Improving Processor Performance and Simulation Methodology

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Joshua Jeffrey Yi

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

David J. Lilja, Advisor

December 2003

UNIVERSITY OF MINNESOTA

This is to certify that I have examined this copy of a doctoral dissertation by

Joshua Jeffrey Yi

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Name of Faculty Adviser(s)

Signature of Faculty Adviser(s)

Date

GRADUATE SCHOOL

Acknowledgements

As the end of my days as a graduate student draws nigh, I have the occasion to reflect upon the many long days, and the even longer nights, that all too quickly lengthened into years, that finally culminated with a doctorate in Electrical Engineering, *dei gratia*. What this reflection – tinged with nostalgia – has revealed is that the splendor of this achievement is possible only as a result of the support of others. And while my name shall be the only one on the diploma, these others deserve similar or equal recognition. Accordingly, I would like to avail myself of this opportunity to gratefully thank those people who unfailingly supported and comforted me when I was disheartened, gently challenged and questioned me when it was necessary, and, finally, loved me always. To you, I owe a debt that my clumsy eloquence cannot fully articulate and that my feeble efforts can never adequately remunerate.

I would first wish to acknowledge the infinite debt that I owe to my God and His Son, the Lord Jesus Christ, the sovereign Lord of my life. I also wish to thank God and Christ Jesus for His tender compassion and mercy, His everlasting love and goodness, His unmerited favor and blessings, and His perfect peace and golden happiness. Without any and all of these gifts, this dissertation would have been impossible.

> *Praise the Lord, O my soul, and forget not all His benefits – who forgives all your sins and heals all your diseases, who redeems your life from the pit and crowns you with love and compassion, who satisfies your desires with good things so that your youth is renewed like the eagle's.*
>
> **– Psalm 103:2-5**

I also am profoundly indebted to my advisor, Professor David Lilja, for his technical knowledge, his generous financial assistance through many of my years of graduate school, his unflagging encouragement, and especially his ample patience (as he had to listen to my complaints, answer my many questions, and endure my stubbornness). He also had faith in my abilities when others, often including myself, had doubts. Above these characteristics, however, the fundamental principles that I learned from him were professional integrity and gentlemanly deportment. My one professional aspiration is to, one day, be as fondly regarded by my students as he is by me, and to engender the same deep respect from my colleagues.

I would also like to recognize the invaluable help that was freely and generously given to me by Professor Douglas Hawkins, Professor Wei-Chung Hsu, and Professor Pen-Chung Yew. I would like to thank Professor Hawkins for all the statistical help that he gave to this statistical neophyte in the form of lengthy e-mail replies and through clear explanations of statistical principles. Professor Hsu, over the years, challenged me with pointed questions to achieve more than I already had and to further improve the quality and depth of my work. For these things, I am grateful. Finally, I would like to thank Professor Yew for his encouragement and counsel.

I would also like to thank Professor Sachin Sapatnekar and Professor George Karypis, two of my doctoral examination committee members, for their willingness to serve on my committee and for their insightful comments and criticisms.

I would like to acknowledge the technical advice, the all-too-often unrequited help, and, foremost, the friendship that was given to me by Ying Chen, Youngsoo Choi, Peng-fei Chuang, Bob Glamm, Chris Hescott, Baris Kazar, Sreekumar Kodakara, Dr. Jin Lin, Keith Osowski, Professor Resit Sendag, and Keqiang Wu. We laughed together, we “enjoyed life” together, and we grew up as engineers – and as people – together. Collectively, you inspired me, you challenged me, and you reassured me. Truly, our shared memories make parting such sweet sorrow.

And finally, to my family, I offer my most heartfelt and gracious thanks. To my parents, C. James Yi, Ph.D. and Judy Yi, and to my sisters, Jennifer Yi, Ph.D. and Joanna Yi, M.D.-to-be, the work encapsulated in this dissertation was possible only with your steadfast encouragement, your earnest prayers, and your loving comfort. You lifted me up when I was down, you brightened my path in the midst of stormy trials, and you sustained my confidence in times of doubt. For these things, and the countless others that are not written here, since any words that I may muster are starkly pale in comparison, I simply say, thank you, and thank you for loving me.

