REG.1	
30						REG.2	ADD.0	REG.7	REG.0	
31							LOAD_TX.0		FU_OUT	
32						REG.5	LOAD_RX.0			
33						REG.2	ADD.0	1	REG.2	
34							LC_PUSH_EQ.0	FU_OUT	REG.8	
35						REG.2	PASS.0	REG.7		
36	CJSR	FORC	40	1022	POP	REG.6	MULT.0	REG.4	REG.5	
37										
38							STORE.0	REG.6	REG.3	
39	LTST	ICT0	27			REG.3	ADD.0	1	REG.3	
40						REG.4	LDN0.0	REG.4		
41							LOAD_TX.0		REG.2	
42						REG.5	LOAD_RX.0			
43						REG.2	ADD.0	1	REG.2	
44							LC_PUSH_EQ.0	FU_OUT	REG.8	
45						REG.2	PASS.0	REG.7		
46					POP	REG.9	MULT.0	REG.4	REG.5	
47	LTST	ICT0	40			REG.6	ADD.0	REG.6	FU_OUT	
48						REG.2	ADD.0	1	REG.2	
49							LOAD_TX.0		FU_OUT	
50						REG.1	LOAD_RX.0			
51	SPX	2	1				IO_ST.0	LEM_OUT		0
52	LTST	ICT0	48							

Figure C.4: Machine-code Program for Baseline Linear Array Square Matrix Multiply

Operationally Structured F_0 Computation

Figure C.5 shows a program for the matrix multiplication computation using an F_0 cache.

Operationally Structured F_2 Computation

Figure C.6 shows a program for the matrix multiplication computation using an F_2 cache.

C.6 Measured Throughput and Speedup

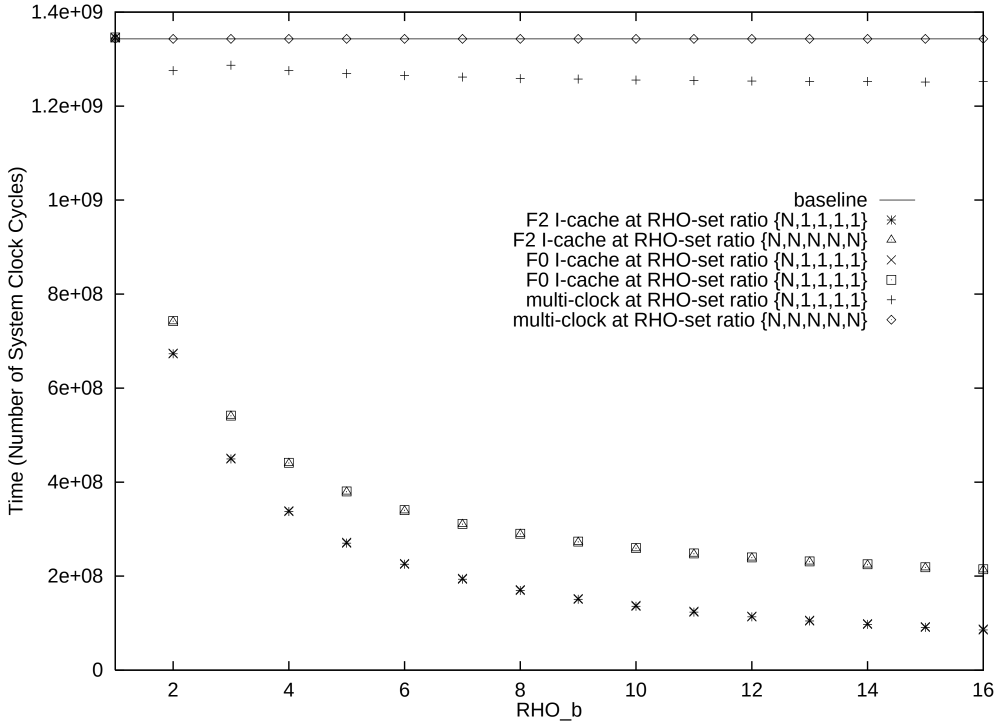
The assembly language programs are recompiled, simulated, and verified. Measurements shown in the following figures for each of the four SIMD computer variants SIMD-A, SIMD-B, SIMD-C, and SIMD-D. Measurements are taken for ρ_b ranging from 1 to 16 using the ρ-sets $\{N, N, N, N, N\}$ and $\{N, 1, 1, 1, 1\}$. The cache sizes used to obtain the the measured I-cache speedups in the four SIMD computer variants were $1196 \ldots 1204$, $307 \ldots 312$, $48 \ldots 62$, and $10 \ldots 22$, respectively.

```
program lsmm_1ce_ass(&A_pi, &B_pi, &C_pi, P):
        LDX IR0 `0' ; R2 = LITERAL(`&A_pi - 1')
        LDX IR2 `-1' ;
        CJSR FORC INPUT `P-1' ;
        LDX IR0 `0' ; R2 = LITERAL(`&B_pi - 1')
        LDX IR2 `P-1' ;
        CJSR FORC INPUT `P-1' ;
CACHE1:
        ; CC_BSTO
        ; R4 = LDN0(R4)
        ; R5 = LOAD(R2)
        ; R2 = ADD(`1',R2)
        ; LC_PUSH_EQ(R2, R8)
        ; R2 = PASS(R7)
        ; [POP] R9 = MULT(R4, R5)
        ; R6 = ADD(R6, R9)
        ; CC_ESTO
        LDX IR0 `0' ; R1 = LITERAL(`&A_pi')
        LDX IR0 `0' ; R7 = LITERAL(`&B_pi')
        LDX IR0 `0' ; R8 = LITERAL(`&B_pi + P')
        LDX IR0 `0' ; R3 = LITERAL(`&C_pi')
        CJSR FORC OUTER `P-1' ;
        LDX IR0 `0' ; R2 = LITERAL(`&C_pi - 1')
        LDX IR2 `2 * P - 1' ;
        CJSR FORC OUTPUT `P-1' ;
        HALT ;
INPUT:
        SPX IR2 `1' ; R1 = IO_LD(`0')
        ; R2 = ADD(`1',R2)
        LTST ICT0 INPUT ; STORE(R1, R2)
OUTER:
        ; R4 = LOAD(R1)
        ; R1 = ADD(`1', R1)
        ; R2 = ADD(R7, R0)
        ; R5 = LOAD(R2)
        ; R2 = ADD(`1',R2)
        ; LC_PUSH_EQ(R2,R8)
        ; R2 = PASS(R7)
        CJSR FORC INNER `P-2' ; [POP] R6 = MULT(R4, R5)
        ; STORE(R6,R3)
        LTST ICT0 OUTER ; R3 = ADD(`1', R3)
INNER:
        FORK CACHE1 ;
        LTST ICT0 INNER ;
OUTPUT:
        ; R2 = ADD(`1',R2)
        ; R1 = LOAD(R2)
        SPX IR2 `1' ; IO_ST(R1,`0')
        LTST ICT0 OUTPUT ;
```

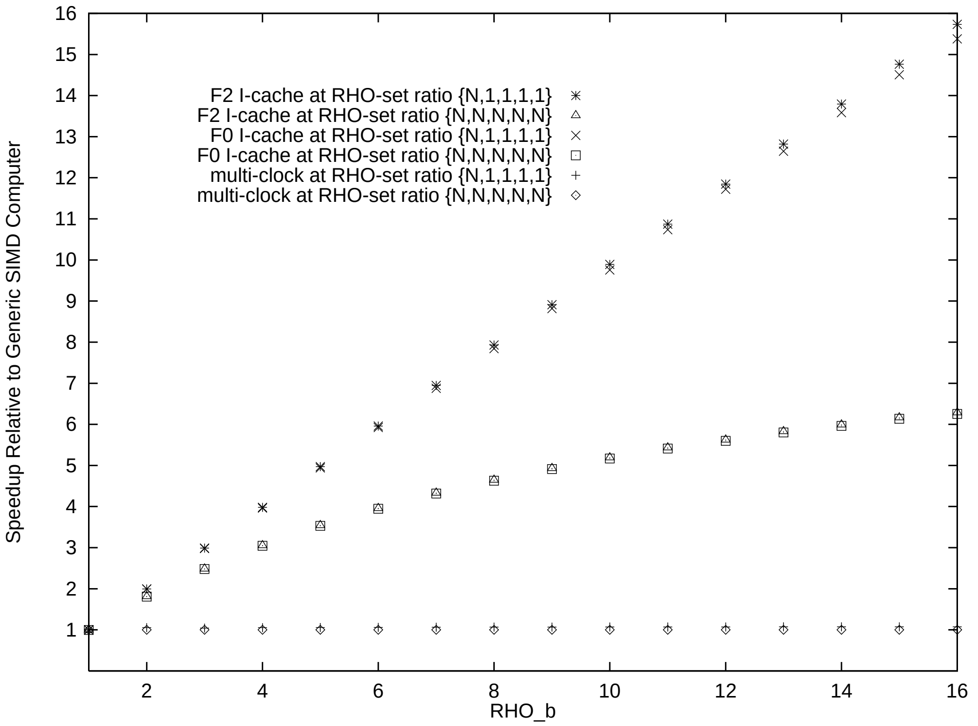
Figure C.5: assembly Language Program for F_0 Linear Array Square Matrix Multiply

```
program lsmm_2ce_ass(&A_pi, &B_pi, &C_pi, P):
        LDX IR0 `0' ; R2 = LITERAL(`&A_pi - 1')
        LDX IR2 `-1' ;
        CJSR FORC INPUT `P-1' ;
        LDX IR0 `0' ; R2 = LITERAL(`&B_pi - 1')
        LDX IR2 `P-1' ;
        CJSR FORC INPUT `P-1' ;
CACHE1:
        ; CC_BSTO
        ; R4 = LDN0(R4)
        ; R5 = LOAD(R2)
        ; R2 = ADD(`1',R2)
        ; LC_PUSH_EQ(R2, R8)
        ; R2 = PASS(R7)
        ; [POP] R9 = MULT(R4, R5)
        ; R6 = ADD(R6, R9)
        ; CC_ESTO
        LDX IR0 `0' ; R1 = LITERAL(`&A_pi')
        LDX IR0 `0' ; R7 = LITERAL(`&B_pi')
        LDX IR0 `0' ; R8 = LITERAL(`&B_pi + P')
        LDX IR0 `0' ; R3 = LITERAL(`&C_pi')
        CJSR FORC OUTER `P-1' ;
        LDX IR0 `0' ; R2 = LITERAL(`&C_pi - 1')
        LDX IR2 `2 * P - 1' ;
        CJSR FORC OUTPUT `P-1' ;
        HALT ;
INPUT:
        SPX IR2 `1' ; R1 = IO_LD(`0')
        ; R2 = ADD(`1',R2)
        LTST ICT0 INPUT ; STORE(R1, R2)
OUTER:
        ; R4 = LOAD(R1)
        ; R1 = ADD(`1', R1)
        ; R2 = ADD(R7, R0)
        ; R5 = LOAD(R2)
        ; R2 = ADD(`1',R2)
        ; LC_PUSH_EQ(R2,R8)
        ; R2 = PASS(R7)
        ; [POP] R6 = MULT(R4, R5)
        FORK CACHE1 `P-2' ;
        ; STORE(R6,R3)
        LTST ICT0 OUTER ; R3 = ADD(`1', R3)
OUTPUT:
        ; R2 = ADD(`1',R2)
        ; R1 = LOAD(R2)
        SPX IR2 `1' ; IO_ST(R1,`0')
        LTST ICT0 OUTPUT ;
```

Figure C.6: assembly Language Program for F_2 Linear Array Square Matrix Multiply

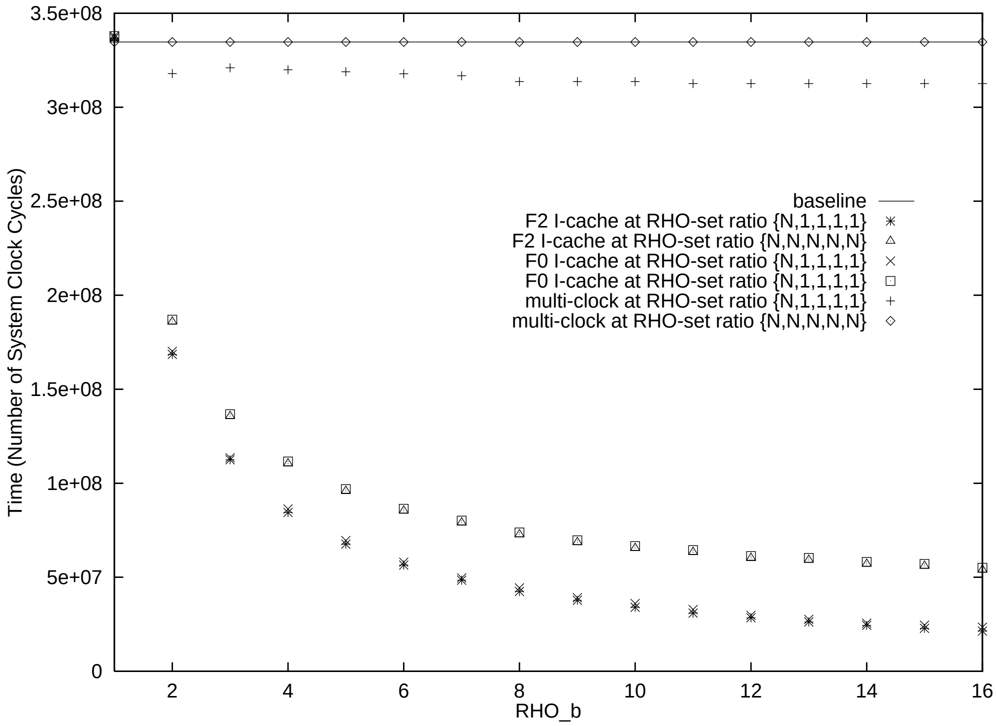


(a) Computation Time

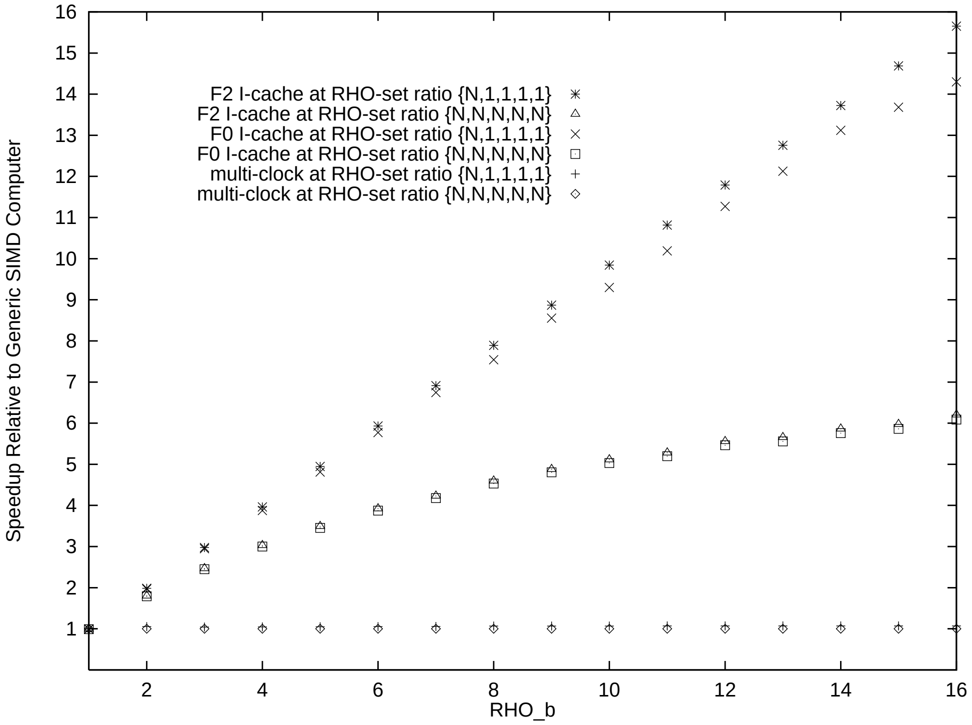


(b) Speedup

Figure C.7: Results for 1K Square Matrix Multiply on SIMD-A

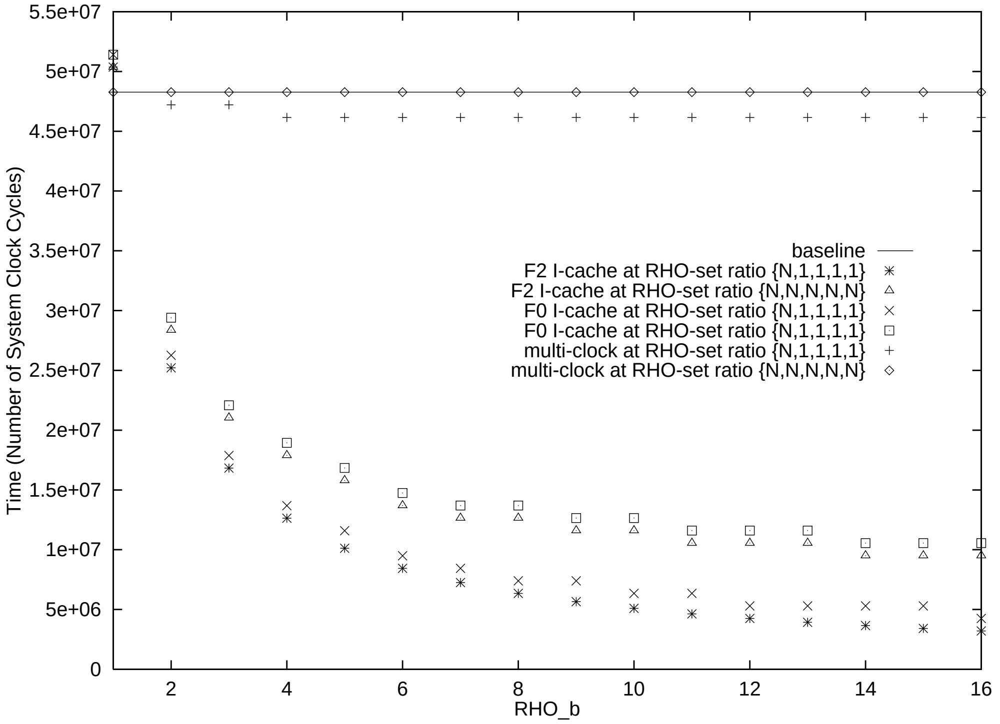


(a) Computation Time

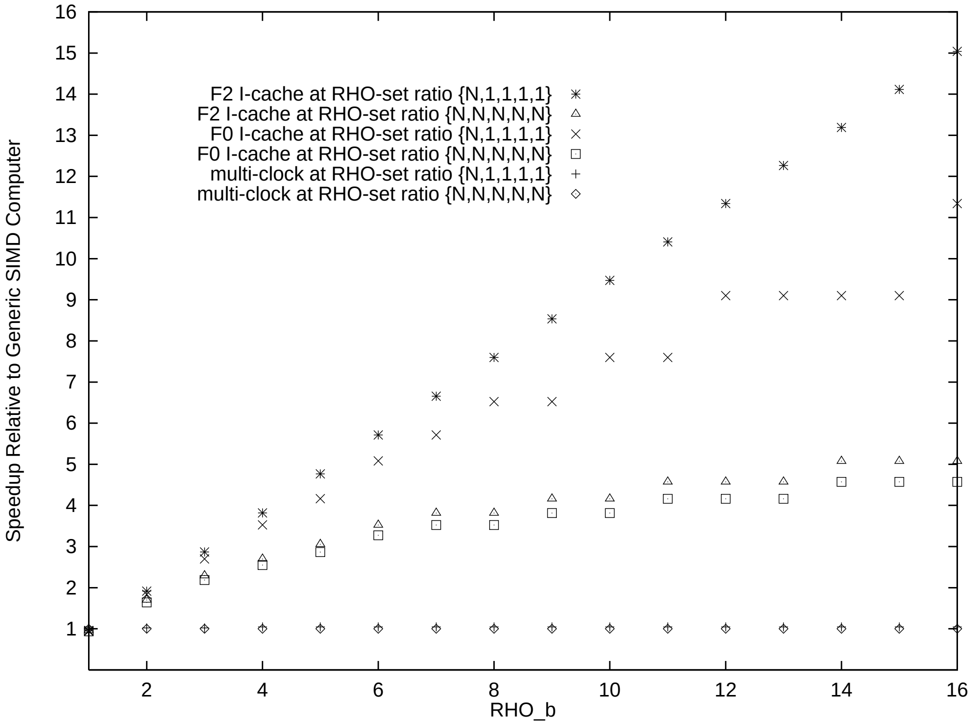


(b) Speedup

Figure C.8: Results for 1K Square Matrix Multiply on SIMD-B

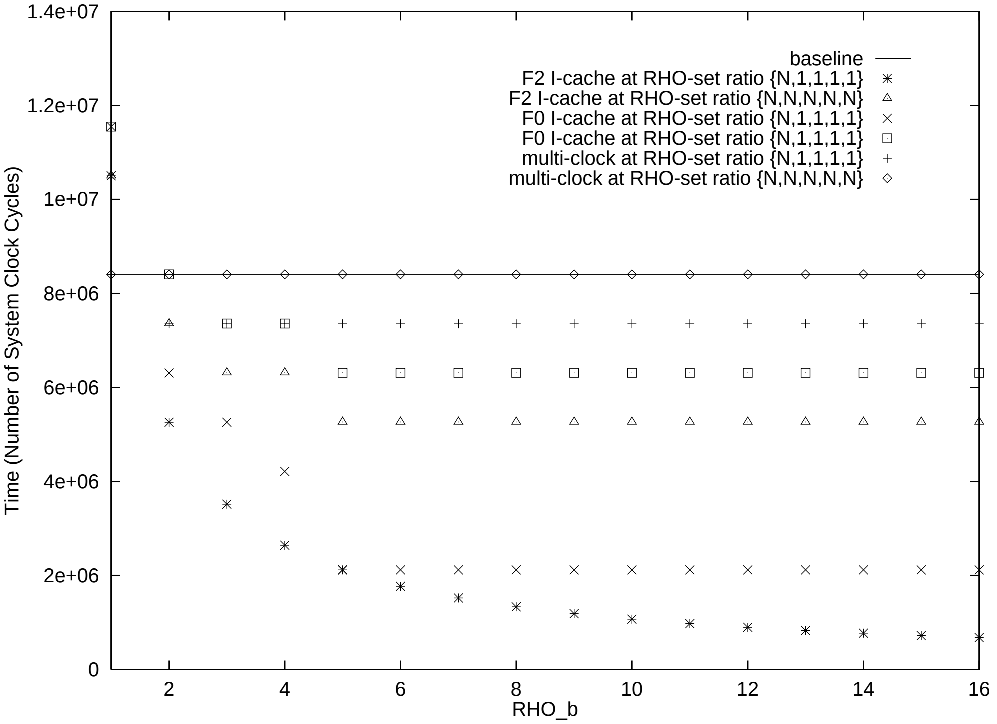


(a) Computation Time

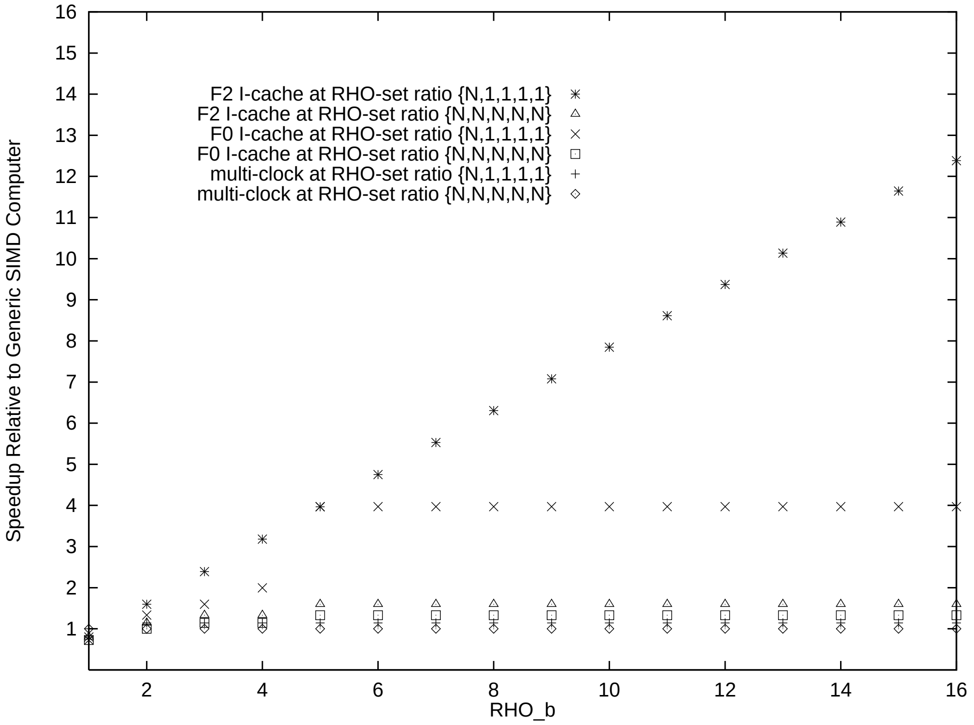


(b) Speedup

Figure C.9: Results for 1K Square Matrix Multiply on SIMD-C



(a) Computation Time



(b) Speedup

Figure C.10: Results for 1K Square Matrix Multiply on SIMD-D

Appendix D

The Sample Programs

This appendix shows the assembly language programs used in the I-cache evaluations.

```
program tree_sum_1(logP):
! tree summation. logP is log(P), where P is the number of PEs.
! loop index is in R5.
!
! Load the datum A[i] from system data memory into R1:
        LDX IR0 `0' ; R1 = IO_LD(`0')
        ; R5 = PASS(`-1')
        CJSR FORC SUMUP `logP - 1' ; R3 = PASS(R0)
! wake all the PEs up again
        ; [CLR]
! PE 0 provides the answer, mask all other PE's answers
        ; LC_PUSH_NE(`0',R0)
        ; R1 = PASS(`0')
! Store the result from R1 into system data memory;
        LDX IR0 `0' ; IO_ST(R1,`1')
        HALT ;
SUMUP:
        ; R2 = PASS(R1)
        ; R6 = AND(`1',R3)
        ; LC_PUSH_EQ(`0',R6)
        ; R4 = ADD(`1',R3)
        ; [INV] R4 = ADD(`-1',R3)
        ; [POP] R5 = ADD(`1',R5)
        ; R4 = LSHIFT(R4,R5)
        ; LC_PUSH_EQ(`0',R6)
        ; R2 = ROUTE(R2,R4)
        ; [INV] R4 = PASS(`-1')
        ; [INV] R1 = ADD(R1,R2)
        ; R6 = PASS(`1')
        LTST ICT0 SUMUP ; R3 = RSHIFT(R3,R6)
```

Figure D.1: Assembly Language Program for `tree`

```
program excl_plus_scan(logP):
! exclusive plus scan.
! The argument logP is log(P), where P is the number of PEs.
! The local buffer of "L" values starts at LEM[0], while
! The local buffer of indexes starts at LEM[logP]
! No messing around, so index register system only used if necessary
!
! Load the datum A[pi] from system data memory into R1:
        LDX IR0 `0' ; R1 = IO_LD(`0')
        ; R2 = PASS(R0)
        ; R8 = PASS(`-1')
        ; R3 = PASS(`0')
        ; R5 = PASS(`0')
        ; R4 = PASS(`1')
        CJSR FORC SWEEPUP `logP - 1' ;
        ; R1 = PASS(`0')
        ; R8 = ADD(`1',R8)
        ; [FRC] R9 = PASS(`0')
        CJSR FORC SWEEPDN `logP - 1' ;
! Store the result from R1 into system data memory;
        LDX IR0 `0' ; IO_ST(R1,`1')
        HALT ;
SWEEPUP:
        ; R10 = AND(`1',R2)
        ; R8 = ADD(`1',R8)
        ; LC_PUSH_EQ(`0',R10)
        ; R7 = ADD(`1',R2)
        ; R7 = LSHIFT(R7,R8)
        ; R7 = OR(R7,R5)
        ; [INV] R7 = PASS(`-1')
        ; R5 = LSHIFT(R5,R4)
        ; R5 = OR(`1',R5)
        ; [POP]
        ; LC_PUSH_NE(`0',R10)
        ; R6 = ROUTE(R1,R7)
        ; [INV] R7 = PASS(`-1')
        ; [INV] R1 = ADD(R1,R6)
        ; STORE(R6,R3)
        ; R3 = ADD(`1',R3)
        LTST ICT0 SWEEPUP ; R2 = RSHIFT(R2,R4)
SWEEPDN:
        ; R5 = RSHIFT(R5,R4)
        ; R7 = LSHIFT(R2,R8)
        ; R8 = ADD(`-1',R8)
        ; R7 = OR(R7,R5)
        ; R2 = LSHIFT(R2,R4)
        ; R2 = OR(R2,R4)
        ; R9 = PASS(`1')
        ; [POP]
        ; LC_PUSH_EQ(`0',R9)
        ; R1 = ROUTE(R1,R7)
        ; [INV] R3 = ADD(`-1',R3)
        ; R6 = LOAD(R3)
        ; R1 = ADD(R6,R1)
        LTST ICT0 SWEEPDN ; [POP]
```

Figure D.2: Assembly Language Program for **scan**

```
program bubble_sort(P):
! Sorting P values on a P-element linear array
! Perform local swaps until no body detects any out of order values.
!
        LDX IR0 ‘0’ ; R1 = IO_LD(‘0’)
        ; R3 = PASS(‘0’)
        ; LC_PUSH_EQ(‘0’,R0)
        ; R3 = PASS(‘1’)
        LDX IR0 ‘0’ ; [POP] R7 = LITERAL(‘P-1’)
        ; LC_PUSH_EQ(R7,R0)
        ; R3 = PASS(‘1’)
        ; [POP] R5 = AND(‘1’,R0)
        ; LC_PUSH_EQ(‘0’,R5)
        ; R4 = PASS(‘1’)
        ; [INV] R4 = PASS(‘0’)
! jump to the sorting step inner loop --
! 2019 is the measured number of STEP iterations for a 4K-element
! problem-instance; providing this number in the CJSR lets the compiler
! figure out how long the computation runs.
        CJSR FORC STEP ‘2018’ ; [POP]
        LDX IR0 ‘0’ ; IO_ST(R1,‘0’)
        HALT ;
STEP:
        ; R6 = PASS(‘0’)
! even phase:
        ; LC_PUSH_EQ(‘1’,R4)
        ; R2 = LDN0(R1)
        ; [INV] R2 = LUP0(R1)
        ; [POP] LC_PUSH_EQ(‘1’,R4)
        ; LC_PUSH_LT(R2,R1)
        ; R1 = PASS(R2)
        ; R6 = PASS(‘1’)
        ; [POP]
        ; [INV] LC_PUSH_LT(R1,R2)
        ; R1 = PASS(R2)
        ; [POP]
! odd phase:
        ; [POP] LC_PUSH_EQ(‘0’,R4)
        ; R2 = LDN0(R1)
        ; [INV] R2 = LUP0(R1)
        ; [POP] LC_PUSH_EQ(‘0’,R3)
        ; LC_PUSH_EQ(‘1’,R4)
        ; LC_PUSH_LT(R1,R2)
        ; R1 = PASS(R2)
        ; R6 = PASS(‘1’)
        ; [POP]
        ; [INV] LC_PUSH_LT(R2,R1)
        ; R1 = PASS(R2)
        ; [POP]
        ; [POP]
        ; [POP]
        LTST RSP0 STEP ; RESPOND(R6)
```

Figure D.3: Assembly Language Program for **bubble**

```
program rc_sort(sqrtP,logP):
! Sorting P values on a P-element square mesh using Leighton's
! alternating row and column bubble sort.
!
! Input the array to be sorted:
        LDX IR0 `0' ; R1 = IO_LD(`0')
! Set up the sorting loops
        ; R4 = PASS(`0')
        LDX IR0 `0' ; R8 = LITERAL(`sqrtP')
        ; R9 = MOD(R0,R8)
        ; LC_PUSH_LT(R0,R8)
        ; R4 = PASS(`-1')
        LDX IR0 `0' ; [POP] R10 = LITERAL(`sqrtP * (sqrtP - 1)')
        ; LC_PUSH_GE(R0,R10)
        ; R4 = PASS(`-1')
        ; [POP] R3 = PASS(`0')
        ; LC_PUSH_EQ(`0',R9)
        ; R3 = PASS(`-1')
        ; [POP] R10 = ADD(`-1',R8)
        ; LC_PUSH_EQ(R9,R10)
        ; R3 = PASS(`-1')
        ; [POP] R11 = DIV(R0,R8)
        ; R11 = AND(`1',R11)
        ; R14 = AND(`1',R9)
        ; LC_PUSH_EQ(`0',R11)
        ; R5 = PASS(`-1')
        ; LC_PUSH_EQ(`0',R14)
        ; R6 = PASS(`-1')
        ; [INV] R6 = PASS(`0')
        ; [POP]
        ; [INV] R5 = PASS(`0')
        ; LC_PUSH_EQ(`0',R14)
        ; R6 = PASS(`0')
        ; [INV] R6 = PASS(`-1')
        ; [POP]
! Do the sort:
        CJSR FORC SORT_STEP `logP - 1' ; [POP]
! Set up the reversal for odd-indexed rows (works for sqrtP > 2):
        ; LC_PUSH_EQ(`0',R5)
        ; R12 = ADD(`-1',R8)
        ; R12 = SUB(R12,R9)
        ; R13 = ADD(`1',R9)
        ; LC_PUSH_GE(R13,R8)
        ; R13 = SUB(R13,R8)
        ; [POP] R2 = SDN0(R1)
        ; LC_PUSH_EQ(R13,R12)
        CJSR FORC REVERSE_STEP `sqrtP / 2 - 2'; R1 = PASS(R2)
        ; [POP]
        ; [POP]
```

Figure D.4: (continued next page)

```
! Output the sorted array:
        LDX IR0 `0' ; IO_ST(R1,`0')
        HALT ;
SORT_STEP:
! Simulation measures 217 iterations for 4K data set
! But gets counted logP=12 times ~= 18 iters (18.1)
        CJSR FORC SORT_ROW `17';
! Simulation measures 58 iterations for 4K data set
! But gets counted logP=12 times ~= 5 iters (4.9)
        CJSR FORC SORT_COL `4';
        LTST ICT0 SORT_STEP ;
SORT_ROW:
        ; R7 = PASS(`0')
! even phase:
        ; LC_PUSH_EQ(R5,R6)
        ; R2 = SDN0(R1)
        ; [INV] R2 = SUP0(R1)
        ; [POP] LC_PUSH_NE(`0',R6)
        ; LC_PUSH_LT(R2,R1)
        ; R1 = PASS(R2)
        ; R7 = PASS(`-1')
        ; [POP]
        ; [INV] LC_PUSH_GT(R2,R1)
        ; R1 = PASS(R2)
        ; [POP]
! odd phase:
        ; [POP] LC_PUSH_NE(R5,R6)
        ; R2 = SDN0(R1)
        ; [INV] R2 = SUP0(R1)
        ; [POP] LC_PUSH_EQ(`0',R3)
        ; LC_PUSH_EQ(`0',R6)
        ; LC_PUSH_LT(R2,R1)
        ; R1 = PASS(R2)
        ; R7 = PASS(`-1')
        ; [POP]
        ; [INV] LC_PUSH_GT(R2,R1)
        ; R1 = PASS(R2)
        ; [POP]
        ; [POP]
        ; [POP]
        LTST RSP0 SORT_ROW ; RESPOND(R7)
```

Figure D.4: (continued next page)

```
SORT_COL:
        ; R7 = PASS(`0')
! even phase:
        ; LC_PUSH_NE(`0',R5)
        ; R2 = SDN1(R1)
        ; [INV] R2 = SUP1(R1)
        ; [INV] LC_PUSH_LT(R2,R1)
        ; R1 = PASS(R2)
        ; R7 = PASS(`-1')
        ; [POP]
        ; [INV] LC_PUSH_GT(R2,R1)
        ; R1 = PASS(R2)
        ; [POP]
! odd phase:
        ; [POP] LC_PUSH_EQ(`0',R5)
        ; R2 = SDN1(R1)
        ; [INV] R2 = SUP1(R1)
        ; [POP] LC_PUSH_EQ(`0',R4)
        ; LC_PUSH_EQ(`0',R5)
        ; LC_PUSH_LT(R2,R1)
        ; R1 = PASS(R2)
        ; R7 = PASS(`-1')
        ; [POP]
        ; [INV] LC_PUSH_GT(R2,R1)
        ; R1 = PASS(R2)
        ; [POP]
        ; [POP]
        ; [POP]
        LTST RSP0 SORT_COL ; RESPOND(R7)
REVERSE_STEP:
        ; [POP] R2 = SDN0(R2)
        ; R13 = ADD(`2',R13)
        ; LC_PUSH_GE(R13,R8)
        ; R13 = SUB(R13,R8)
        ; [POP] R2 = SDN0(R2)
        ; LC_PUSH_EQ(R13,R12)
        LTST ICT0 REVERSE_STEP ; R1 = PASS(R2)
```

Figure D.4: Assembly Language Program for **`rowcol`**

```
program bitonic_sort(logP):
! Bitonic sort on P elements.
!
! Load the datum A[i] from system data memory bank address 0 into R6:
        LDX IR0 `0' ; R6 = IO_LD(`0')
        LDX IR0 `0' ; R11 = LITERAL(`logP')
        LDX IR0 `0' ; R12 = LITERAL(`1')
        ; R1 = LSHIFT(`1',R11)
        ; R2 = PASS(R1)
        LDX IR1 `-1' ;
        CJSR FORC OUTER `logP - 1' ;
! Store the result from R6 back into system data memory bank address 0;
        LDX IR0 `0' ; IO_ST(R6,`0')
        HALT ;
OUTER:
        ; R2 = RSHIFT(R2,R12)
        ; R4 = PASS(`0')
        ; R5 = PASS(R2)
        ; R3 = PASS(R1)
        SPX IR1 `1' ;
        ;
! When the iteration count specified in the CJSR instruction is less
! than 0, the system controller indexer subsystem provides the loop
! iteration count:
        CJSR FORC INNER `-1' ;
        LTST ICT0 OUTER ;
```

Figure D.5: (continued next page)

```
INNER:
        ; R3 = RSHIFT(R3,R12)
! evaluate first condition
        ; R13 = SUB(R1,R5)
        ; LC_PUSH_LT(R0,R13)
        ; R15 = PASS(`1')
        ; [INV] R15 = PASS(`0')
        ; [POP] R14 = AND(R0,R2)
        ; LC_PUSH_EQ(R14,R4)
        ; R15 = AND(`1',R15)
        ; [INV] R15 = PASS(`0')
! first 'then' leg
        ; [POP] LC_PUSH_EQ(`1',R15)
        ; R9 = PASS(`1')
        ; R10 = PASS(`0')
        ; R8 = ADD(R0,R5)
! evaluate second condition
        ; [INV] R13 = SUB(R0,R5)
        ; LC_PUSH_LE(`0',R13)
        ; R15 = PASS(`1')
        ; [INV] R15 = PASS(`0')
        ; [POP] R14 = AND(R13,R2)
        ; LC_PUSH_EQ(R14,R4)
        ; R15 = AND(`1',R15)
        ; [INV] R15 = PASS(`0')
! second 'then' leg
        ; [POP] LC_PUSH_EQ(`1',R15)
        ; R9 = PASS(`0')
        ; R10 = PASS(`1')
        ; R8 = SUB(R0,R5)
! last leg of conditional
        ; [INV] R9 = PASS(`0')
        ; R10 = PASS(`0')
        ; R8 = PASS(`-1')
        ; [POP]
! pass, compare, conditionally swap
        ; [POP] R7 = ROUTE(R6,R8)
        ; LC_PUSH_LT(R7,R6)
        ; LC_PUSH_EQ(`1',R9)
        ; R6 = PASS(R7)
        ; [POP]
        ; [POP] LC_PUSH_GT(R7,R6)
        ; LC_PUSH_EQ(`1',R10)
        ; R6 = PASS(R7)
        ; [POP]
        ; [POP] R5 = SUB(R3,R2)
        LTST ICT0 INNER ; R4 = PASS(R2)
```

Figure D.5: Assembly Language Program for **bitonic**

```
program lsmm_ass(&A_pi, &B_pi, &C_pi, P):
        LDX IR0 `0' ; R2 = LITERAL(`&A_pi - 1')
        LDX IR2 `-1' ;
        CJSR FORC INPUT `P-1' ;
        LDX IR0 `0' ; R2 = LITERAL(`&B_pi - 1')
        LDX IR2 `P-1' ;
        CJSR FORC INPUT `P-1' ;
        LDX IR0 `0' ; R1 = LITERAL(`&A_pi')
        LDX IR0 `0' ; R7 = LITERAL(`&B_pi')
        LDX IR0 `0' ; R8 = LITERAL(`&B_pi + P')
        LDX IR0 `0' ; R3 = LITERAL(`&C_pi')
        CJSR FORC OUTER `P-1' ;
        LDX IR0 `0' ; R2 = LITERAL(`&C_pi - 1')
        LDX IR2 `2 * P - 1' ;
        CJSR FORC OUTPUT `P-1' ;
        HALT ;
INPUT:
        SPX IR2 `1' ; R1 = IO_LD(`0')
        ; R2 = ADD(`1',R2)
        LTST ICT0 INPUT ; STORE(R1, R2)
OUTER:
        ; R4 = LOAD(R1)
        ; R1 = ADD(`1', R1)
        ; R2 = ADD(R7, R0)
        ; R5 = LOAD(R2)
        ; R2 = ADD(`1',R2)
        ; LC_PUSH_EQ(R2,R8)
        ; R2 = PASS(R7)
        CJSR FORC INNER `P-2' ; [POP] R6 = MULT(R4, R5)
        ; STORE(R6,R3)
        LTST ICT0 OUTER ; R3 = ADD(`1', R3)
INNER:
        ; R4 = LDN0(R4)
        ; R5 = LOAD(R2)
        ; R2 = ADD(`1',R2)
        ; LC_PUSH_EQ(R2, R8)
        ; R2 = PASS(R7)
        ; [POP] R9 = MULT(R4, R5)
        LTST ICT0 INNER ; R6 = ADD(R6, R9)
OUTPUT:
        ; R2 = ADD(`1',R2)
        ; R1 = LOAD(R2)
        SPX IR2 `1' ; IO_ST(R1,`0')
        LTST ICT0 OUTPUT ;
```

Figure D.6: Assembly Language Program for **matmul**

```
program mesh_sobel(sqrtP):
! Load the input pixel:
        LDX IR0 ‘0’ ; R1 = IO_LD(‘0’)
! Flag the edge pixels:
        LDX IR0 ‘0’ ; R8 = LITERAL(‘sqrtP’)
        ; R7 = MOD(R0,R8)
        LDX IR0 ‘0’ ; R2 = LITERAL(‘sqrtP * (sqrtP - 1)’)
        ; R3 = ADD(‘-1’,R8)
        ; LC_PUSH_LT(R0,R8)
        ; R9 = PASS(‘-1’)
        ; [INV] R9 = PASS(‘0’)
        ; [POP] LC_PUSH_GE(R0,R2)
        ; R12 = PASS(‘-1’)
        ; [INV] R12 = PASS(‘0’)
        ; [POP] LC_PUSH_EQ(‘0’,R7)
        ; R10 = PASS(‘-1’)
        ; [INV] R10 = PASS(‘0’)
        ; [POP] LC_PUSH_EQ(R3,R7)
        ; R11 = PASS(‘-1’)
        ; [INV] R11 = PASS(‘0’)
! Calculate the x and y gradients:
        ; [POP] R2 = SDN0(R1)
        ; LC_PUSH_EQ(‘-1’,R11)
        ; R2 = PASS(R1)
        ; [POP] R3 = SUP0(R1)
        ; LC_PUSH_EQ(‘-1’,R10)
        ; R3 = PASS(R1)
        ; [POP] R13 = PASS(‘1’)
        ; R4 = LSHIFT(R1,R13)  ! pix * 2
        ; R4 = ADD(R3,R4)
        ; R4 = ADD(R4,R2)
        ; R6 = SDN1(R4)
        ; LC_PUSH_EQ(‘-1’,R12)
        ; R6 = PASS(R4)
        ; [POP] R5 = SUP1(R4)
        ; LC_PUSH_EQ(‘-1’,R9)
        ; R5 = PASS(R4)
        ; [POP] R8 = SUB(R5,R6)
        ; R4 = SUB(R2,R3)
        ; R6 = SDN1(R4)
        ; LC_PUSH_EQ(‘-1’,R12)
        ; R6 = PASS(R4)
        ; [POP] R5 = SUP1(R4)
        ; LC_PUSH_EQ(‘-1’,R9)
        ; R5 = PASS(R4)
        ; [POP] R7 = LSHIFT(R4,R13) ! R4 * 2
        ; R7 = ADD(R5,R7)
        ; R7 = ADD(R7,R6)
```

Figure D.7: (continued next page)

```
! Iteratively calculate the root square gradient value:
        ; R7 = MULT(R7,R7)
        ; R8 = MULT(R8,R8)
        ; R2 = ADD(R7,R8)
        ; R3 = RSHIFT(R2,R13)
        ; R4 = MULT(R3,R3)
        CJSR FORC SQRT `12' ; R4 = SUB(R4,R2)
! Write the gradient magnitude result
        LDX IR0 `0' ; IO_ST(R3,`1')
        HALT ;
SQRT:
        ; LC_PUSH_EQ(`0',R4)
        ; R9 = PASS(`0')
        ; [INV] R5 = LSHIFT(R3,R13)
        ; R5 = DIV(R4,R5)
        ; LC_PUSH_EQ(`0',R5)
        ; R10 = PASS(`-1')
        ; LC_PUSH_LT(`0',R4)
        ; R5 = PASS(`1')
        ; [INV] R5 = PASS(`-1')
        ; [POP]
        ; [INV] R10 = PASS(`0')
        ; [POP] R6 = SUB(R3,R5)
        ; R7 = MULT(R6,R6)
        ; R7 = SUB(R7,R2)
        ; LC_PUSH_EQ(`-1',R10)
        ; LC_PUSH_GT(`0',R4)
        ; R4 = SUB(`0',R4)
        ; [POP] LC_PUSH_GT(`0',R7)
        ; R7 = SUB(`0',R7)
        ; [POP]
        ; LC_PUSH_LT(R7,R4)
        ; R3 = PASS(R6)
        ; [POP] R9 = PASS(`0')
        ; R4 = PASS(`0')
        ; [INV] R3 = PASS(R6)
        ; R4 = PASS(R7)
        ; R9 = PASS(`-1')
        ; [POP]
        ; [POP]
        LTST RSP0 SQRT ; RESPOND(R9)
```

Figure D.7: Assembly Language Program for **sobel**

```
program median(&buf,P):
! &buf is address in local external memory of 9-element buffer that
! is used to hold each of the latest 3 sorted local rows.
! P is the number of PEs and the number of pixels per scanline and
! the number of scanlines.
!
! IR1 holds the input line number, IR2 holds the output line number.
!
        LDX IR1 `-2' ;
        SPX IR1 `2' ; R10 = IO_LD(`0')
        LDX IR2 `-1' ;
        LDX IR0 `0' ; R14 = LITERAL(`512')
        LDX IR0 `0' ; R15 = LITERAL(`P - 1')
        LDX IR0 `0' ; R5 = LITERAL(`&buf')
        LDX IR0 `0' ; R3 = LITERAL(`&buf + 3')
        LDX IR0 `0' ; R1 = LITERAL(`&buf + 6')
        ; R6 = PASS(R5)
        ; R4 = PASS(R3)
        ; R2 = PASS(R1)
! Grab the first line, sort the local 3 pixels
        ; R7 = PASS(R10)
        SPX IR1 `2' ; R10 = IO_LD(`0')
        ; R8 = LUP0(R7)
        ; LC_PUSH_EQ(`0',R0)
        ; R8 = PASS(R7)
        ; [POP] R9 = LDN0(R7)
        ; LC_PUSH_EQ(R15,R0)
        ; R9 = PASS(R7)
        CJSR FORC SORT_3 `0' ; [POP]
! Store the locally sorted first row twice
        ; STORE(R7,R6)
        ; R6 = ADD(`1',R6)
        ; STORE(R8,R6)
        ; R6 = ADD(`1',R6)
        ; STORE(R9,R6)
        ; R6 = PASS(R5)
        ; STORE(R7,R4)
        ; R4 = ADD(`1',R4)
        ; STORE(R8,R4)
        ; R4 = ADD(`1',R4)
        ; STORE(R9,R4)
        ; R4 = PASS(R3)
! Grab the second row, sort the local 3 pixels
        ; R7 = PASS(R10)
        SPX IR1 `2' ; R10 = IO_LD(`0')
        ; R8 = LUP0(R7)
        ; LC_PUSH_EQ(`0',R0)
        ; R8 = PASS(R7)
        ; [POP] R9 = LDN0(R7)
        ; LC_PUSH_EQ(R15,R0)
        ; R9 = PASS(R7)
        CJSR FORC SORT_3 `0' ; [POP]
```