Joshua J. Yi
December 2003
Minneapolis, Minnesota

Soli Deo Gloria

Table of Contents

List of Figures

List of Tables

Abstract

Although current commercial processors are capable of fetching and executing multiple instructions per cycle, processor resources such as issue slots, functional units, and buffers are frequently idle due to the lack of available instruction-level parallelism. As a result, the processor's actual performance is often far below its theoretical maximum performance. To increase the amount of instruction-level parallelism, this dissertation proposes two microarchitectural techniques that dynamically remove redundant and trivial computations. A redundant computation is a computation that the processor performs repeatedly during the course of a program's execution while a trivial computation is one where the output is zero, one, 0xffffffff, or a shifted version of one of the inputs.

The first technique, Instruction Precomputation, compares each instruction's opcode and input operands against the opcode and input operands that are stored in the Precomputation Table. If the opcodes and input operands match, the Precomputation Table forwards the result for that redundant computation to the associated instruction. Our results show that the 2048 highest frequency redundant computations account for 14.68% to 44.49% of the total dynamic instruction count. Using Instruction Precomputation to dynamically remove these redundant computations yields speedups of

0.71% to 45.40%, with an average of 10.53%, when using a 2048-entry Precomputation Table.

The second technique, the Simplification and Elimination of Trivial Computations, checks the opcode and input operands of each instruction to determine whether or not that computation is trivial or not. When the trivial computation can be simplified, the instruction is converted to another type of instruction that produces the same result, but with a lower execution frequency. When the trivial computation can be eliminated, the trivial computation hardware "computes" its result and removes the instruction from the pipeline. Our results show that 12.24% and 5.73% of all dynamic instructions in selected SPEC 2000 and MediaBench, respectively, benchmarks are trivial computations. Adding hardware to exploit these trivial computations yields speedups of 1.31% to 27.36%, with an average of 8.86%, for the SPEC 2000 benchmarks and speedups of 2.97% to 13.97%, with an average of 4.00%, for the MediaBench benchmarks.

Finally, due to cost, time, and flexibility constraints, simulators are used in the design and implementation of next-generation processors and to evaluate the performance of processor enhancements. Despite this dependence on simulators, computer architects usually approach the simulation process in an ad-hoc manner. Mistakes and irregularities in the simulation process may introduce errors into the simulation results. On the other hand, using statistically-based simulation methodology helps the architect decrease the number of errors in the simulation process, gives more insight into the effect of a processor design or enhancement, and provides statistical support to the observed behavior. This dissertation proposes and demonstrates the efficacy of using the statistical Plackett and Burman design to improve how processor parameter values are chosen, how benchmarks are chosen, and how processor enhancements are analyzed. In particular, the results show the effect that Instruction Precomputation and exploiting trivial computations have on the processor.

This dissertation makes the following primary contributions. First, this dissertation quantifies the amount of redundant and trivial computations that are present in typical programs. Second, to exploit these two program characteristics, this dissertation proposes and demonstrates the performance potential of two microarchitectural enhancements: Instruction Precomputation and Simplifying and Eliminating Trivial Computations. Finally, this dissertation identifies problems with existing simulation methodologies and offers specific, statistically-based recommendations to improve the overall quality of simulation methodology.

Chapter 1

Introduction

In the past few years, superscalar processors have become the most popular processor architecture due to their high-performance. Examples of superscalar processors include the Alpha 21264 [Kessler98, Kessler99, Leiholz97], the MIPS R10000 [Yeager96], the UltraSparc III [Horel99], and the PA-RISC 8000 [Kumar97]. To achieve even more performance, processor designers can increase the clock frequency and/or increase the number of instructions that the processor decodes, issues, executes, and retires per clock cycle.