Figure D.8: (continued next page)

```
! Store the locally sorted row to memory, set up local median calc:
        ; STORE(R7,R2)
        ; R2 = ADD(`1',R2)
        ; STORE(R8,R2)
        ; R2 = ADD(`1',R2)
        ; STORE(R9,R2)
        ; R2 = ADD(`-1',R2)
        ; R8 = LOAD(R4)
        ; R4 = ADD(`1',R4)
        ; R9 = LOAD(R6)
! Skip over the four least of the 9 pixels in the 3 sorted buffers:
        CJSR FORC SKIP_LEAST `3' ; R6 = ADD(`1',R6)
! Pick the least remaining element, put it into R11
        CJSR FORC PICK_LEAST `0' ;
! rotate the buffer pointer base addresses, re-initialize buffer pointers
        ; R12 = PASS(R5)
        ; R5 = PASS(R3)
        ; R3 = PASS(R1)
        ; R1 = PASS(R12)
        ; R6 = PASS(R5)
        ; R4 = PASS(R3)
! Iterate the steady-state loop P-3 times
        CJSR FORC MF_LOOP `P-4' ; R2 = PASS(R1)
        ; R7 = PASS(R10)
        SPX IR2 `2' ; IO_ST(R11,`0')
        ; R8 = LUP0(R7)
        ; LC_PUSH_EQ(`0',R0)
        ; R8 = PASS(R7)
        ; [POP] R9 = LDN0(R7)
        ; LC_PUSH_EQ(R15,R0)
        ; R9 = PASS(R7)
        CJSR FORC SORT_3 `0' ; [POP]
! Store the locally sorted row to memory, set up local median calc:
        ; STORE(R7,R2)
        ; R2 = ADD(`1',R2)
        ; STORE(R8,R2)
        ; R2 = ADD(`1',R2)
        ; STORE(R9,R2)
        ; R2 = ADD(`-1',R2)
        ; R8 = LOAD(R4)
        ; R4 = ADD(`1',R4)
        ; R9 = LOAD(R6)
! Skip over the four least of the 9 pixels in the 3 sorted buffers:
        CJSR FORC SKIP_LEAST `3' ; R6 = ADD(`1',R6)
! Pick the least remaining element, put it into R11
        CJSR FORC PICK_LEAST `0' ;
```

Figure D.8: (continued next page)

```
! rotate the buffer pointer base addresses, re-initialize buffer pointers
        SPX IR2 `2' ; IO_ST(R11,`0')
        ; R12 = PASS(R5)
        ; R5 = PASS(R3)
        ; R3 = PASS(R1)
        ; R1 = PASS(R12)
        ; R6 = PASS(R5)
! Copy 1back into curr and set up median calculation for final row
        ; R4 = ADD(`2',R3)
        ; R9 = LOAD(R4)
        ; R4 = ADD(`-1',R4)
        ; R2 = ADD(`2',R1)
        ; STORE(R9,R2)
        ; R2 = ADD(`-1',R2) ! curr points to second element of row
        ; R8 = LOAD(R4)
        ; R4 = ADD(`-1',R4)
        ; STORE(R8,R2)
        ; R7 = LOAD(R4)
        ; R4 = ADD(`1',R4) ! 1back points to second element of row
        ; R8 = PASS(R7)
        ; R9 = LOAD(R6)
        CJSR FORC SKIP_LEAST `3' ; R6 = ADD(`1',R6) ! 2back points to 2nd elt
! Pick the least remaining element, put it into R11
        CJSR FORC PICK_LEAST `0' ;
        SPX IR2 `2' ; IO_ST(R11,`0')
        HALT ;
!
SORT_3:
        ; LC_PUSH_LT(R8,R7)
        ; R12 = PASS(R7)
        ; R7 = PASS(R8)
        ; R8 = PASS(R12)
        ; [POP] LC_PUSH_LT(R9,R7)
        ; R12 = PASS(R7)
        ; R7 = PASS(R9)
        ; R9 = PASS(R12)
        ; [POP] LC_PUSH_LT(R9,R8)
        ; R12 = PASS(R8)
        ; R8 = PASS(R9)
        ; R9 = PASS(R12)
        LTST ICT0 SORT_3 ; [POP]
```

Figure D.8: (continued next page)

```
SKIP_LEAST:
        ; LC_PUSH_LT(R7,R8)
        ; LC_PUSH_LT(R7,R9)
        ; R13 = ADD(`3',R1)
        ; LC_PUSH_EQ(R2,R13)
        ; R7 = PASS(R14)  ! make pix0 bigger than any pixel
        ; [INV] R7 = LOAD(R2)
        ; R2 = ADD(`1',R2)
        ; [POP]
        ; [INV] R13 = ADD(`3',R5)
        ; LC_PUSH_EQ(R6,R13)
        ; R9 = PASS(R14)  ! make pix2 bigger than any pixel
        ; [INV] R9 = LOAD(R6)
        ; R6 = ADD(`1',R6)
        ; [POP]
        ; [POP]
        ; [INV] LC_PUSH_LT(R8,R9)
        ; R13 = ADD(`3',R3)
        ; LC_PUSH_EQ(R4,R13)
        ; R8 = PASS(R14)  ! make pix1 bigger than any pixel
        ; [INV] R8 = LOAD(R4)
        ; R4 = ADD(`1',R4)
        ; [POP]
        ; [INV] R13 = ADD(`3',R5)
        ; LC_PUSH_EQ(R6,R13)
        ; R9 = PASS(R14)  ! make pix2 bigger than any pixel
        ; [INV] R9 = LOAD(R6)
        ; R6 = ADD(`1',R6)
        ; [POP]
        ; [POP]
        LTST ICT0 SKIP_LEAST ; [POP]
PICK_LEAST:
        ; LC_PUSH_LT(R7,R8)
        ; LC_PUSH_LT(R7,R9)
        ; R11 = PASS(R7)
        ; [INV] R11 = PASS(R9)
        ; [POP]
        ; [INV] LC_PUSH_LT(R8,R9)
        ; R11 = PASS(R8)
        ; [INV] R11 = PASS(R9)
        ; [POP]
        LTST ICT0 PICK_LEAST ; [POP]
MF_LOOP:
! Internalize next row of pixels
        ; R7 = PASS(R10)
        SPX IR2 `2' ; IO_ST(R11,`0')        ! on SLAP, these two I/O
        SPX IR1 `2' ; R10 = IO_LD(`0')        ! operations overlap fully.
        ; R8 = LUP0(R7)
        ; LC_PUSH_EQ(`0',R0)
        ; R8 = PASS(R7)
```

Figure D.8: (continued next page)

```
        ; [POP] R9 = LDN0(R7)
        ; LC_PUSH_EQ(R15,R0)
        ; R9 = PASS(R7)
! Sort the 3 pixels in the neighborhood
        CJSR FORC SORT_3 `0' ; [POP]
! Store the sorted 3 elements to memory, set up local median calculation:
        ; STORE(R7,R2)
        ; R2 = ADD(`1',R2)
        ; STORE(R8,R2)
        ; R2 = ADD(`1',R2)
        ; STORE(R9,R2)
        ; R2 = ADD(`-1',R2)
        ; R8 = LOAD(R4)
        ; R4 = ADD(`1',R4)
        ; R9 = LOAD(R6)
! Skip over the four least of the 9 pixels in the 3 sorted buffers:
        CJSR FORC SKIP_LEAST `3' ; R6 = ADD(`1',R6)
! Pick the least remaining element, put it into R11
        CJSR FORC PICK_LEAST `0' ;
! rotate the buffer pointers
        ; R12 = PASS(R5)
        ; R5 = PASS(R3)
        ; R3 = PASS(R1)
        ; R1 = PASS(R12)
        ; R6 = PASS(R5)
        ; R4 = PASS(R3)
        LTST ICT0 MF_LOOP ; R2 = PASS(R1)
```

Figure D.8: Assembly Language Program for **median**

Appendix E

Measured F_0 and F_2 Speedup Bounds

Two simplest single-port I-caches, F_0 and F_2, were designed. Each of these I-caches is capable of storing only a single cache block at a time. The difference between them is that F_2 contains an iteration counter and is able to sequence through multiple iterations of a cache block from a single activation, whereas F_0 executes only single iterations of cache blocks.

These I-cache variants were evaluated for the sample programs through detailed simulation on each of four SIMD computer variants, SIMD-A, B, C, and D. These computers differ, for example, in PE datapath widths and in numbers of PEs per PE chip. In the simulations, operation stepcount parameters were used to express the number of clock cycles required to perform each operation on problem data. The following table summarizes the characteristics of the four SIMD computer variants, including typical stepcounts for 32-bit problem data:

	SIMD-A	SIMD-B	SIMD-C	SIMD-D
PEs per chip	128	32	4	2
FU bit-width	1	4	16	32
NOR stepcount	32	8	2	1
ADD stepcount	32	8	2	1
MULT stepcount	1056	263	34	1
LC_PUSH_EQ stepcount	32	8	2	1
LOAD stepcount	128	32	4	2
LDN0 stepcount	32	8	2	1

Maximum subsystem clock rates depend on wire geometries and electrical propagation characteristics of the implementation technology. In the simulations, ρ-set parameters were used to express relative multi-chip subsystem (MCS) clock rates. The PE clock rate is ρ_X times higher than the clock rate for MCS X. Highest I-cache speedups are obtained where all MCSs other than global instruction broadcast operate at the highest possible clock rates. The limitation arising from relatively slow instruction broadcast is most severe in this case. Lowest I-cache speedups are obtained where all MCSs operate at the low rate of global instruction broadcast. Subsystem-boundedness is most severe in this case, and I-cache is least advantageous because a greatest proportion of computation time is spent waiting for MCS operations to complete, as opposed to waiting for globally broadcast instructions to arrive at the PE chips.

I-cache speedup bounds were obtained by simulating I-cached computations at the ρ-sets characterizing the limiting extremes of relative MCS clock rates. This appendix presents the complete set of measured I-cache speedups, for each sample problem on each SIMD computer variant. In each case, the speedup bounds are plotted against values of ρ_b ranging from 1 to 16.

Superimposed on each set of measured speedup bounds is a curve of the form

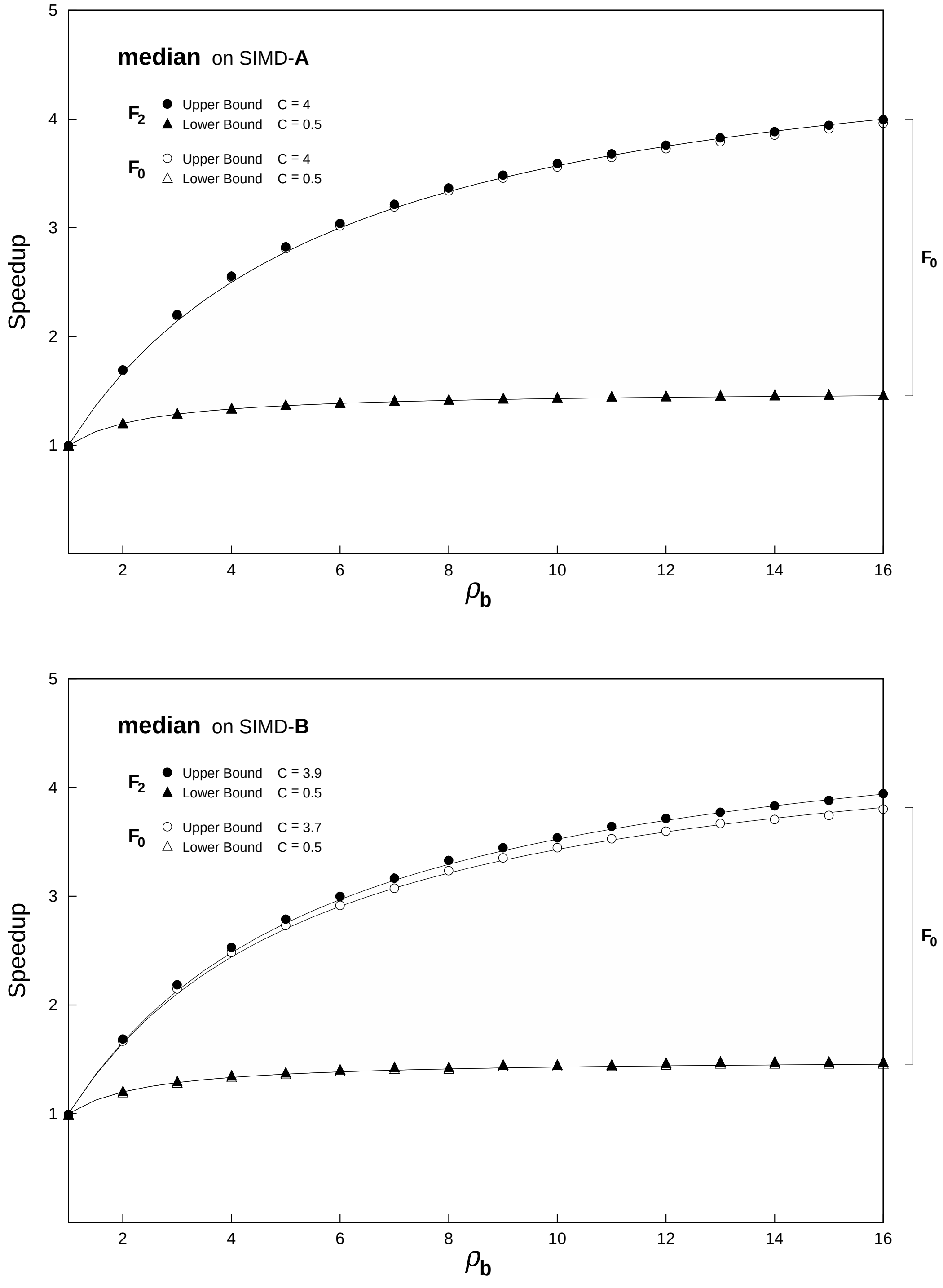
$$\frac{(1 + C)\,\rho_b}{\rho_b + C}$$

This speedup formula, derived in Section 5.10, is an approximation of the speedup of program **`simple`**, consisting of one loop. C represents the product of the fraction of instructions in **`simple`** that lie within the loop times the number of loop iterations. The closeness of fit of this approximation to the I-cache speedups for the sample programs is remarkable, in view of their varying and sometimes complex loop structures. The one significant departure from a good fit occurs for F_2 for the sample problem **`rowcol`**, due to poor cache management used in that case. For F_2 speedup bounds on **`rowcol`**, the measured values are fit to a curve of the form

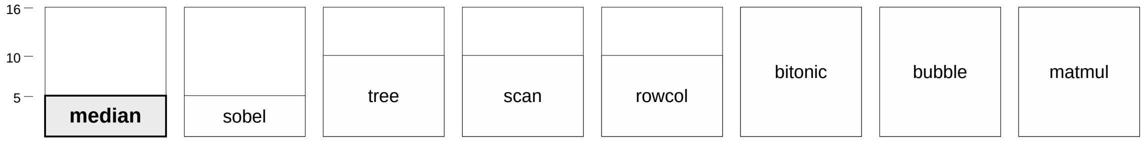
$$\frac{(1 + C)\,\rho_b}{\rho_b + C} - \frac{g\rho_b}{\rho_b + C}$$

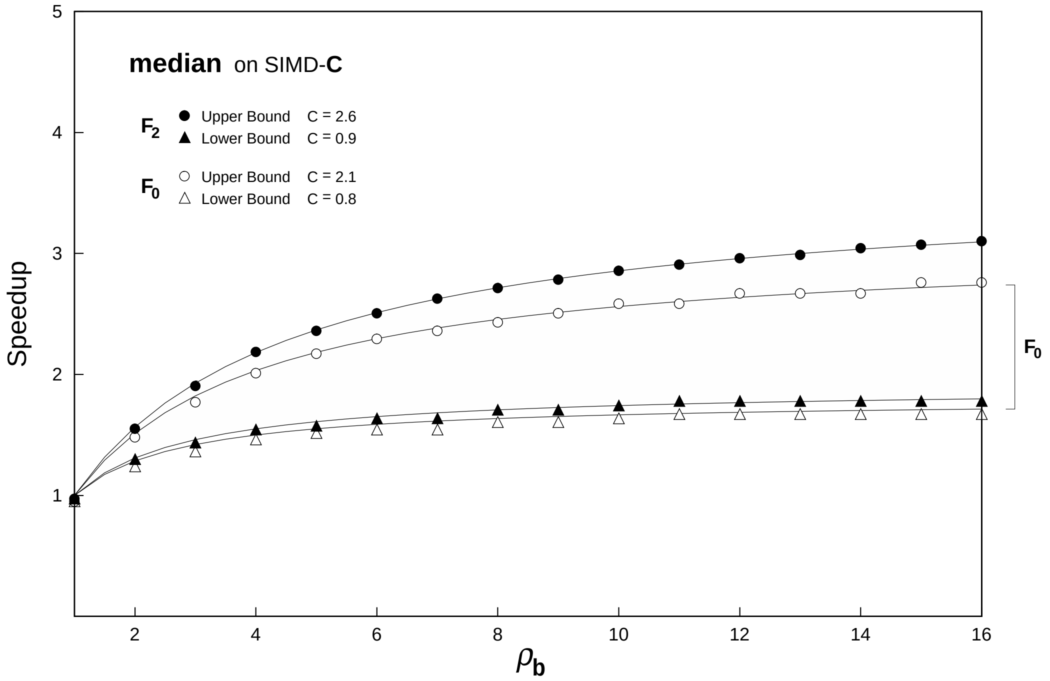
where the parameter g expresses the fraction of instructions in **`simple`** that lie within the loop. The second term in this formula represents an I-cache penalty.

The graphs on the following pages are plotted to scales indicated at the bottom of each left-hand (even-numbered) page.


median on SIMD-A
F2 ● Upper Bound C = 4
▲ Lower Bound C = 0.5
F0 ○ Upper Bound C = 4
△ Lower Bound C = 0.5
Speedup
1
2
3
4
5
2
4
6
8
10
12
14
16
ρb
F0
median on SIMD-B
F2 ● Upper Bound C = 3.9
▲ Lower Bound C = 0.5
F0 ○ Upper Bound C = 3.7
△ Lower Bound C = 0.5
Speedup
1
2
3
4
5
2
4
6
8
10
12
14
16
ρb
F0

(E.1A – E.1B)


16
10
5
median
sobel
tree
scan
rowcol
bitonic
bubble
matmul



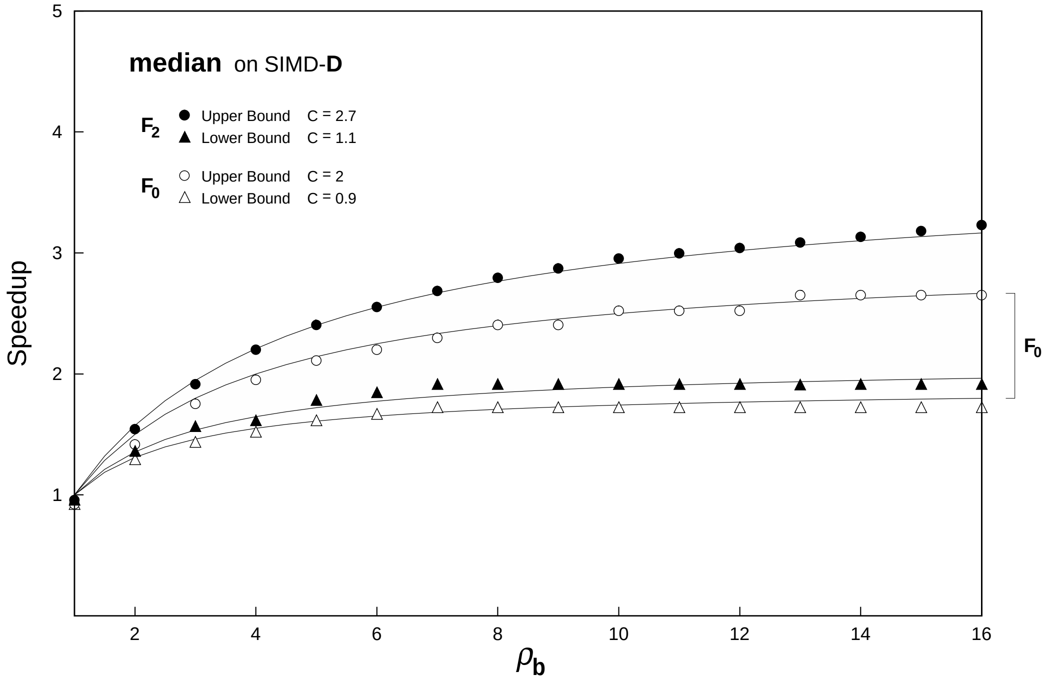
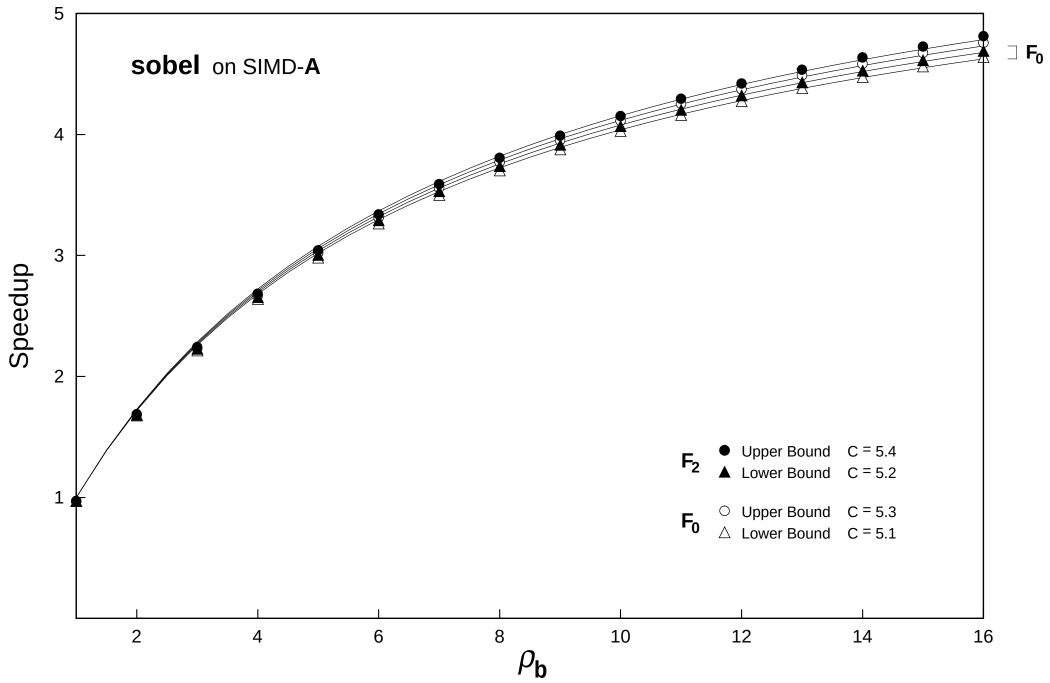
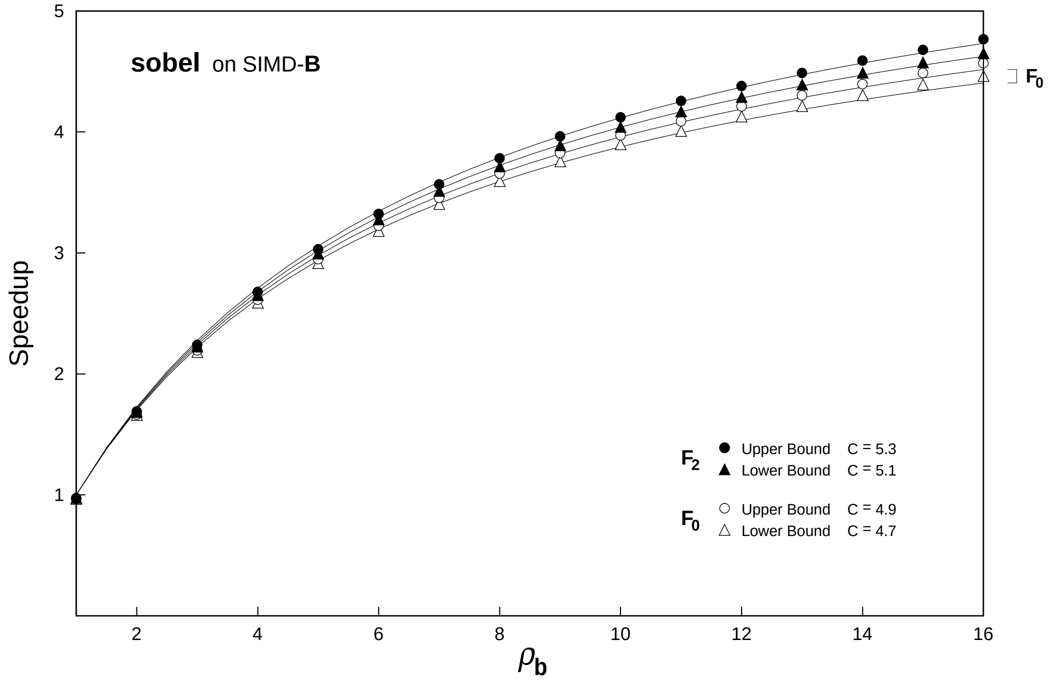


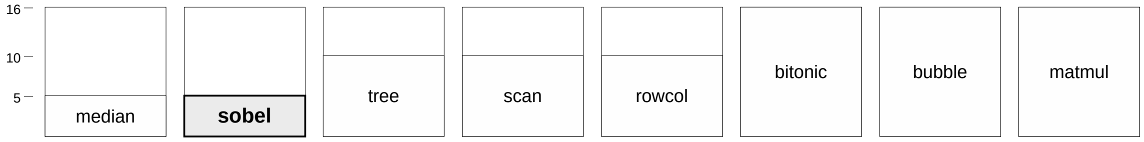
Figure E.1: **F$_0$**- and **F$_2$**-Speedup Bounds for the program **median** on SIMD-**A, B, C,** & **D.**

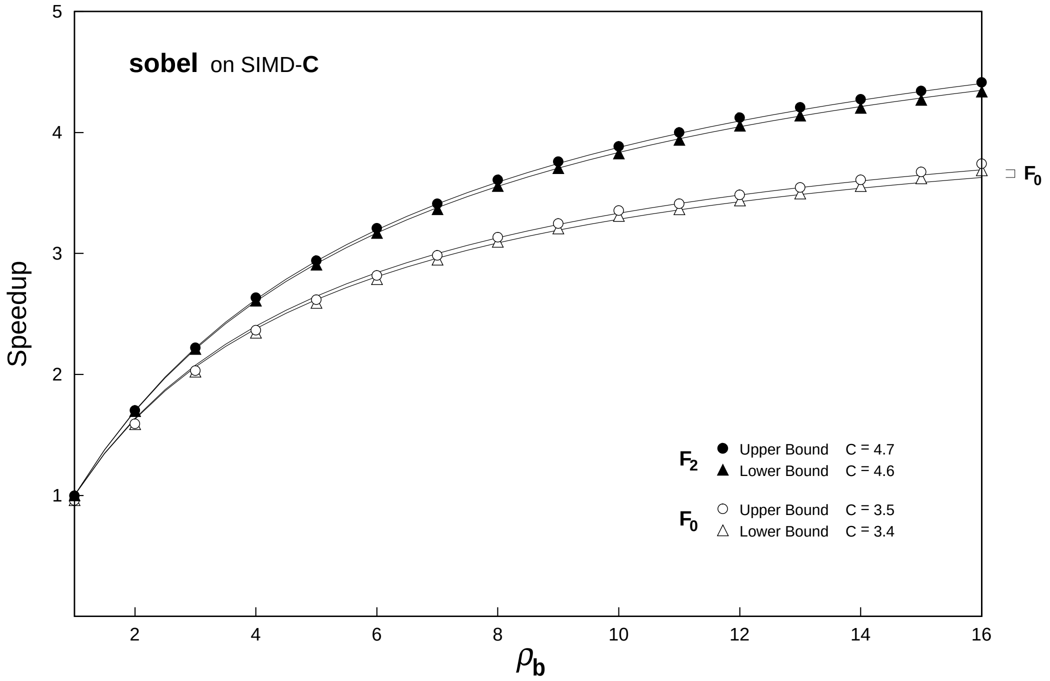
The I-cache speedups are low because of the program's complex loop structure. To get good speedup for this program, an I-cache variant needs to be able to store multiple cache blocks at once. **F$_0$** and **F$_2$** are capable of storing only a single cache block at a time, so the cache blocks tend to thrash these simple I-cache variants.


sobel on SIMD-A
Speedup
5
4
3
2
1
ρ_b
2
4
6
8
10
12
14
16
F0
F2 ● Upper Bound C = 5.4
▲ Lower Bound C = 5.2
F0 ○ Upper Bound C = 5.3
△ Lower Bound C = 5.1


sobel on SIMD-B
Speedup
5
4
3
2
1
ρ_b
2
4
6
8
10
12
14
16
F0
F2 ● Upper Bound C = 5.3
▲ Lower Bound C = 5.1
F0 ○ Upper Bound C = 4.9
△ Lower Bound C = 4.7

(E.2A – E.2B)


16
10
5
median
sobel
tree
scan
rowcol
bitonic
bubble
matmul



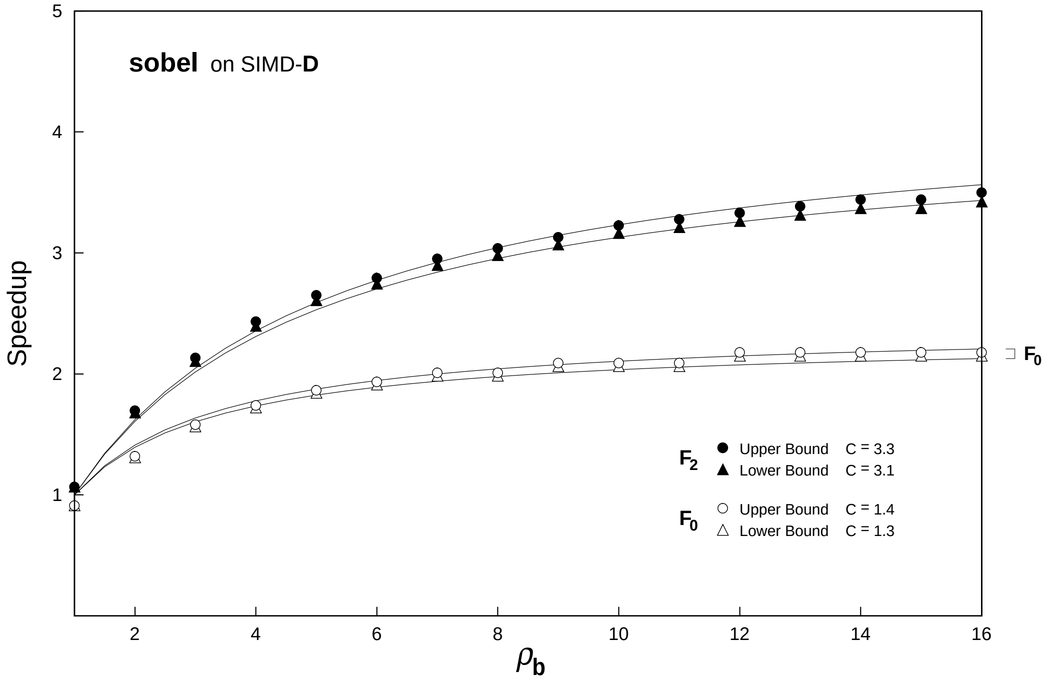
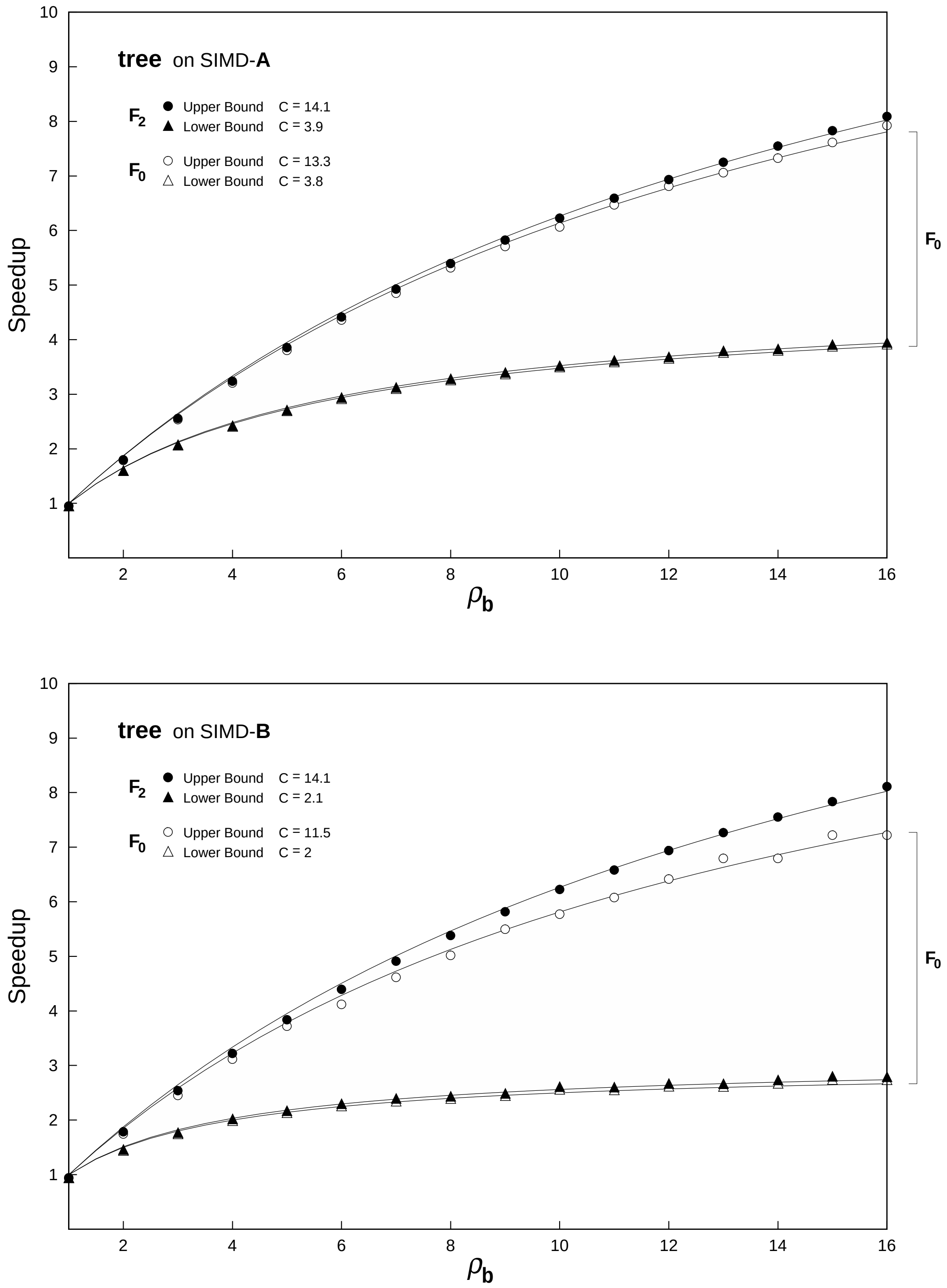
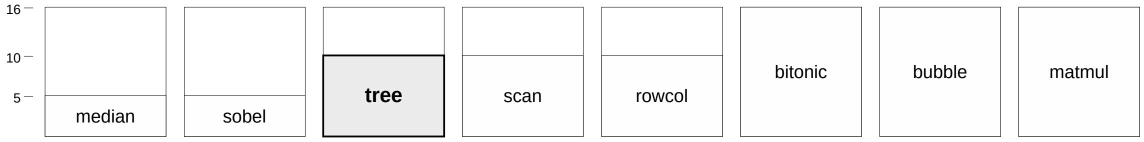


Figure E.2: F_0- and F_2-Speedup Bounds for the program **sobel** on SIMD-**A, B, C,** & **D.**

The loop body of this program is a square-root approximation-refinement step whose iteration is globally data-dependent. The small differences between upper and lower speedup bounds confirm that the repeated instruction sequence is extremely calculation-intensive. The modest I-cache speedups for this program are due primarily to a small total iteration count of the loop body.


tree on SIMD-A
F2 ● Upper Bound C = 14.1
▲ Lower Bound C = 3.9
F0 ○ Upper Bound C = 13.3
△ Lower Bound C = 3.8
Speedup
ρb
F0
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16
tree on SIMD-B
F2 ● Upper Bound C = 14.1
▲ Lower Bound C = 2.1
F0 ○ Upper Bound C = 11.5
△ Lower Bound C = 2
Speedup
ρb
F0

(E.3A – E.3B)


16
10
5
median
sobel
tree
scan
rowcol
bitonic
bubble
matmul

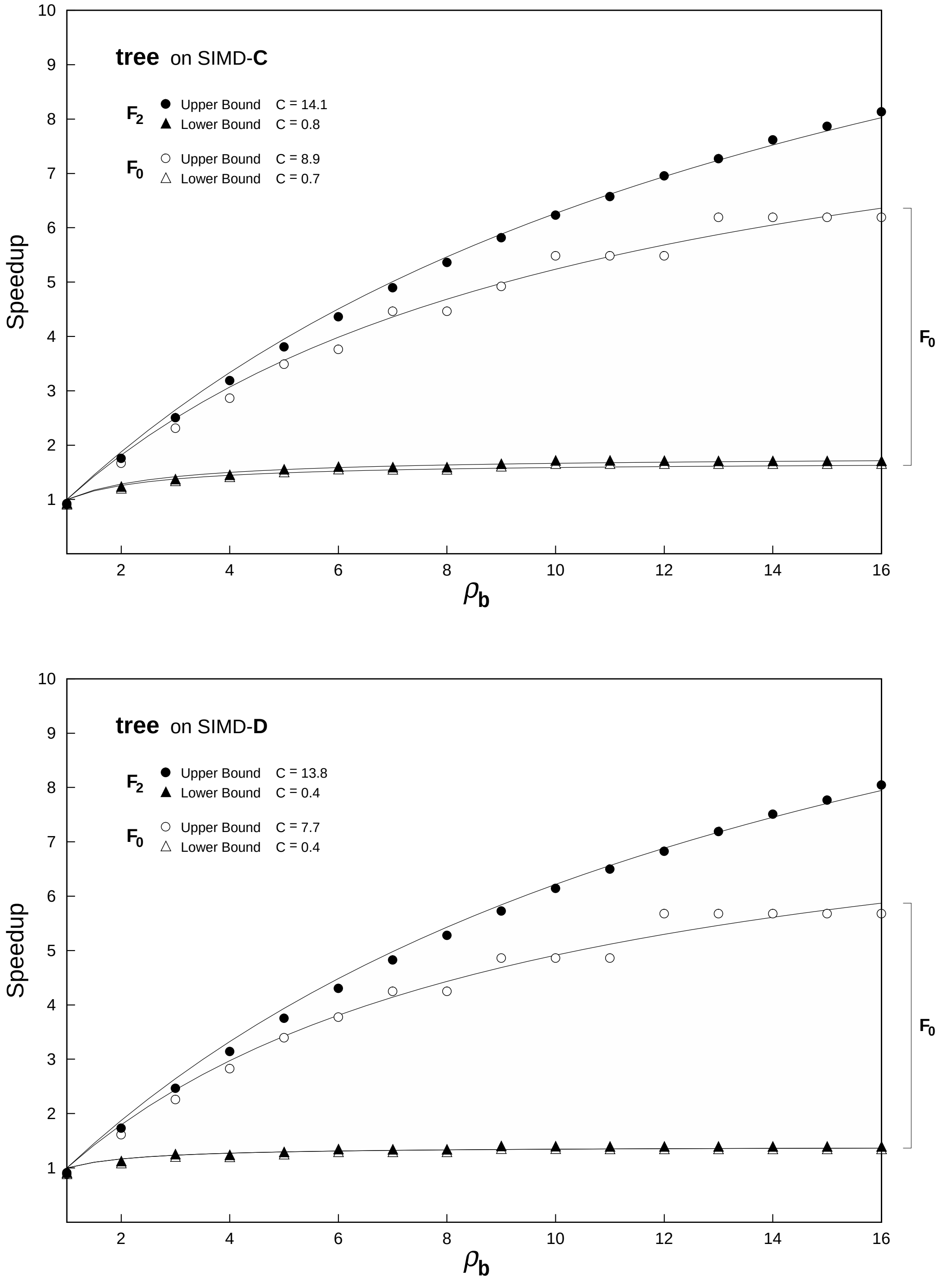
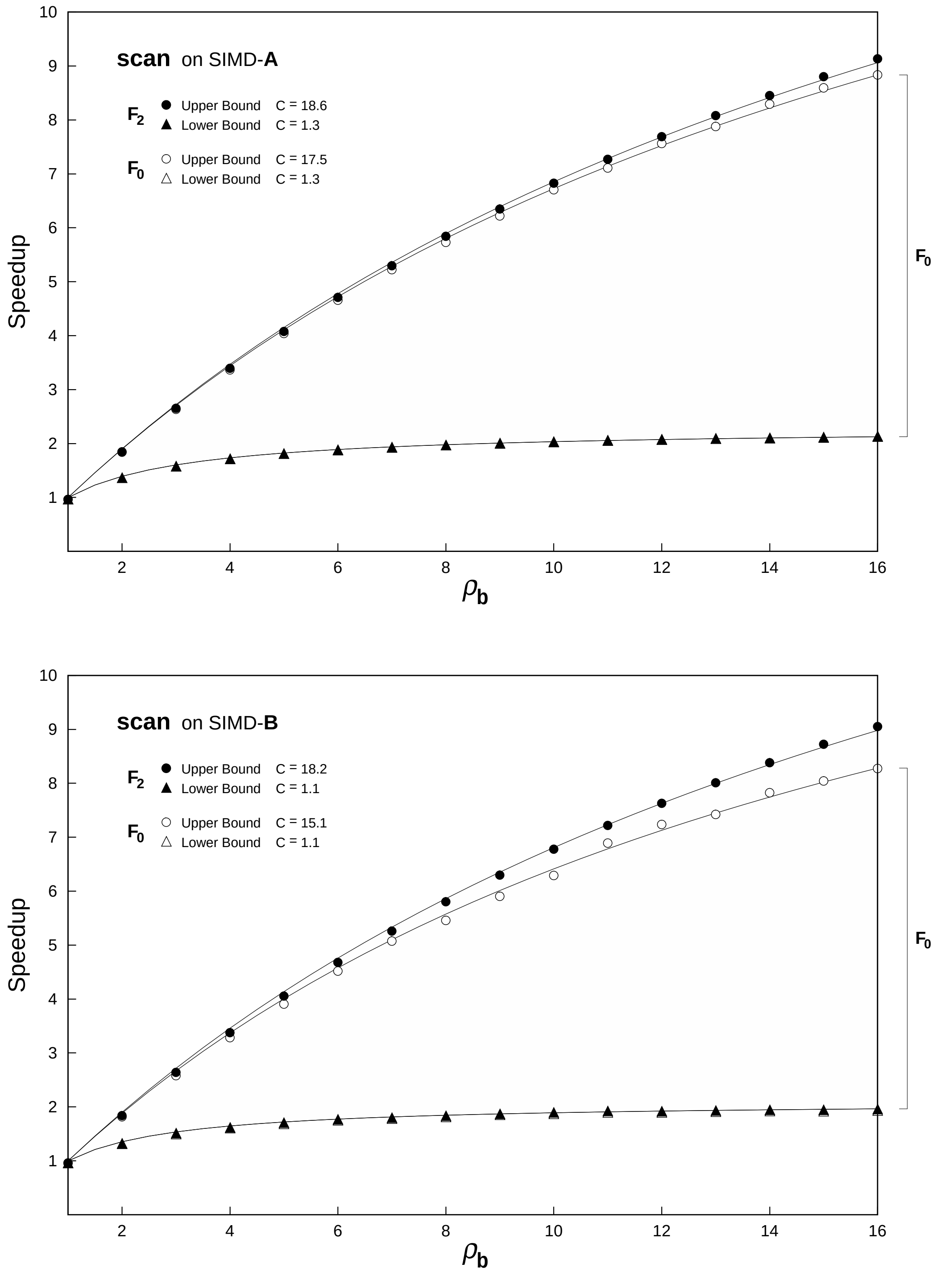
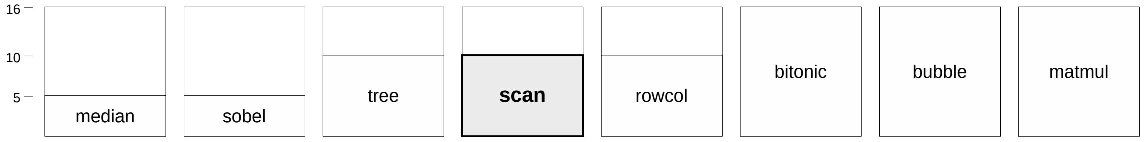


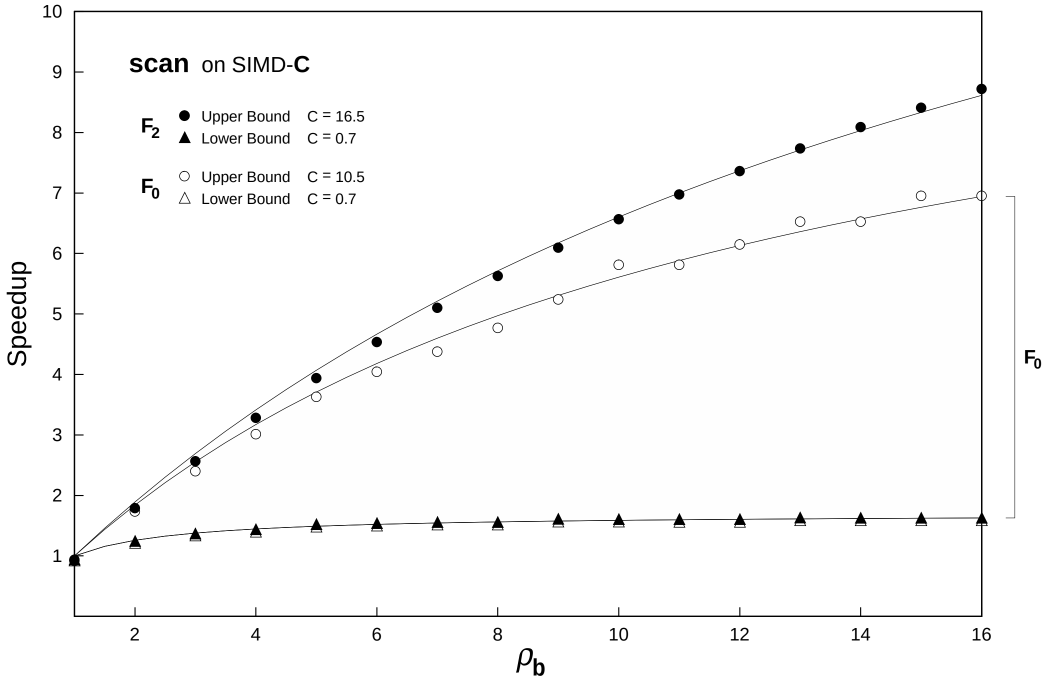
Figure E.3: $\mathbf{F_0}$- and $\mathbf{F_2}$-Speedup Bounds for the program **tree** on SIMD-**A, B, C,** & **D.**

This program consists of a short prolog followed by single iterated loop. The speedup upper bounds are lower for $\mathbf{F_0}$ than for $\mathbf{F_2}$ because of quantization.


scan on SIMD-A
F2 Upper Bound C = 18.6
F2 Lower Bound C = 1.3
F0 Upper Bound C = 17.5
F0 Lower Bound C = 1.3
F0
Speedup
ρb
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16
scan on SIMD-B
F2 Upper Bound C = 18.2
F2 Lower Bound C = 1.1
F0 Upper Bound C = 15.1
F0 Lower Bound C = 1.1
F0
Speedup
ρb
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16

(E.4A – E.4B)