Since higher clock frequencies do not yield proportional increases in the processor's performance and since the maximum clock frequency is constrained by the minimum transistor width, processor designers primarily try to improve the processor's performance by maximizing the number of retired instructions per cycle. While current-generation processors are capable of decoding, issuing, executing, and retiring several instructions in a single cycle, only independent instructions can be issued in the same cycle. One measure of how independent the instructions are for a given region of a program is the amount of instruction-level parallelism (ILP). The next two sections briefly discuss the key components of superscalar processors and the effect that dependences between instructions have on the ILP.

1.1. Superscalar Microprocessors

As specified by the von Neumann model [Hennessy96], the two main components of modern processors are the execution core and memory. Caches, the memory structures closest to the processor (in terms of access time), are used to hold recently accessed data and instructions. To improve the performance of retrieving data and instructions, the L1 Caches – the caches closest to the processor – are split into separate caches, one for data and one for instructions. This model of cache partitioning is called the Harvard Architecture [Hennessy96].

The L1 Instruction Cache (or I-Cache), the L1 Data Cache (D-Cache), and the L2 Cache buffer instructions and data that will likely be needed in the near future. When using these caches, instead of retrieving the requested blocks from main memory – which requires hundreds of cycles – the requested blocks can be retrieved from a cache. Consequently, reducing the access time for those memory blocks reduces the overall program execution time.

The purpose of the execution core is to execute the assembly-level instructions of the program. Its main components are: the instruction fetch logic, the branch predictor and target buffer, the decode logic, the register file, the reorder buffer (ROB), the issue logic, and the functional units. The instruction fetch logic uses predictions and target addresses from the branch predictor and target buffer, respectively, to efficiently retrieve instructions from memory and store them into the instruction fetch queue (IFQ). The decode logic decodes instructions in the IFQ, retrieves their input operands from the register file or ROB, and moves the decoded instructions into the ROB. In addition to buffering the current state of each in-flight instruction, the ROB stores and also retires these instructions in program order to support precise exceptions. The issue logic determines which instructions are ready to execute – the ones that have all of their input operand values – and sends them to the functional units. The functional units compute the results of arithmetic and logical instructions, store or load data values to and from the memory hierarchy, and determine the branch direction and target.

Figure 1.1.1 shows the functional-level organization of the typical superscalar processor. To improve the readability of the figure, the branch target buffer is represented by the branch predictor. For the same reason, the L2 Cache and main memory are omitted.