16
10
5
median
sobel
tree
scan
rowcol
bitonic
bubble
matmul



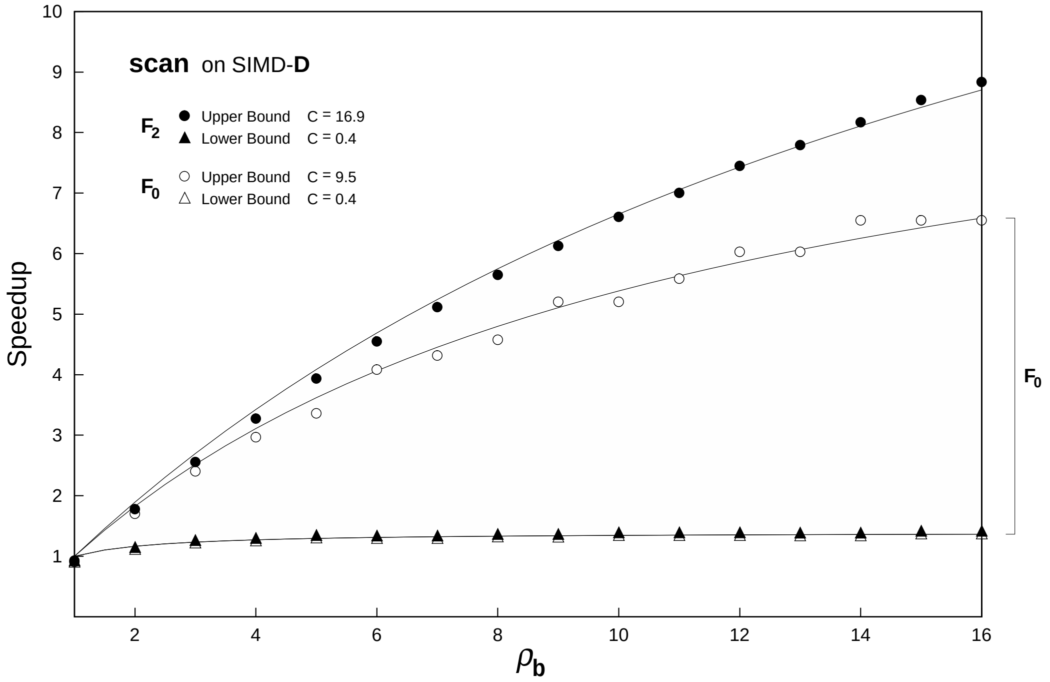
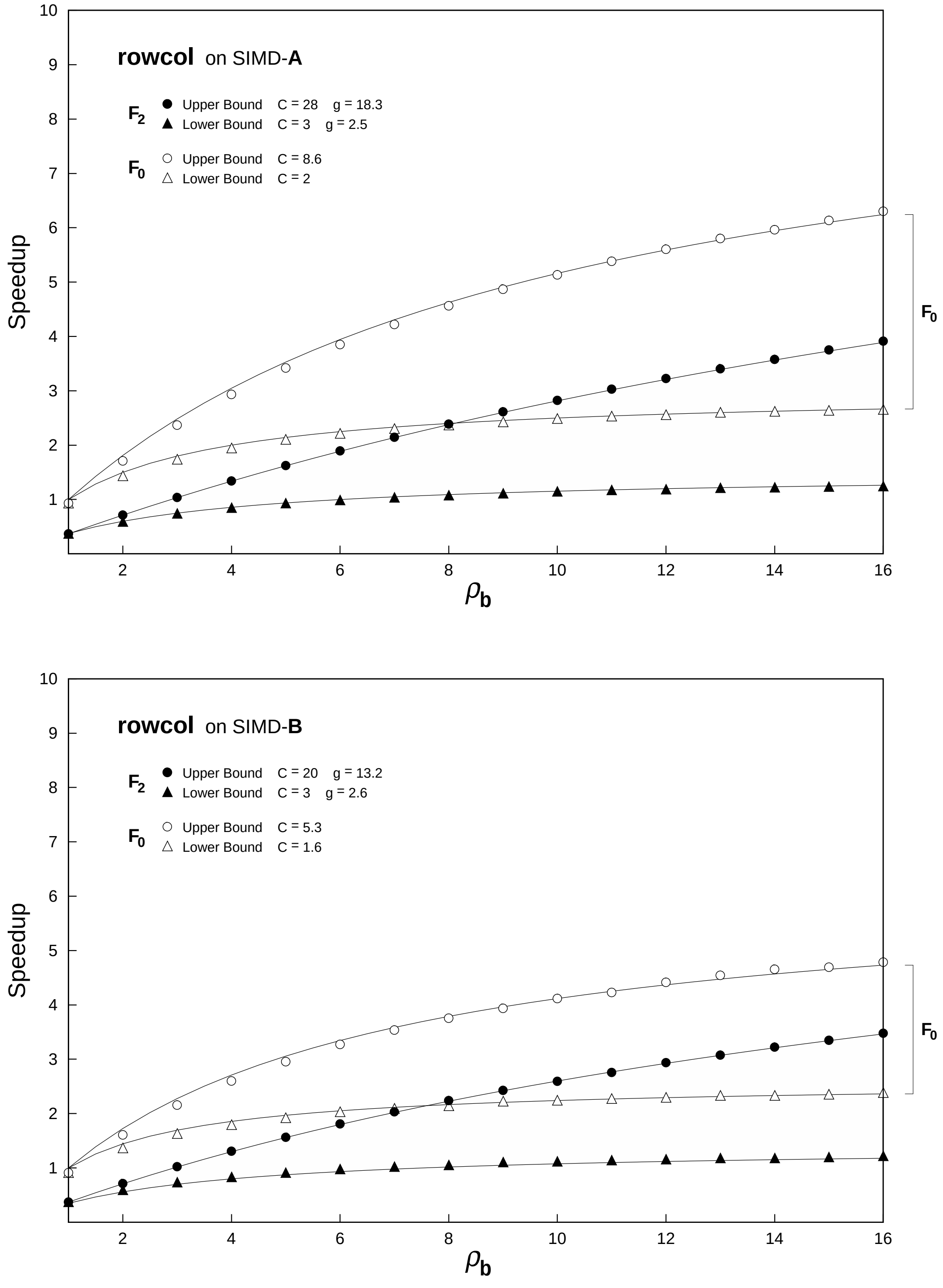
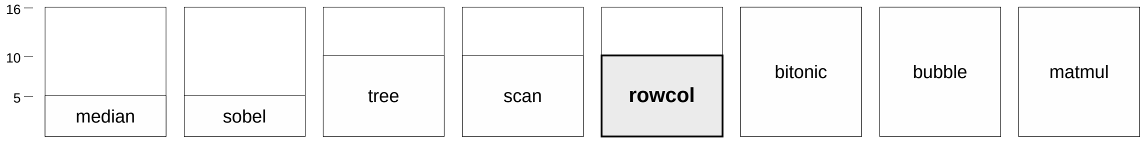


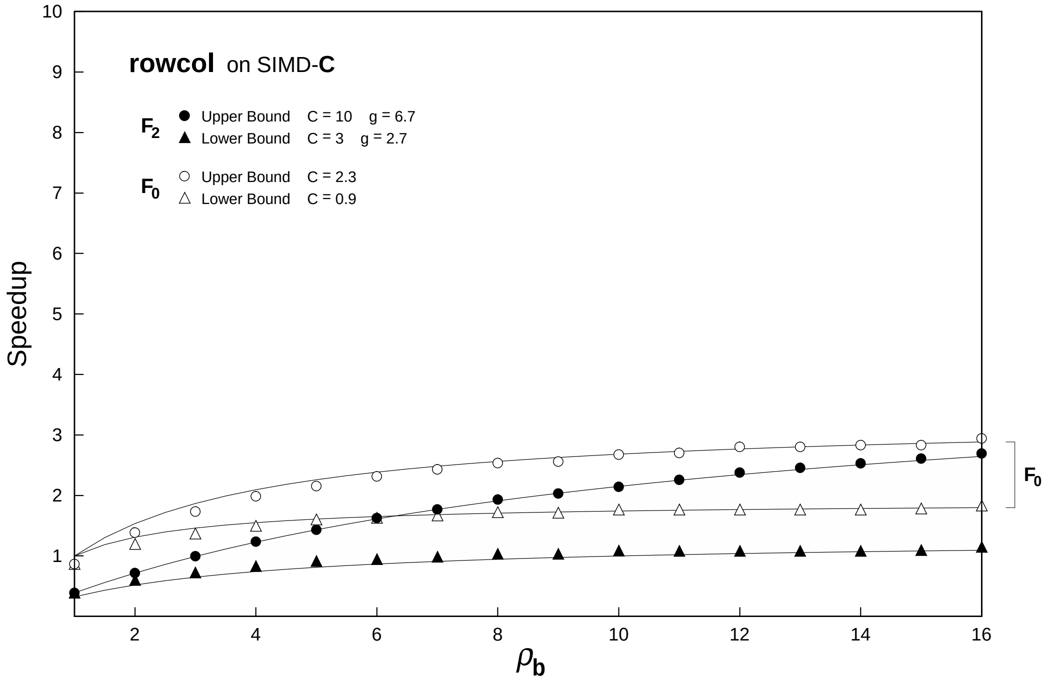
Figure E.4: $\mathbf{F_0}$- and $\mathbf{F_2}$-Speedup Bounds for the program **scan** on SIMD-**A, B, C,** & **D.**

This program consists of two loops that execute successively. The loop bodies do not conflict in cache, so the speedup curves look like those for a program consisting of a single loop.


rowcol on SIMD-A
F2 ● Upper Bound C = 28 g = 18.3
▲ Lower Bound C = 3 g = 2.5
F0 ○ Upper Bound C = 8.6
△ Lower Bound C = 2
F0
Speedup
1
2
3
4
5
6
7
8
9
10
2
4
6
8
10
12
14
16
ρb
rowcol on SIMD-B
F2 ● Upper Bound C = 20 g = 13.2
▲ Lower Bound C = 3 g = 2.6
F0 ○ Upper Bound C = 5.3
△ Lower Bound C = 1.6
F0
Speedup
ρb

(E.5A – E.5B)


16
10
5
median
sobel
tree
scan
rowcol
bitonic
bubble
matmul



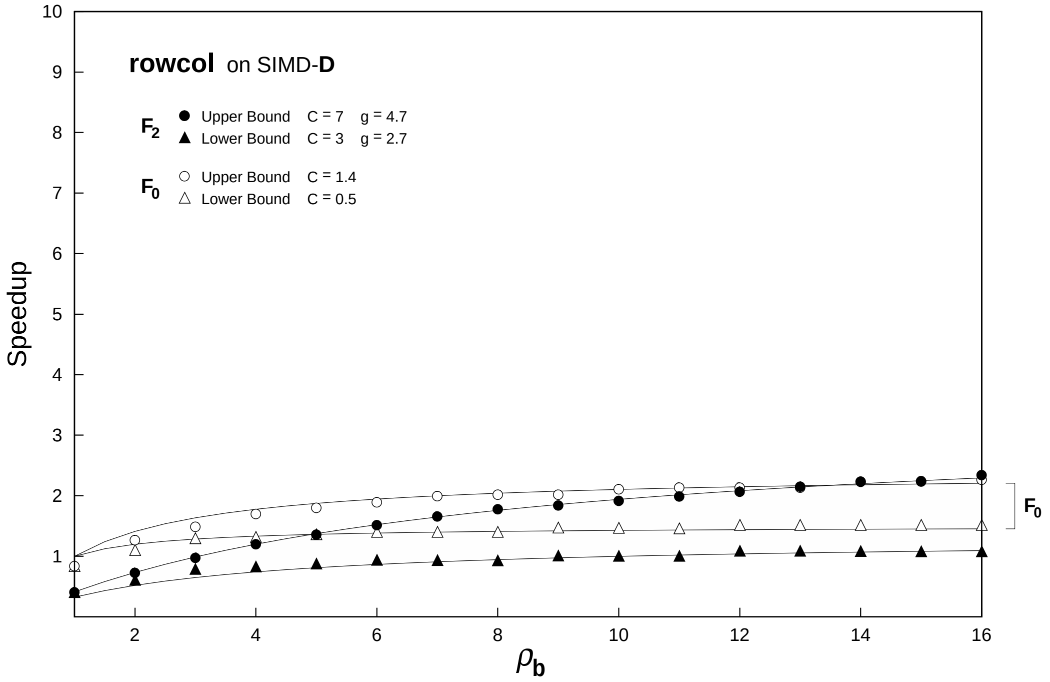
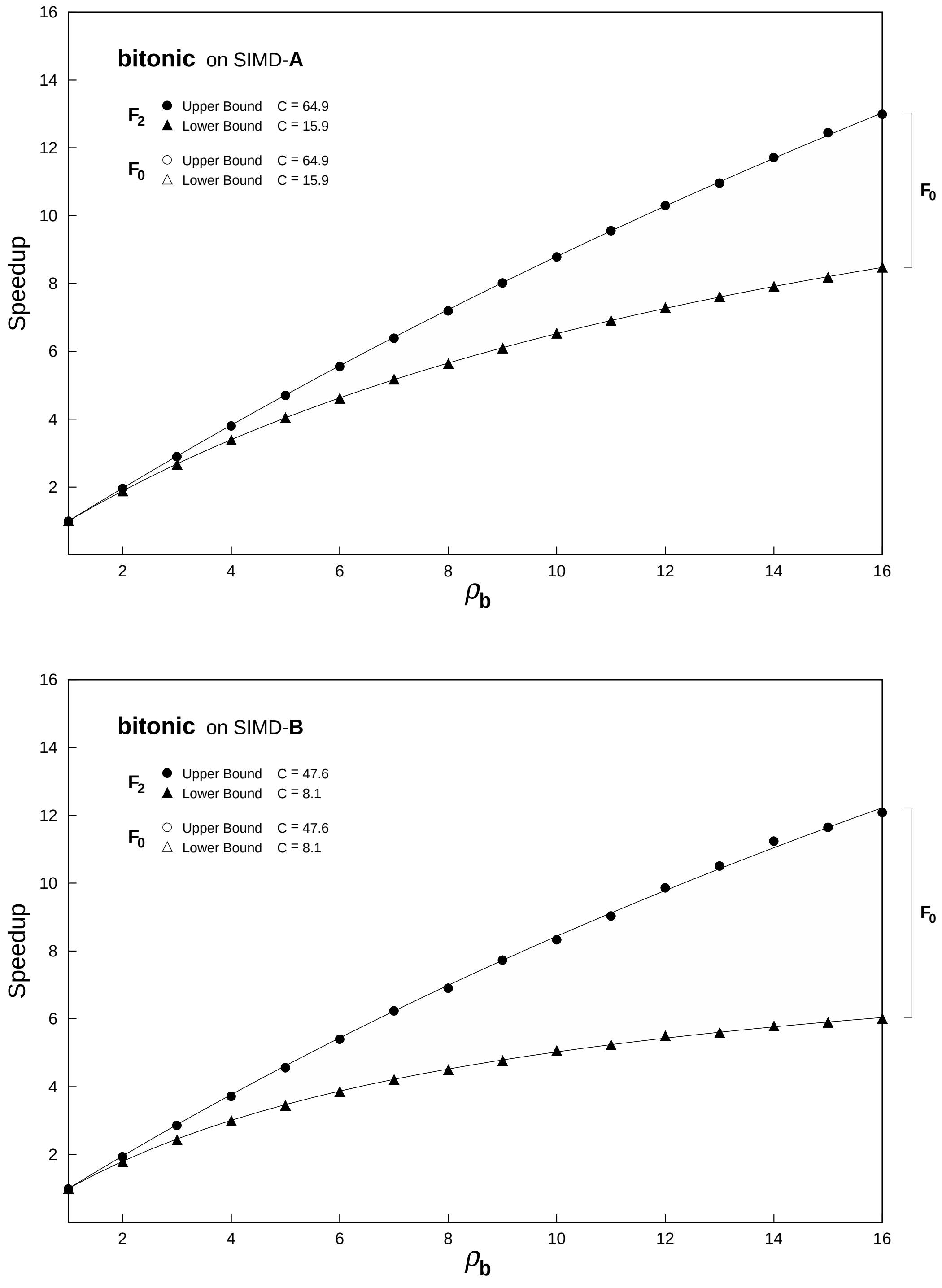
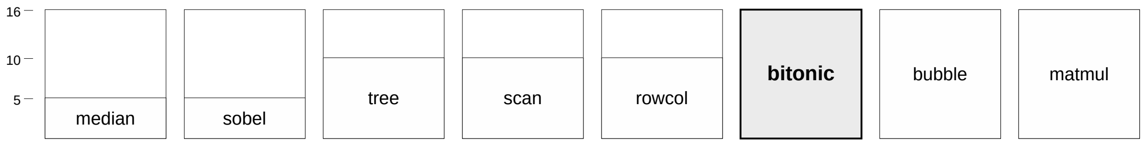


Figure E.5: F_0- and F_2-Speedup Bounds for the program **rowcol** on SIMD-**A, B, C,** & **D.**

The row and column sorts that are the inner loops of this program execute for data-dependent numbers of iterations. The low F_2 speedups show that the management used, effectively unrolling the loops by 4, is disadvantageous. This poor result illustrates the hazards of poor cache management. The F_2 results are so poor that the error term (g) must be included in the simple-equivalent speedup function in order to achieve a good fit.


bitonic on SIMD-A
F2
● Upper Bound C = 64.9
▲ Lower Bound C = 15.9
F0
○ Upper Bound C = 64.9
△ Lower Bound C = 15.9
F0
Speedup
16
14
12
10
8
6
4
2
2
4
6
8
10
12
14
16
ρb
bitonic on SIMD-B
F2
● Upper Bound C = 47.6
▲ Lower Bound C = 8.1
F0
○ Upper Bound C = 47.6
△ Lower Bound C = 8.1
F0
Speedup
ρb

(E.6A – E.6B)


16
10
5
median
sobel
tree
scan
rowcol
bitonic
bubble
matmul

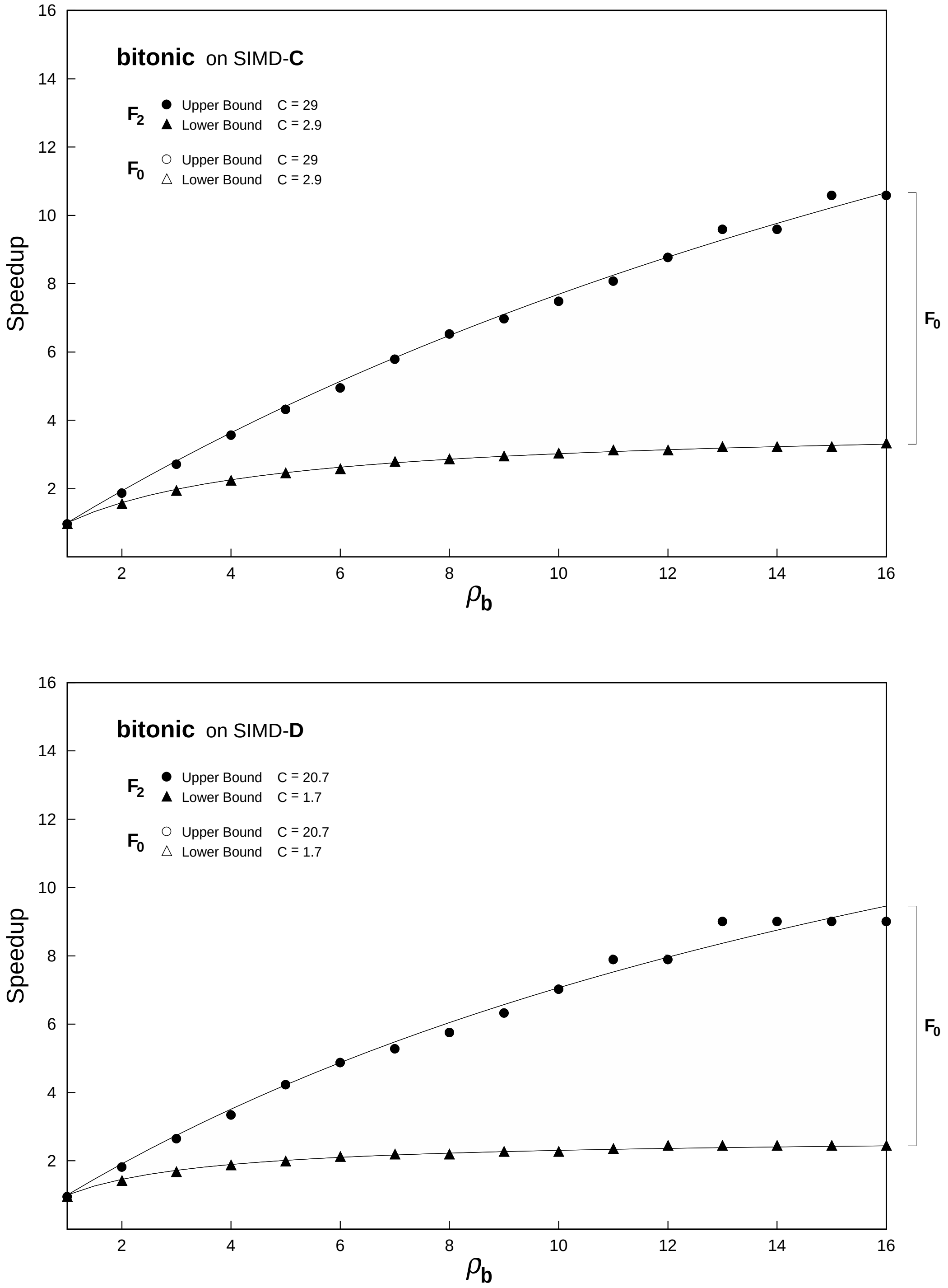
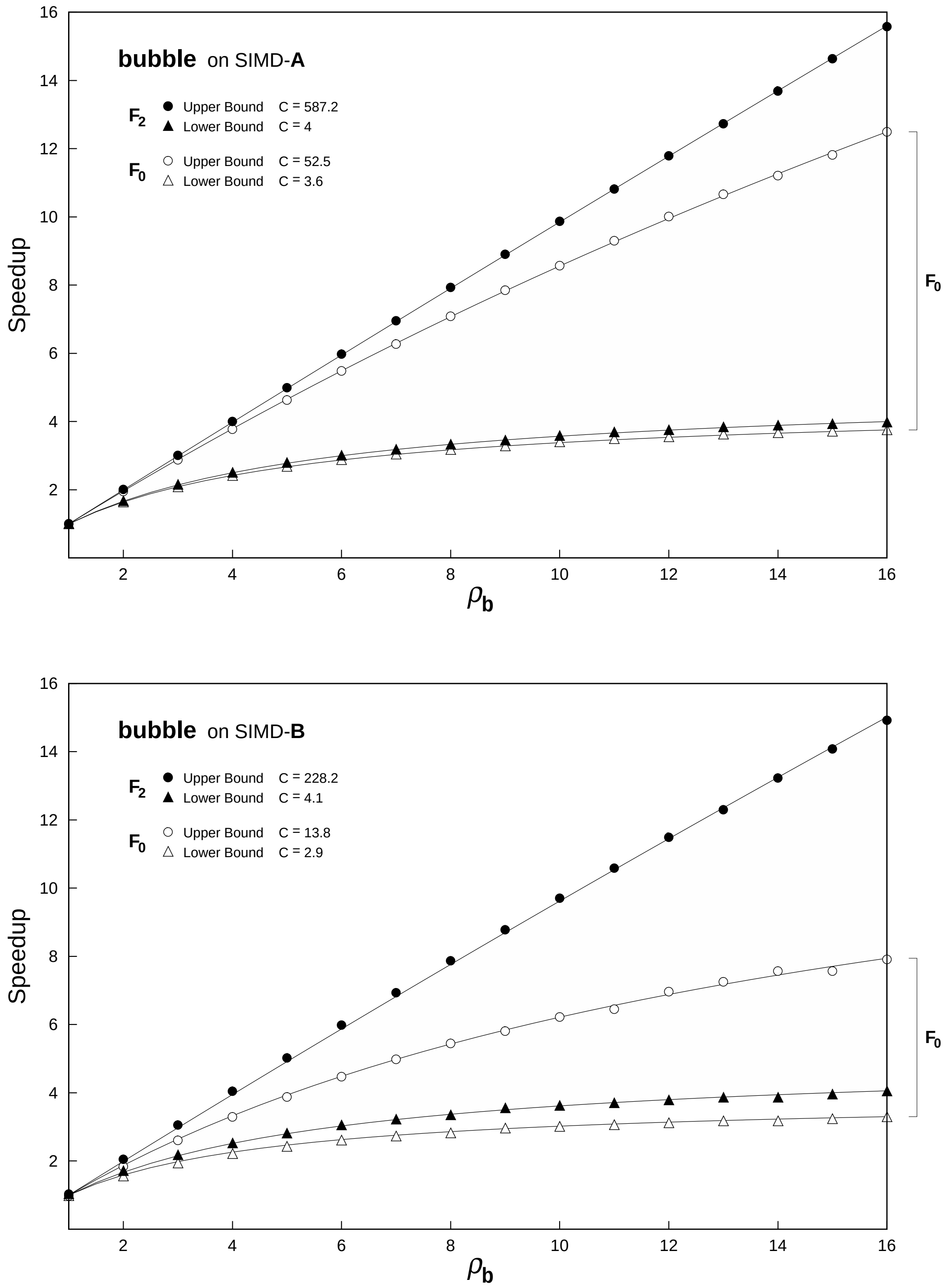
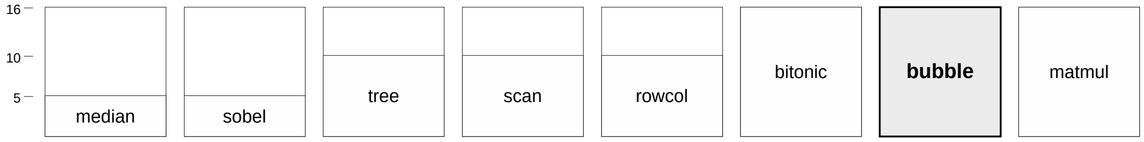


Figure E.6: F_0- and F_2-Speedup Bounds for the program **bitonic** on SIMD-**A, B, C,** & **D.**

The number of iterations of the inner loop varies on each iteration of the outer loop as a function of the outer loop index and the input data set size. The basis computer's system controller lacks the ability to activate F_2 cache blocks for varying numbers of iterations. The resulting management of F_2 I-cache is to iterate cache blocks singly. Therefore, results obtained with F_2 are identical to those obtained with F_0.


bubble on SIMD-A
F2 ● Upper Bound C = 587.2
▲ Lower Bound C = 4
F0 ○ Upper Bound C = 52.5
△ Lower Bound C = 3.6
F0
Speedup
ρb
2 4 6 8 10 12 14 16
bubble on SIMD-B
F2 ● Upper Bound C = 228.2
▲ Lower Bound C = 4.1
F0 ○ Upper Bound C = 13.8
△ Lower Bound C = 2.9
F0
Speedup
ρb
2 4 6 8 10 12 14 16

(E.7A – E.7B)


16
10
5
median
sobel
tree
scan
rowcol
bitonic
bubble
matmul

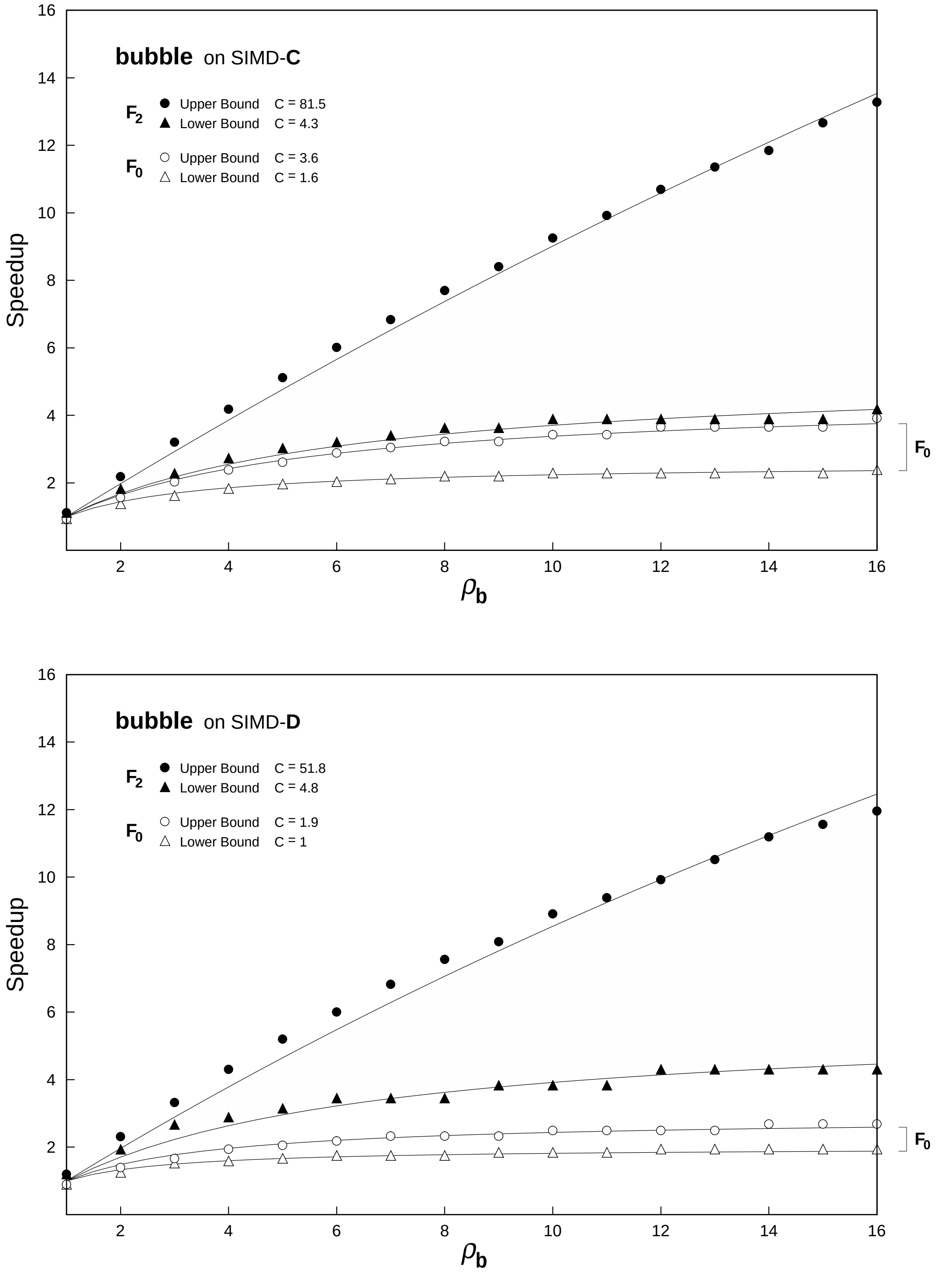
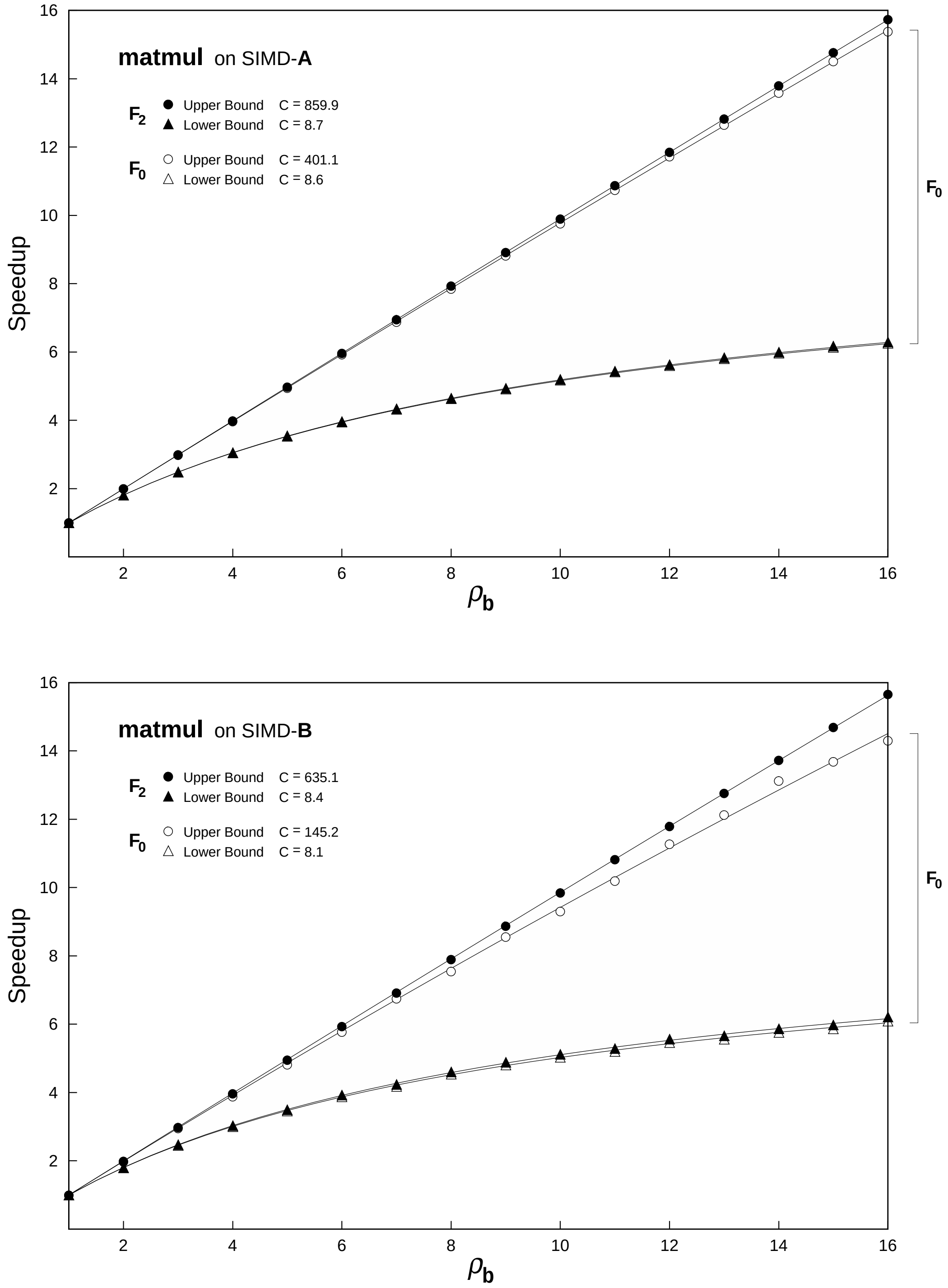
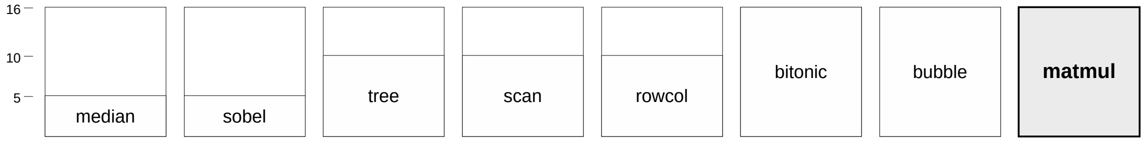


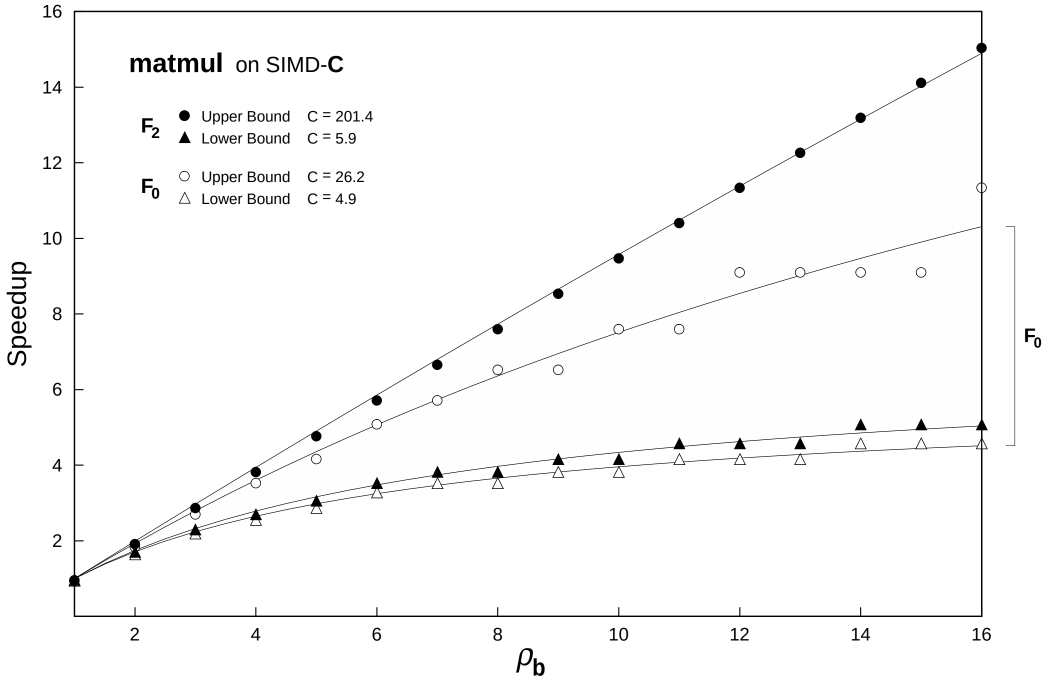
Figure E.7: F_0- and F_2-Speedup Bounds for the program **bubble** on SIMD-**A, B, C,** & **D.**

The difference between the speedup upper bounds is less on SIMD-**A** than on SIMD-**D**. The loop body on SIMD-**A** contains a large number of instructions, as are needed to control the simple FU. Each iteration of the loop body is made nearly ρ_b times faster with F_0 on SIMD-**A**. On SIMD-**D**, whose PEs have a more powerful FU, the speedup is not so great for single iterations, and quantization causes the F_0 speedup upper bound to be much lower than the F_2 upper bound.


matmul on SIMD-A
F2
● Upper Bound C = 859.9
▲ Lower Bound C = 8.7
F0
○ Upper Bound C = 401.1
△ Lower Bound C = 8.6
F0
Speedup
2
4
6
8
10
12
14
16
ρb
matmul on SIMD-B
F2
● Upper Bound C = 635.1
▲ Lower Bound C = 8.4
F0
○ Upper Bound C = 145.2
△ Lower Bound C = 8.1
F0
Speedup
ρb

(E.8A – E.8B)


16
10
5
median
sobel
tree
scan
rowcol
bitonic
bubble
matmul



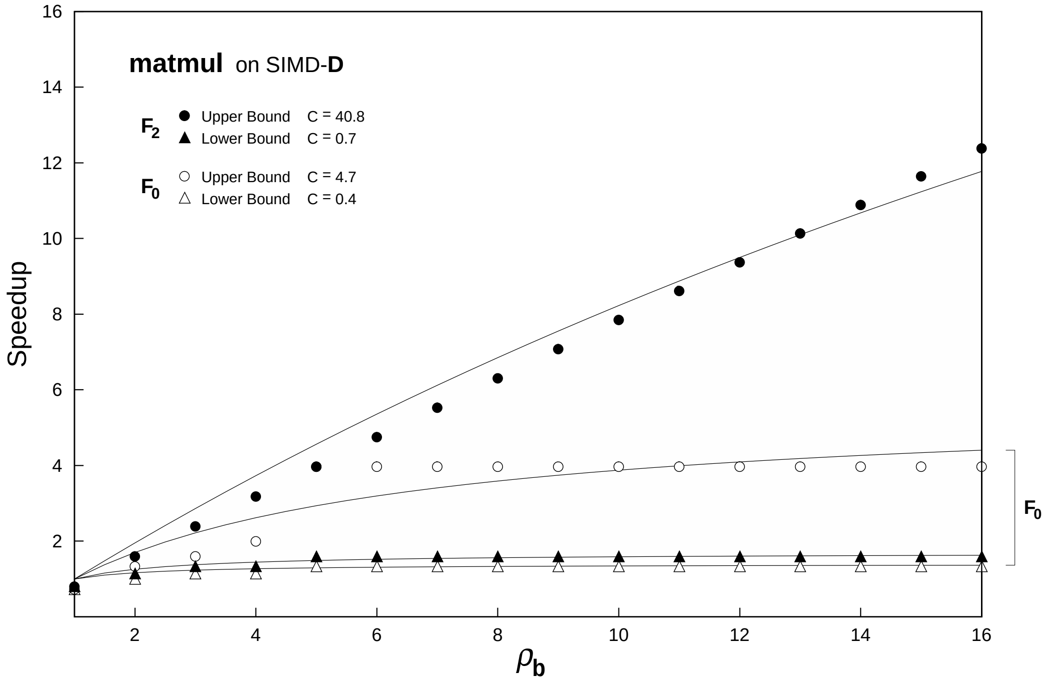


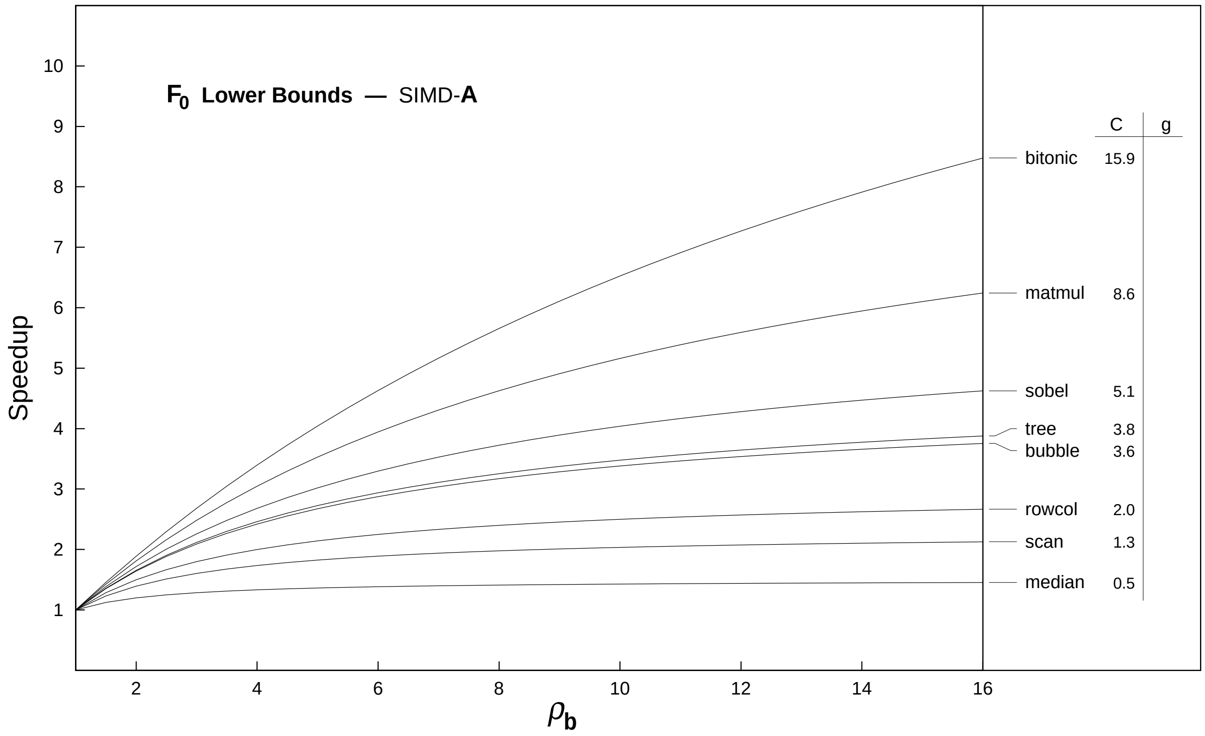
Figure E.8: $\mathbf{F_0}$- and $\mathbf{F_2}$-Speedup Bounds for the program **matmul** on SIMD-**A, B, C,** & **D.**

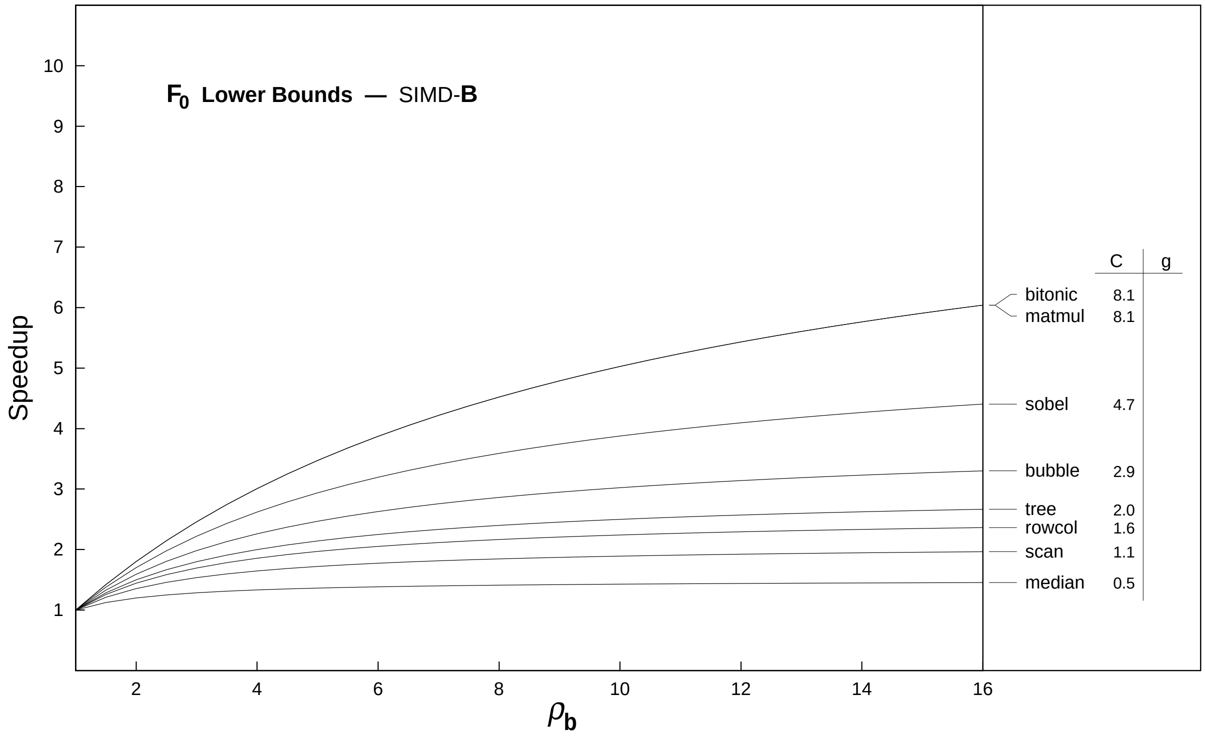
The poor fits of the simple-equivalent speedup curve to the $\mathbf{F_0}$ upper bound on SIMD-**C** and on SIMD-**D** are due to quantization. However, the fit to the $\mathbf{F_2}$ upper bound on SIMD-**D** is also somewhat poor, arising from a slowdown at $\rho_b = 1$. This poor fit suggests that it is not so appropriate to discard the simple-equivalent speedup formula error term in this case as it is for most of the other cases.

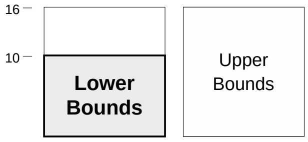
Appendix F

Summary of F_0 Speedup Bounds

To facilitate comparison of the range of I-cache bounds, a complete set of “simple-equivalent” F_0 speedup lower bounds measured on each SIMD computer variant is plotted on one graph, and a complete set of “simple-equivalent” F_0 speedup upper bounds measured on each SIMD computer variant is also plotted on one graph.


F_0 Lower Bounds — SIMD-A
Speedup
1
2
3
4
5
6
7
8
9
10
2
4
6
8
10
12
14
16
ρ_b
C
g
bitonic 15.9
matmul 8.6
sobel 5.1
tree 3.8
bubble 3.6
rowcol 2.0
scan 1.3
median 0.5


F_0 Lower Bounds — SIMD-B
Speedup
1
2
3
4
5
6
7
8
9
10
2
4
6
8
10
12
14
16
ρ_b
C
g
bitonic 8.1
matmul 8.1
sobel 4.7
bubble 2.9
tree 2.0
rowcol 1.6
scan 1.1
median 0.5


16
10
Lower Bounds
Upper Bounds

(F.2A – F.2B)

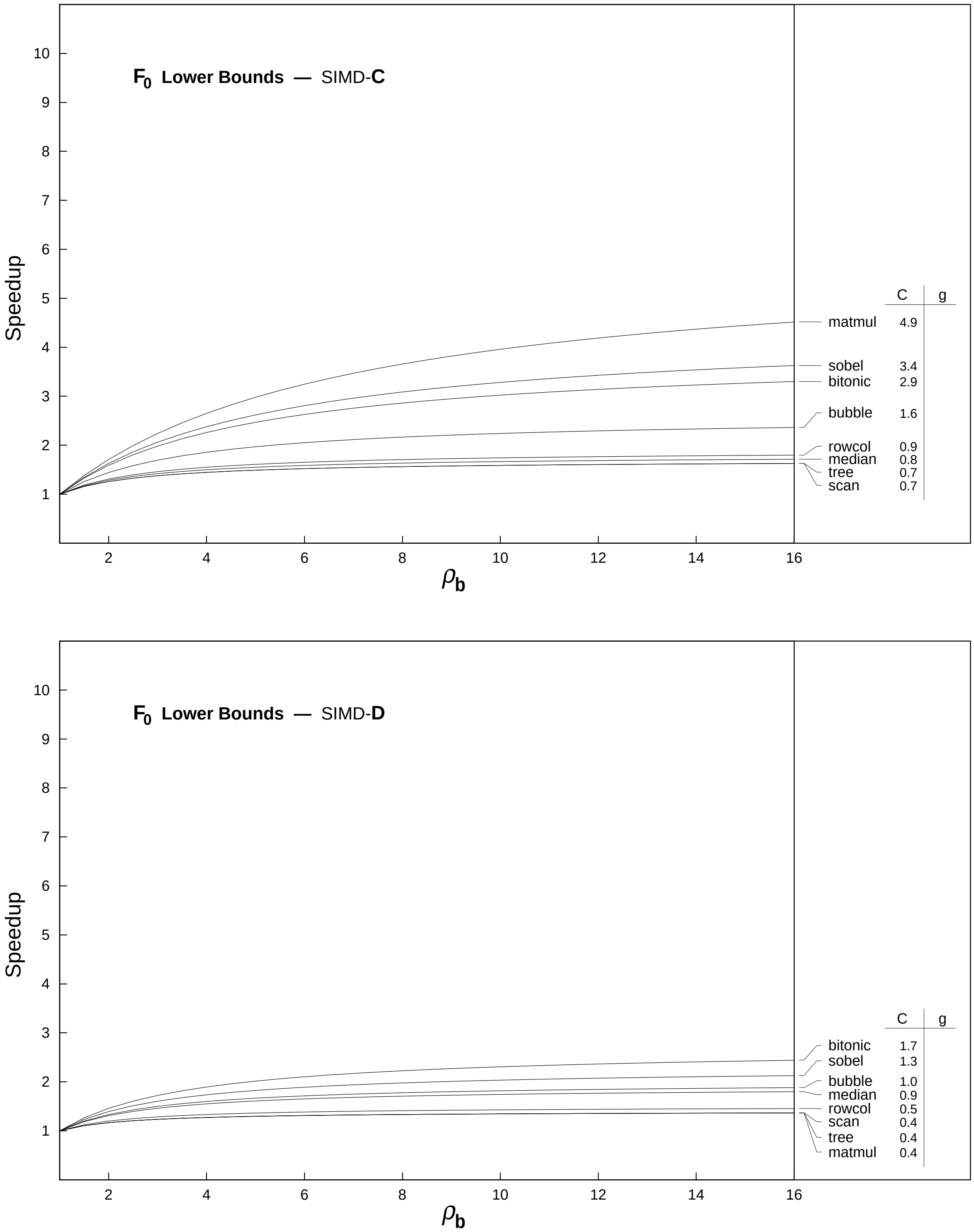
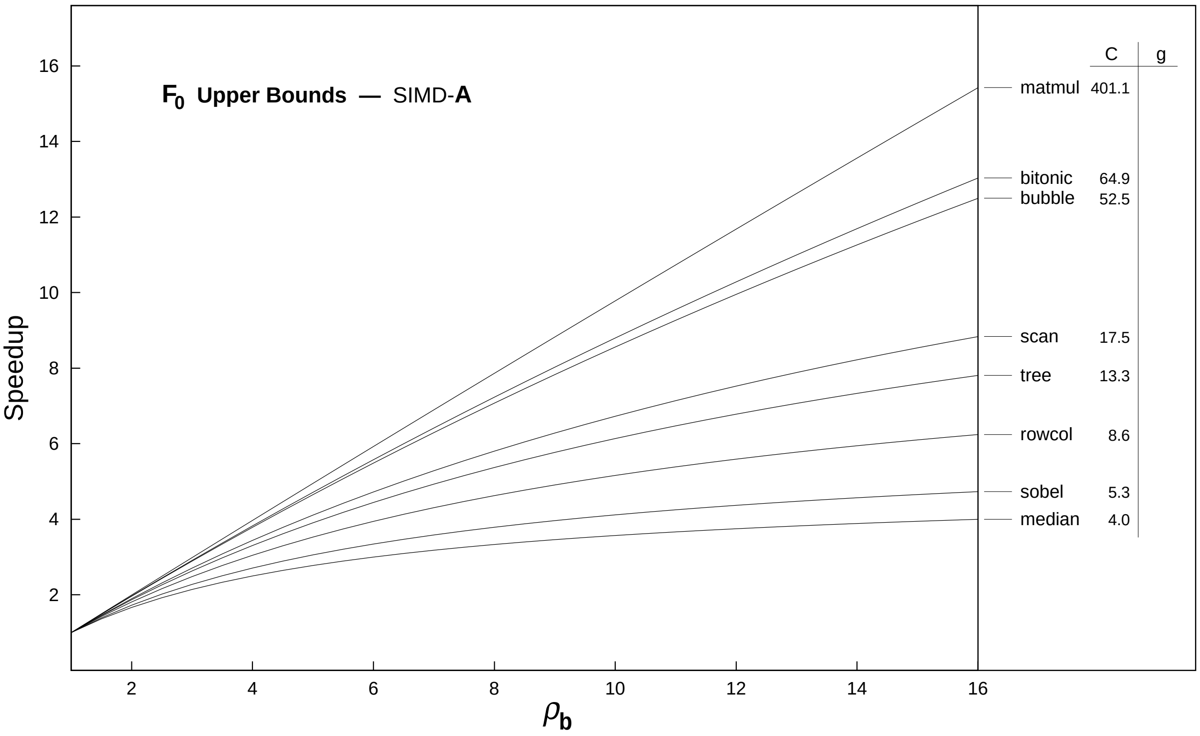
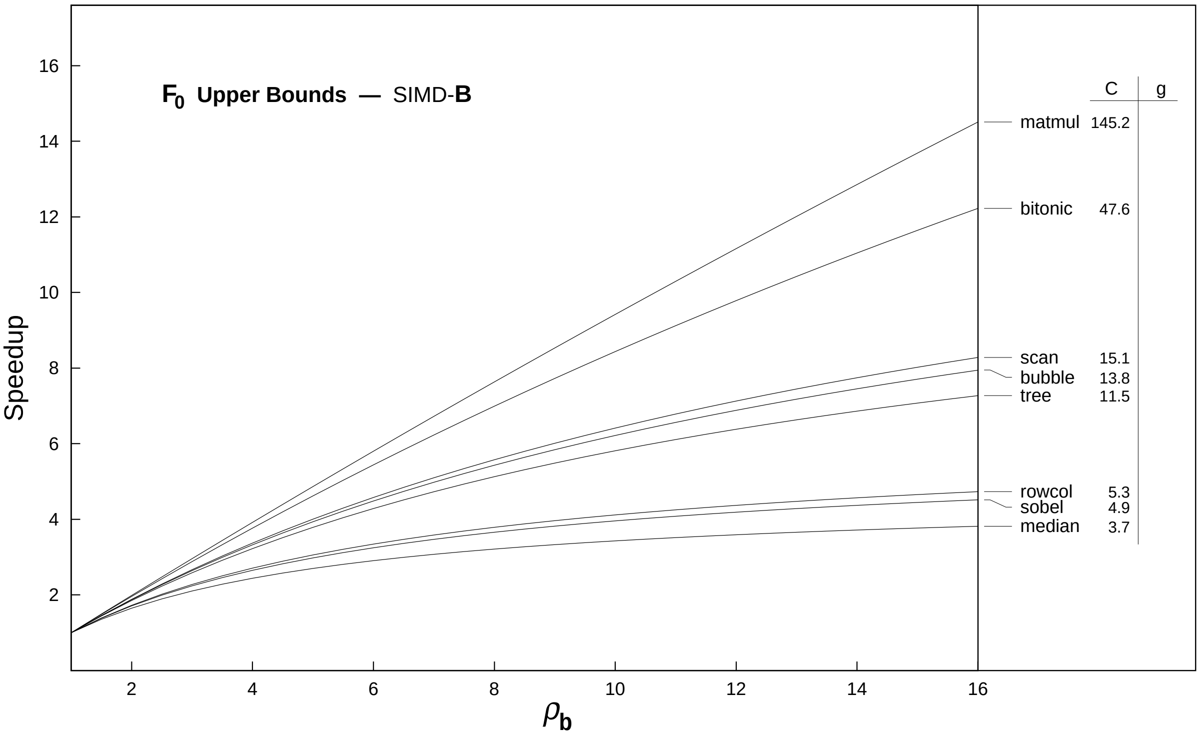
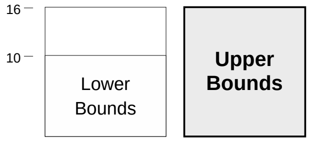


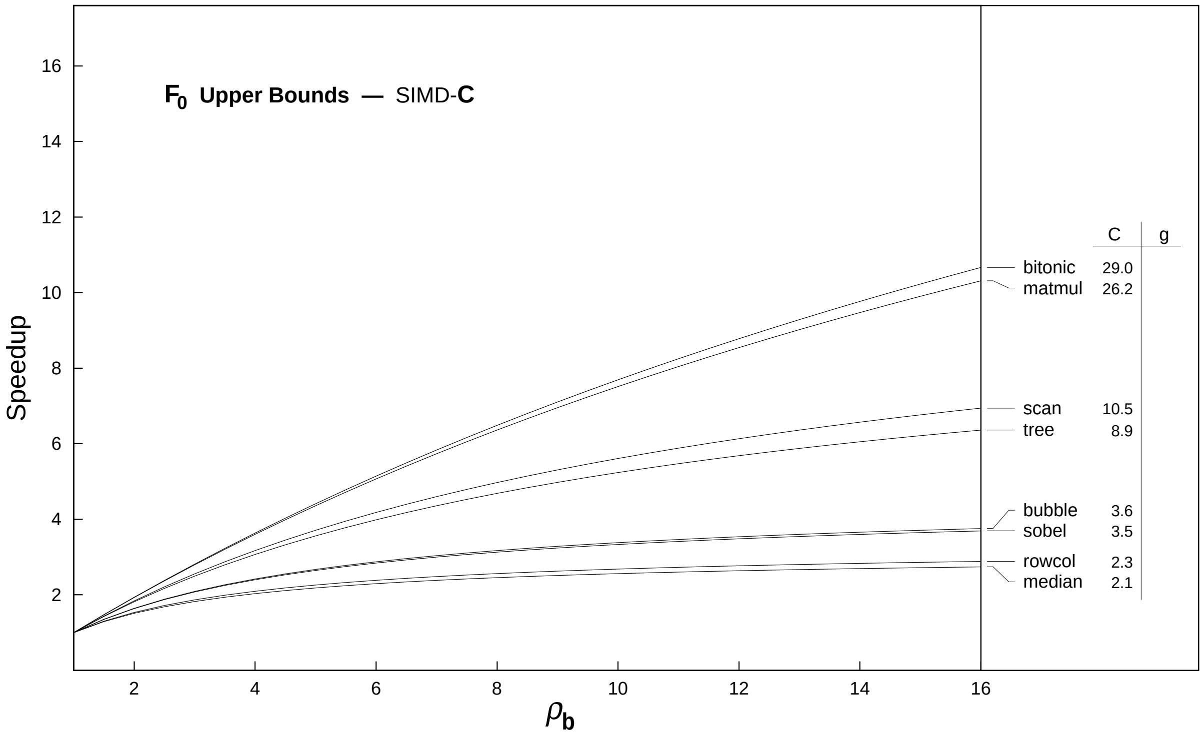
Figure F.2: A summary of **F_0 Lower Bounds** on the subject programs in SIMD-**A, B, C,** & **D.** Curve spreads decrease as the hardware variant becomes more powerful.


F_0 Upper Bounds — SIMD-A
Speedup
ρ_b
2
4
6
8
10
12
14
16
C
g
matmul 401.1
bitonic 64.9
bubble 52.5
scan 17.5
tree 13.3
rowcol 8.6
sobel 5.3
median 4.0


F_0 Upper Bounds — SIMD-B
Speedup
ρ_b
2
4
6
8
10
12
14
16
C
g
matmul 145.2
bitonic 47.6
scan 15.1
bubble 13.8
tree 11.5
rowcol 5.3
sobel 4.9
median 3.7


16
10
Lower Bounds
Upper Bounds

(F.1A – F.1B)



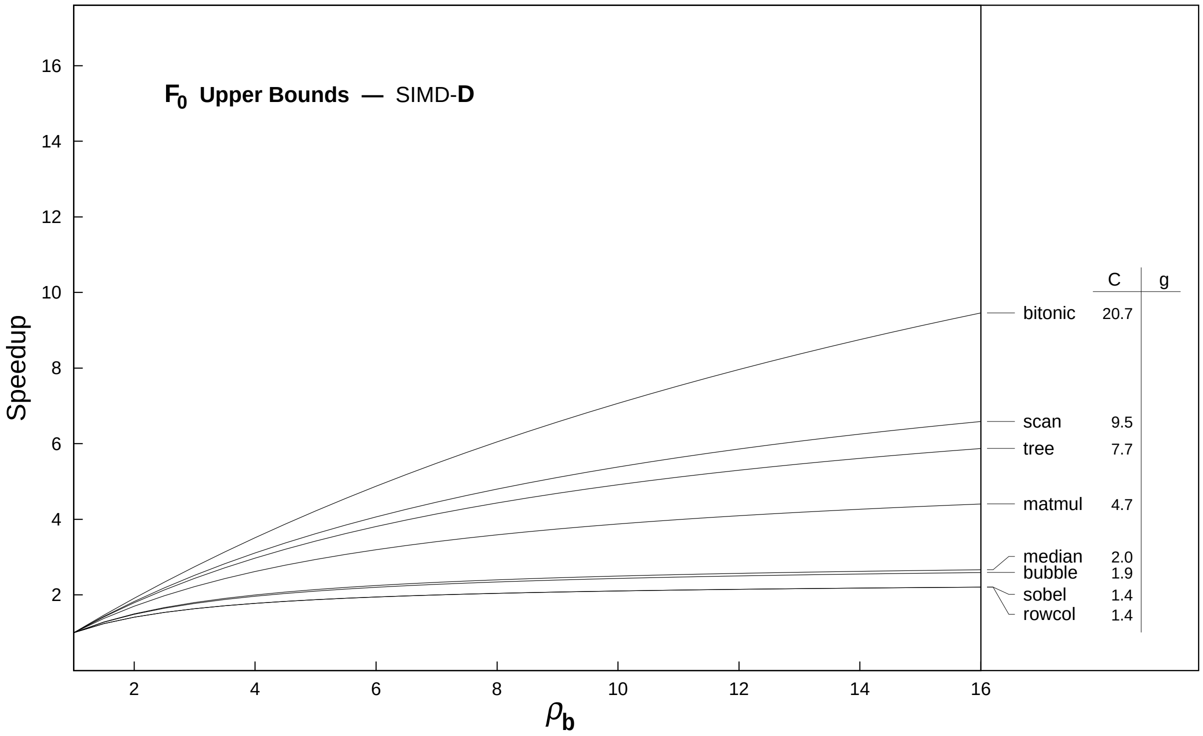
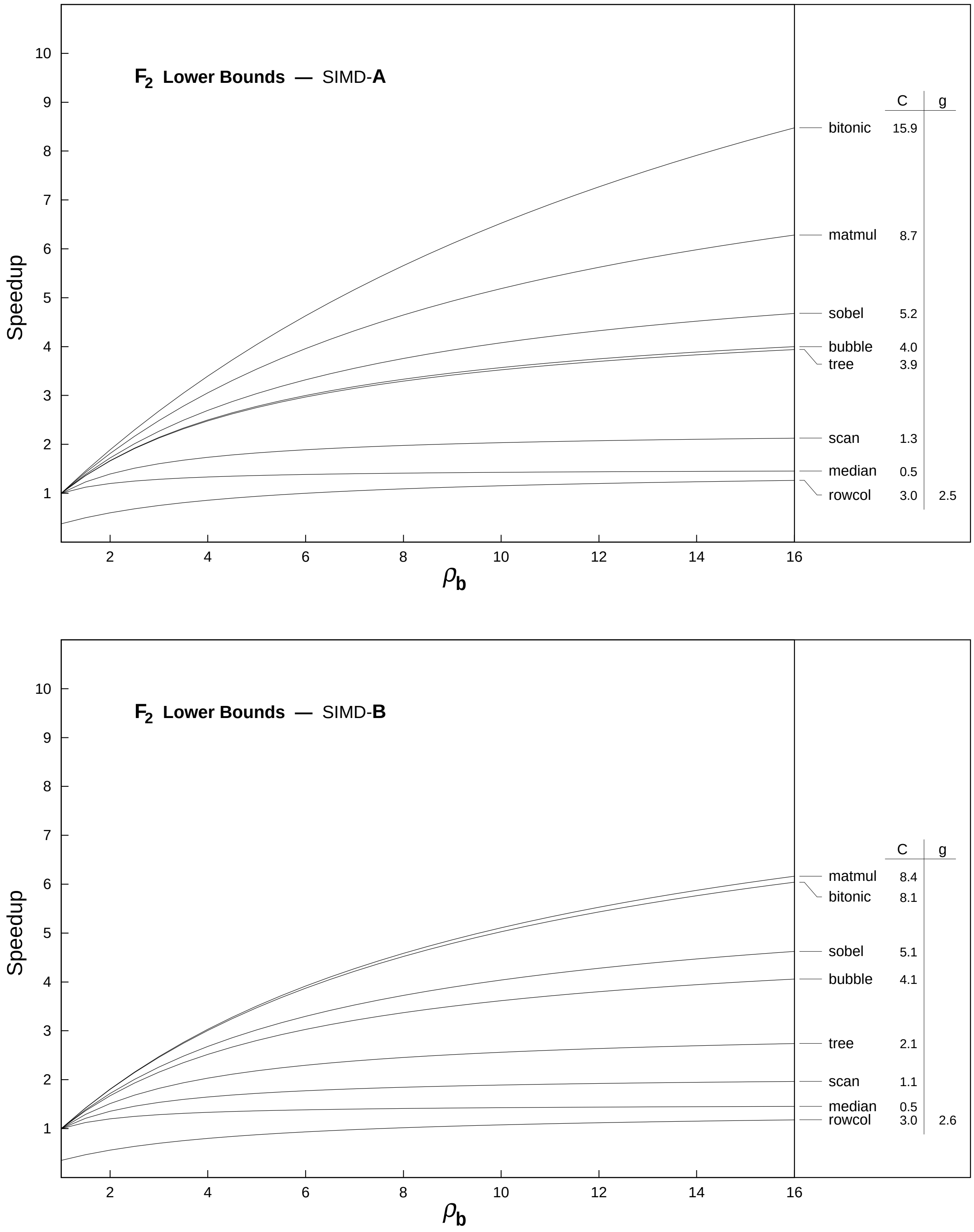


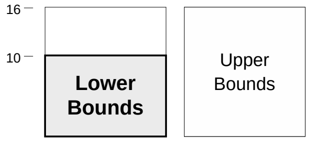
Figure F.1: A summary of **F_0 Upper Bounds** on the subject programs in SIMD-**A, B, C,** & **D.** Quantization for matmul reorders the speedups on SIMD-**D**.

Appendix G

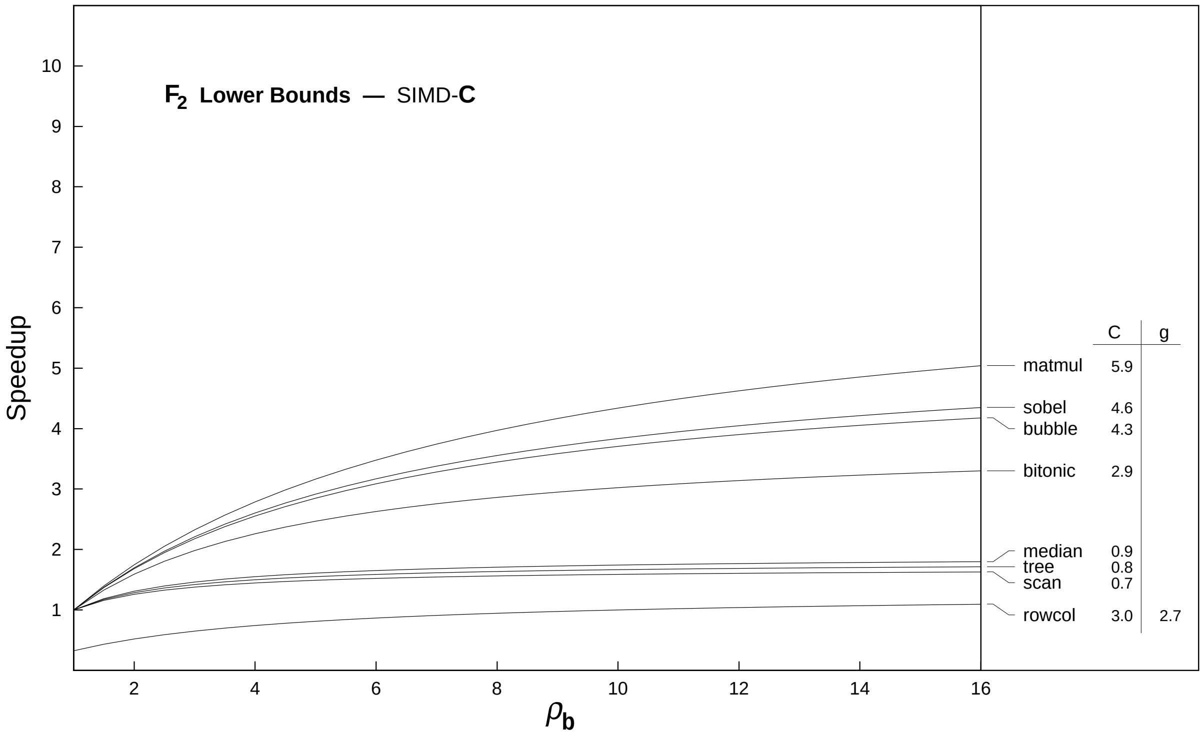
Summary of F_2 Speedup Bounds

To facilitate comparison of the range of I-cache bounds, a complete set of “simple-equivalent” F_2 speedup lower bounds measured on each SIMD computer variant is plotted on one graph, and a complete set of “simple-equivalent” F_2 speedup upper bounds measured on each SIMD computer variant is also plotted on one graph.


F2 Lower Bounds — SIMD-A
Speedup
1
2
3
4
5
6
7
8
9
10
ρb
2
4
6
8
10
12
14
16
C
g
bitonic 15.9
matmul 8.7
sobel 5.2
bubble 4.0
tree 3.9
scan 1.3
median 0.5
rowcol 3.0 2.5
F2 Lower Bounds — SIMD-B
Speedup
ρb
C
g
matmul 8.4
bitonic 8.1
sobel 5.1
bubble 4.1
tree 2.1
scan 1.1
median 0.5
rowcol 3.0 2.6


16
10
Lower Bounds
Upper Bounds

(G.2A – G.2B)



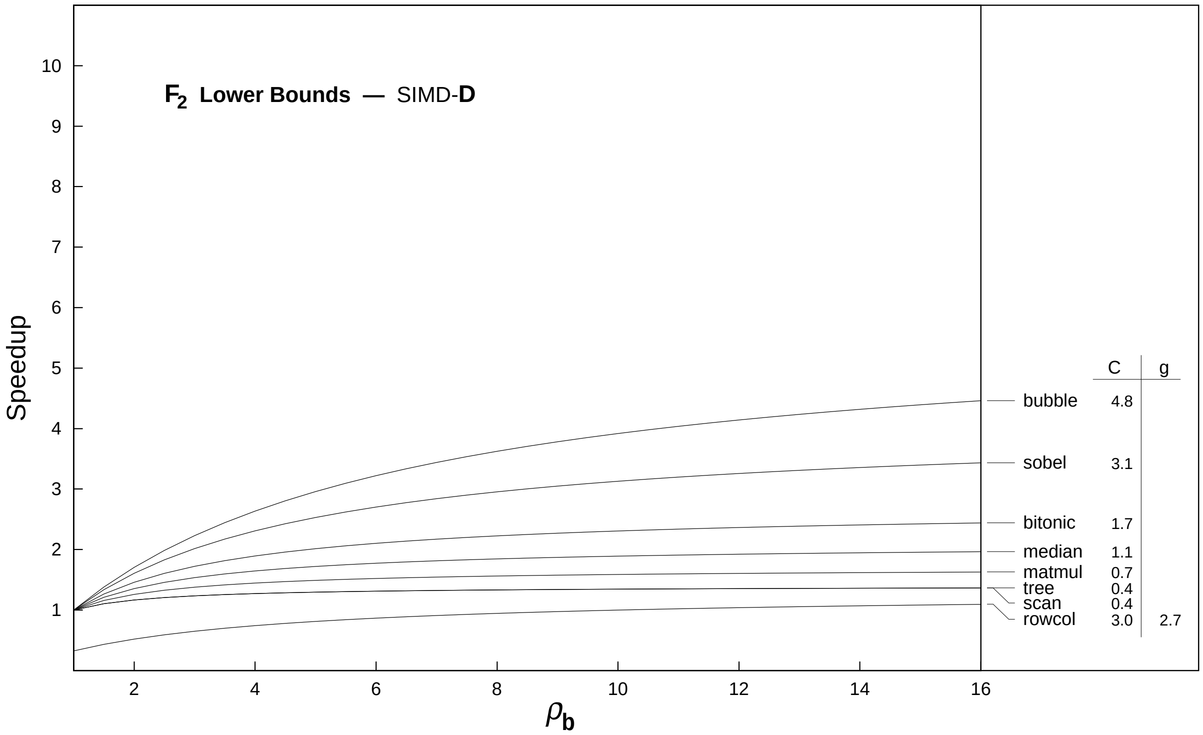
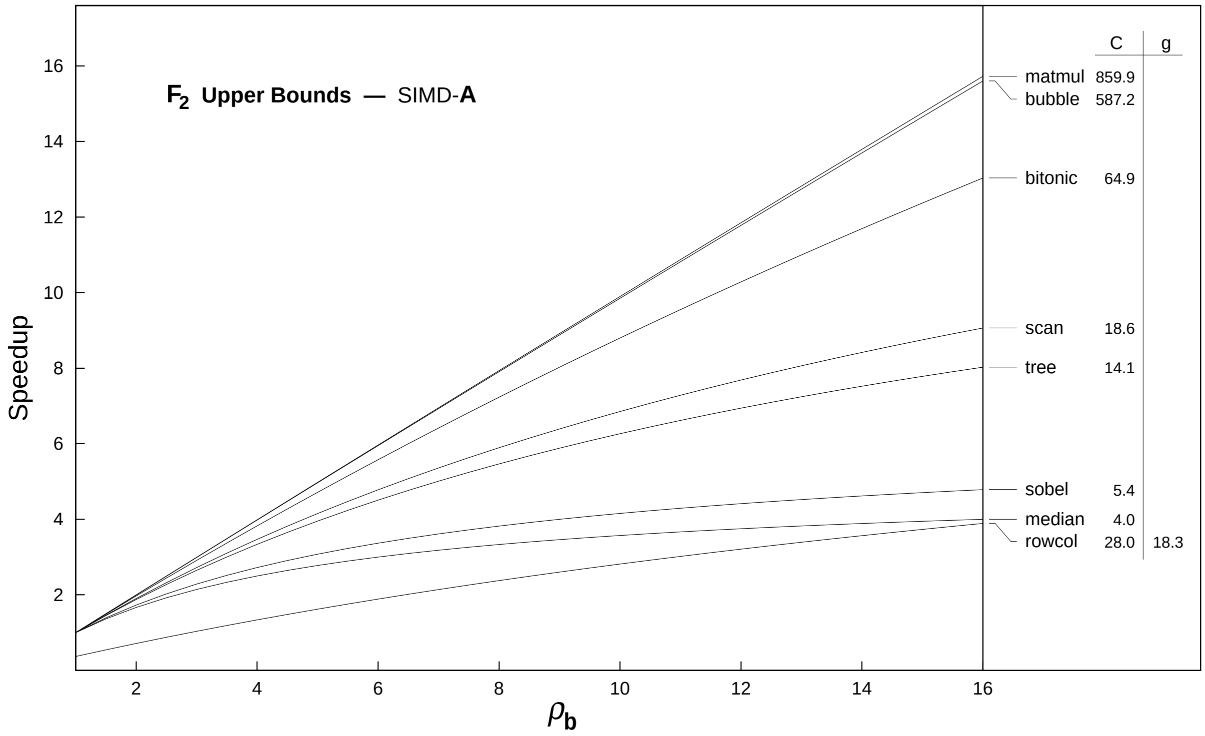
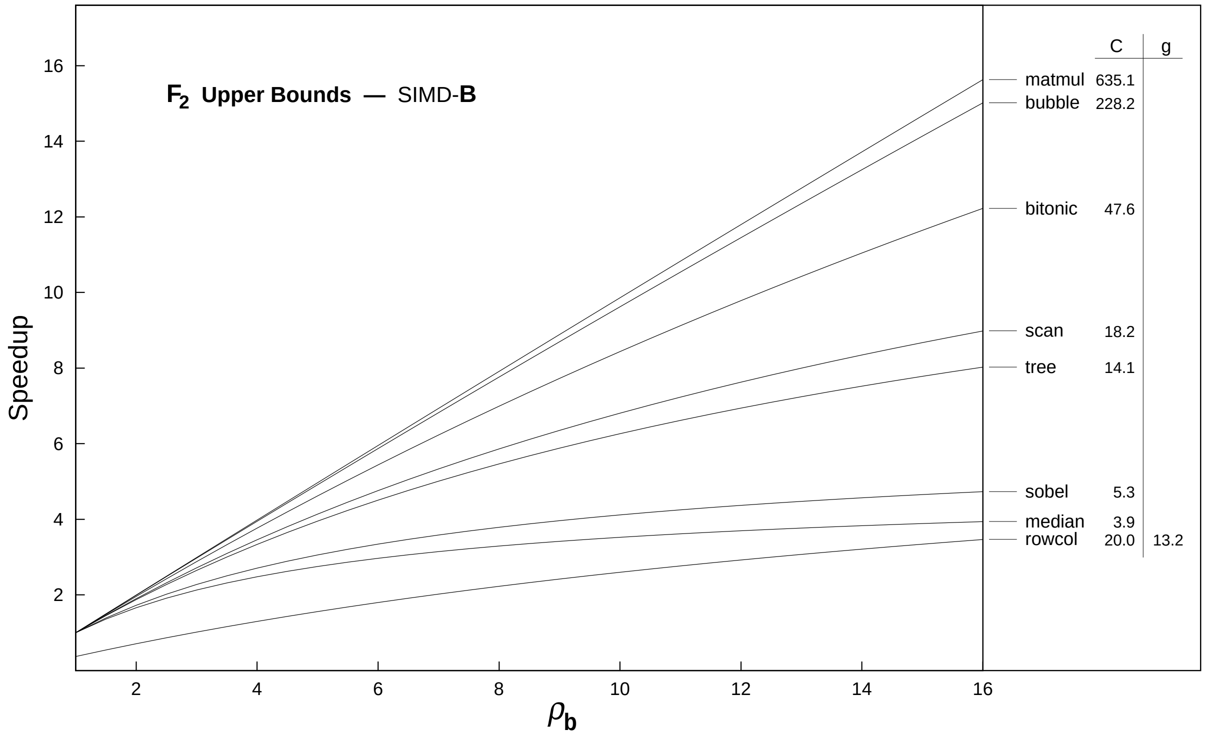
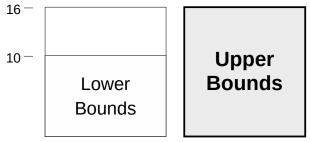


Figure G.2: A summary of **F_2 Lower Bounds** on the subject programs in SIMD-**A, B, C,** & **D.** SIMD-**C** and D I-cache speedups are very similar.


F2 Upper Bounds — SIMD-A
Speedup
2
4
6
8
10
12
14
16
ρb
2
4
6
8
10
12
14
16
C
g
matmul 859.9
bubble 587.2
bitonic 64.9
scan 18.6
tree 14.1
sobel 5.4
median 4.0
rowcol 28.0 18.3


F2 Upper Bounds — SIMD-B
Speedup
2
4
6
8
10
12
14
16
ρb
2
4
6
8
10
12
14
16
C
g
matmul 635.1
bubble 228.2
bitonic 47.6
scan 18.2
tree 14.1
sobel 5.3
median 3.9
rowcol 20.0 13.2


16
10
Lower Bounds
Upper Bounds

(G.1A – G.1B)

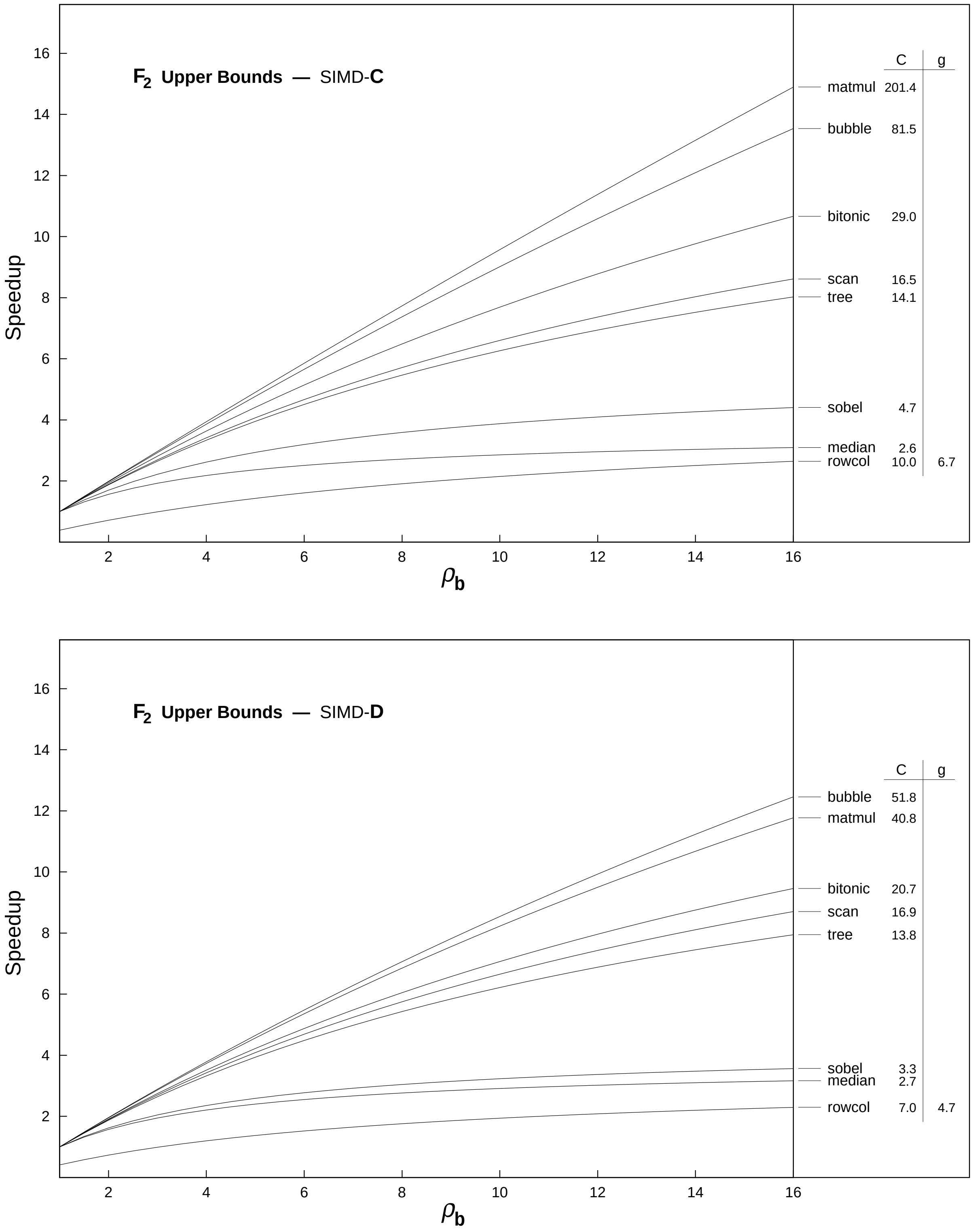


Figure G.1: A summary of **F_2 Upper Bounds** on the subject programs in SIMD-**A, B, C,** & **D.** I-cache speedups are greatest for the SIMD computer variants with the least powerful PE FUs.

Bibliography

[1] H. B. Bakoglu. *Circuits, Interconnections, and Packaging for VLSI*. VLSI Systems Series. Addison-Wesley Publishing Company, Reading, MA, 1990.

[2] Dana H. Ballard and Christopher M. Brown. *Computer Vision*. Prentice-Hall, New Jersey, 1982.

[3] George H. Barnes, Richard M. Brown, Maso Kato, David J. Kuck, Daniel L. Slotnick, and Richard A. Stokes. The ILLIAC IV computer. *IEEE Transactions on Computers*, pages 746–757, 1968.

[4] Kenneth E. Batcher. Design of a massively parallel processor. *IEEE Transactions on Computers*, pages 836–840, September 1980.

[5] Mel Bazes and Roni Ashuri. A novel CMOS digital clock and data decoder. *IEEE Journal of Solid-State Circuits*, 27(12):1934–1940, December 1992.

[6] J. Beetem, M. Denneau, and D. Weingarten. The GF11 supercomputer. In *The 12th Annual International Symposium on Computer Architecture*, pages 108–115. IEEE Computer Society, June 1985.

[7] Gordon Bell. Ultracomputers: A Teraflop before its time. *CACM*, 35(8):26–47, August 1992.

[8] Tom Blank. The MasPar MP-1 architecture. In *Proceedings of IEEE Compcon Spring 1990*. IEEE, February 1990.

[9] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, November 1989.

[10] Timothy Bridges. The GPA machine: A generally partitionable MSIMD architecture. In *The Third Sympoisum on the Frontiers of Massively Parallel Computation*. IEEE, October 1990.

[11] Peter Calingaert. *Operating System Elements*. Prentice-Hall, Englewood Cliffs, N.J., 1982.

[12] Dao-Long Chen. Designing on-chip clock generators. *IEEE Circuits and Devices Magazine*, pages 32–36, July 1992.

[13] Vernon L. Chi. Salphasic distribution of clock signals. Technical Report 90-026, University of North Carolina at Chapel Hill, Department of Computer Science, June 1990.

[14] F. Chow and J. Hennessy. Register allocation by priority-based coloring. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 222–232. ACM SIGPLAN, 1984.

[15] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. P. Papworth, and P. K. Rodman. A VLIW architecture for a Trace Scheduling compiler. *IEEE Transactions on Computers*, pages 967–879, 1988.

[16] Jon Wade (Thinking Machines Corporation). Easy questions about cm-2 chips. Electronic mail message, July 1993. A response to a query through Skef Wholey.

[17] Skef Wholey (Thinking Machines Corporation). Cm-200 specs. Electronic mail message, March 1993.

[18] Thinking Machines Corporation. Connection machine model CM-2 technical summary. Technical Report HA87-4, Thinking Machines Corporation, Boston, MA., April 1987.

[19] William J. Dally. *A VLSI Architecture for Concurrent Data Structures*. PhD thesis, California Institute of Technology, 1986.

[20] Peter J. Denning and Walter F. Tichy. Highly parallel computation. *Science*, 250(4985):1217–1222, November 1990.

[21] Dan Dobberpuhl, Richard Witek, et al. A 200MHz 64b dual-issue CMOS microprocessor. *IEEE Journal of Solid-State Circuits*, 27(11):1555–1567, November 1992.

[22] David C. Douglas, Brewster A. Kahle, and Alex Vasilevsky. The architecture of the CM-2 data processor. Technical Report HA88-1, Thinking Machines Corporation, February 1988.

[23] Frederico Faggin. How VLSI impacts computer architecture. *IEEE Spectrum*, pages 28–31, May 1978.

[24] Allan L. Fisher. *Implementation Issues for Algorithmic VLSI Processor Arrays*. PhD thesis, Carnegie Mellon University, November 1984.

[25] Allan L. Fisher and Peter T. Highnam. Real-time image processing on scan line array processors. In *IEEE Computer Society Workshop on Computer Architectures for Pattern Analysis and Image Database Management*. IEEE, November 1985.

[26] Allan L. Fisher, Peter T. Highnam, and Todd E. Rockoff. Architecture of a VLSI SIMD processing element. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers & Processors*, pages 324–327. IEEE, May 1987.

[27] Allan L. Fisher, Peter T. Highnam, and Todd E. Rockoff. A four-processor building block for SIMD processor arrays. *IEEE Journal of Solid State Circuits*, 25(2):369–375, April 1990.

[28] Allan L. Fisher and John A. Zsarnay. System support for a VLSI SIMD image computer. In Robert W. Brodersen and Howard S. Moscovitz, editors, *VLSI Signal Processing, III*, chapter 14, pages 141–147. IEEE Press, 1988.

[29] J. A. Fisher. Trace Scheduling: A technique for global microcode compaction. *IEEE Transactions on Computing*, pages 478–490, 1981.

[30] P. M. Flanders, D. J. Hunt, S. F. Reddaway, and D. Parkinson. Efficient high speed computing with the distributed array processor. In *High Speed Computer and Algorithm Organization*, pages 113–128. Academic Press, Inc., New York, 1977.

[31] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

[32] Terry J. Fountain, K. N. Matthews, and Michael J. B. Duff. The CLIP7A image processor. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(3):310–319, May 1988.

[33] Robert Grondalski. A VLSI chip set for a massively parallel architecture. In *1987 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 198–199. IEEE, February 1987.

[34] T. R. Gross. *Code Optimization of Pipeline Constraints.* PhD thesis, Stanford University, 1983.

[35] Hiroyuki Hara et al. 0.5-μm 3.3-v BiCMOS standard cells with 32-kilobyte cache and ten-port register file. *IEEE Journal of Solid-State Circuits*, 27(11):1579–1584, November 1992.

[36] R. Heaton, D. Blevins, and E. Davis. A bit-serial VLSI array processing chip for image processing. *IEEE Journal of Solid State Circuits*, 25(2):364–368, April 1990.

[37] R. A. Heaton and D. W. Blevins. BLITZEN: A VLSI array processing chip. In *Proceedings of the IEEE 1989 Custom Integrated Circuits Conference.* IEEE, May 1989.

[38] John L. Hennessy. VLSI processor architecture. *IEEE Transactions on Computers*, 33(12):1221–1246, December 1984.

[39] Peter T. Highnam. *Systems and Programming Issues in the Design and Use of a SIMD Linear Array for Image Processing.* PhD thesis, Carnegie Mellon University, April 1991.

[40] Danny Hillis. CM-5 connection machine. Internet post to comp.arch, November 1991.

[41] W. D. Hillis and G. L. Steele Jr. Data parallel algorithms. *CACM*, pages 1170–1183, December 1986.

[42] W. Daniel Hillis. *The Connection Machine.* The MIT Press, Cambridge, MA, 1985.

[43] B. Hoeneisen and C. A. Mead. Fundamental limitations in micro-electronics – I. MOS technology. *Solid-State Electronics*, 15:819–829, 1972.

[44] Y. Hsu and H. Li. Programmable variable-cycle clock circuit for skew-tolerant array processor architecture. U.S. Patent 4,851,995, July 1989.

[45] Integrated Device Technology, Inc., Santa Clara, CA. *High Performance CMOS Data Book*, 1988. IDT49C410 16-bit CMOS Microprogram Sequencer.

[46] Mark G. Johnson and Edwin L. Hudson. A variable delay line PLL for CPU-coprocessor synchronization. *IEEE Journal of Solid-State Circuits*, 23(5):1218–1223, October 1988.

[47] N. Jouppi et al. A 300MHz 115W 32b ECL Microprocessor. In *Proceedings of the 40th ISSCC.* IEEE, February 1993.

[48] Robert W. Keyes. *The Physics of VLSI Systems.* Addison-Wesley Publishing Company, Reading, MA, 1987.

[49] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1973 Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[50] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8, 1990.

[51] Kathleen Knobe and Natarajan Venkataraman. Data optimization: Minimizing residual interprocessor data motion on SIMD machines. In *Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation*, page 1, October 1990.

[52] Donald E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, Reading, MA, 1973.

[53] Toshio Kondo et al. Pseudo MIMD array processor – AAP2. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 330–337, June 1986.

[54] Lizyamma Kurian, Paul T. Hulina, and Lee D. Coraor. Memory latency effects in decoupled architectures with a single data memory module. In *Proceedings, 19th Annual Symposium on Computer Architecture*, pages 236–245, May 1992.

[55] Tom Leighton, Charles E. Leiserson, and Dina Kravets. *Theory of Parallel and VLSI Computation: Lecture Notes for 18.435/ 6.848.* Massachusetts Institute of Technology, Cambridge, MA, 1990.

[56] MasPar Computer Corporation. Delivering on the promise of massively parallel computing. Marketing Brochure, 1990.

[57] Scott McFarling. Program optimization for instruction caches. In *ASPLOS-III Proceedings*, pages 183–191. ACM, 1983.

[58] Carver A. Mead and Lynn A. Conway. *Introduction to VLSI Systems.* Addison-Wesley Publishing Company, Reading, MA, 1980.

[59] Motorola. *ECLinPS Data*, 1 edition, 1991.

[60] Eric Nee. Interview with John Rollwagen (CEO, Cray Research Inc). *Upside*, V(1):18–32, January 1993.

[61] M. Nemes. Driving large capacitances in MOS LSI systems. *IEEE Journal of Solid-State Circuits*, pages 159–161, 1984. fix this up.

[62] John R. Nickolls. The design of the MasPar MP-1: A cost effective massively parallel computer. In *Proceedings of IEEE Compcon Spring 1990.* IEEE, February 1990.

[63] O. Nishii et al. A 1,000MIPS BiCMOS microprocessor with superscalar architecture. In *Proceedings of the 39th ISSCC*, pages 114–115. IEEE, February 1992.

[64] A. Nowatzyk. *A Communication Architecture for Multiprocessor Networks.* PhD thesis, Carnegie Mellon University, 1989.

[65] David A. Patterson and Carlo H. Séquin. A VLSI RISC. *IEEE Computer*, pages 8–21, September 1982.

[66] Tekla S. Perry. Modeling the world's climate. *IEEE Spectrum*, 30(7):33–42, July 1993.

[67] Ulrich Schmidt, Knut Caesar, and Thomas Himmel. Data-driven array processor for video signal processing. *IEEE Transactions on Consumer Electronics*, 36(3):327–333, August 1990.

[68] Lorenz A. Schmitt and Stephen S. Wilson. The AIS-5000 parallel processor. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(3):320–330, May 1988.

[69] Charles L. Seitz. System timing. In *Introduction to VLSI Systems*, chapter 7, pages 218–262. Addison-Wesley Publishing Company, Reading, MA, 1980.

[70] Charles L. Seitz. Concurrent VLSI architectures. *IEEE Transactions on Computers*, 33(12):1247–1265, December 1984.

[71] Charles L. Seitz. Mosaic C: An experimental fine-grain multicomputer. Invited Address at 25th Anniversary of INRIA, December 1992. This paper should appear in the conference proceedings, but I only have a preprint.

[72] Alan C. Shaw. *The Logical Design of Operating Systems*. Prentice-Hall, Englewood Cliffs, N.J., 1974.

[73] David Shu, Lap-Wai Chow, and J. G. Nash. A content addressable, bit-serial associative processor. In Robert W. Brodersen and Howard S. Moscovitz, editors, *VLSI Signal Processing, III*, chapter 12, pages 120–128. IEEE Press, 1988.

[74] Daniel L. Slotnick, W. Carl Bork, and Robert C. McReynolds. The Solomon computer. In *Proceedings of the Fall Joint Computer Conference*, volume 22, pages 97–107. AFIPS, 1962.

[75] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[76] Ivan E. Sutherland and Carver A. Mead. Microelectronics and computer science. *Scientific American*, 237(3):210–228, September 1977.

[77] Ivan E. Sutherland and Robert F. Sproull. Logical effort: Designing for speed on the back of an envelope. In C.H.Sequin, editor, *Proceedings of the Advanced Research in VLSI Conference*. MIT Press, 1991.

[78] Garold S. Tjaden and Michael J. Flynn. Detection and parallel execution of independent instructions. *IEEE Transactions on Computers*, C-19(10):889–895, October 1970.

[79] Jeffrey D. Ullman. *Computational Aspects of VLSI*. Principles of Computer Science. Computer Science Press, Rockville, MD, 1984.

[80] D. M. H. Walker. Critical area analysis. In V. K. Jain and P. W. Wyatt, editors, *Proceedings, International Conference on Wafer Scale Integration*, pages 281–290, Los Alamitos, CA, January 1992. IEEE Computer Society Press.

[81] C. Weems, S. Levitan, and C. Foster. Titanic: A VLSI based content addressable parallel array processor. In *Proceedings of the 1982 International Conference on Circuits and Computers*. IEEE, September 1982.

[82] Charles C. Weems Jr. The content addressable array parallel processor: Architectural evaluation and enhancement. In *Proceedings of the International Conference on Computer Design: VLSI in Computers*, pages 500–503. IEEE, October 1985.

[83] Neil Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design*. VLSI Systems Series. Addison-Wesley, USA, 1985.

[84] Skef Wholey. *Automatic Data Mapping for Distributed-Memory Parallel Computers*. PhD thesis, Carnegie Mellon University, May 1991.

[85] M. V. Wilkes. Slave memories and dynamic storage allocation. *IEEE Transactions on Electronic Computers*, EC-14(2):270–271, April 1965.

[86] T. E. Williams, M. Horowitz, R. L. Alverson, and T. S. Yang. A self-timed chip for division. In *Advanced Research into VLSI: The 1987 Stanford Conference*, pages 75–95, 1987.

[87] T. H. Yeap, W. M. Loucks, W. M. Snelgrove, and S. G. Zaky. Implementing the VASTOR architecture using a VLSI array of 1-bit processors. In *Proceedings of the International Conference on Computer Design: VLSI in Computers*, pages 494–499. IEEE, October 1985.

[88] Ian Young, Jeffrey Greason, and Keng Wong. A PLL clock generator with 5 to 110 MHz of lock range for microprocessors. *IEEE Journal of Solid-State Circuits*, 27(11):1599–1607, November 1992.

[89] Jiren Yuan and Christer Svensson. High-speed CMOS circuit technique. *IEEE Journal of Solid-State Circuits*, 24(1):62–70, February 1989.

[90] Jiren Yuan and Christer Svensson. Pushing the limits of standard CMOS. *IEEE Spectrum*, 28(2):52–53, February 1991.