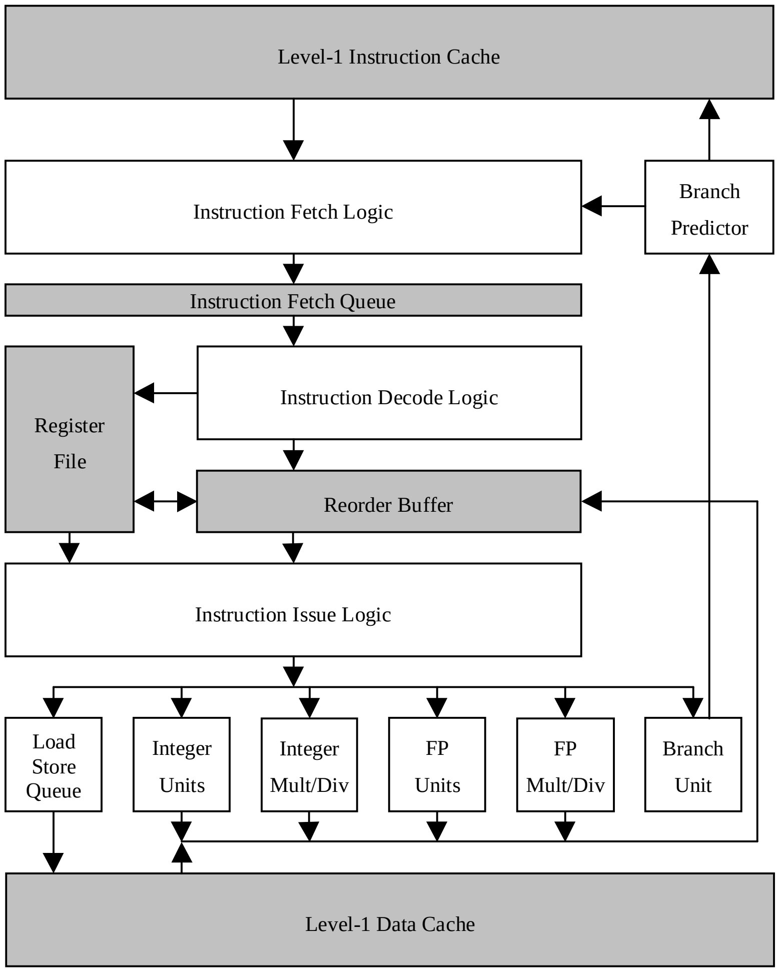

Figure 1.1.1: Functional-Level Organization of a Superscalar Processor

The hallmark of superscalar processors is that they fetch, decode, execute, and retire multiple instructions per cycle. Consequently, the fetch logic, the decode logic, and the functional units operate on several instructions every cycle.

However, due to branch instructions – or control dependences, the average number of fetched instructions is often far below the fetch width (maximum number of instructions that can be fetched per cycle). By lowering the average number of instructions that are fetched per cycle, control dependences also reduce the number of instructions that are being decoded, executed, or retired in any given cycle. Branch predictors, branch target buffers, and instruction fetch queues minimize, but cannot completely eliminate, the effect of control dependences.

1.2. Instruction-Level Parallelism and Data Dependences

In addition to control dependences, data dependences also reduce the actual performance from the theoretical peak performance. A read-after-write (RAW) data dependence exists between two instructions if the result of the first instruction is used to calculate the result of the second. For example, consider the assembly code shown in Figure 1.2.1.

```
loop:     lw   r1,r2        ; r1 = Mem[r2]
          add  r1,r2,r3     ; r1 = r2 + r3
          sub  r4,r1,r5     ; r4 = r1 – r5
          mul  r1,r2,r3     ; r1 = r2 * r3
          sra  r2,r1,1      ; r2 = r1 >> 1
          beq  r7,r1,loop   ; Branch if r7 = r1
```

Figure 1.2.1: Assembly Code Example of Read-After-Write Dependences

In Figure 1.2.1, RAW data dependences exist between the `add` and `sub` instructions as well as between the `mul` and `sra` instructions. In the former pair, the `add` computes and then stores a value into register `r1`, which the `sub` instruction uses as an input value. In the latter pair, the `mul` instruction stores its result in register `r1`, which is subsequently read by the following instruction (`sra`). For each pair of the instructions, the second instruction cannot execute before the first since it needs the result of the first.

Consequently, a RAW data dependence exists between the first and second instructions of each pair through register `r1`.

Figure 1.2.1 also shows two other types of data dependences: write-after-write (WAW) and write-after-read (WAR). A WAW data dependence occurs when two instructions write to the same register value. For example, in Figure 1.2.1, WAW data dependences exist between the `lw` and `add` instructions as well as between the `add` and `mul` instructions, both through register `r1`. For each pair of instructions, the second instruction cannot write its output value to the shared register until after the first instruction has written its output value.

The WAR data dependence exists between two instructions if the second instruction writes to a register that the first reads from. In Figure 1.2.1, WAR dependences exist between the `sub` and `mul` instructions through register `r1` as well as between the `mul` and the `sra` instructions through register `r2`. Due to this dependence, the second instruction cannot write its output value to the shared register until after the first has read the register.

Since data dependences exist between instructions, the processor cannot issue or execute those instructions completely in parallel. This has two important ramifications. First of all, since processors are designed with enough resources to issue and execute multiple instructions in parallel, processor resources are idle when executing instructions with data dependences. Second, the serial execution of these instructions reduces the processor's performance from its theoretical maximum. Therefore, to improve the processor's resource utilization efficiency and, more importantly, to improve its actual overall performance, it is imperative to decrease the number of dependent instructions.

In addition to control and data dependences, the other major factor that limits the amount of ILP is the execution latency of instructions, or more specifically, the execution latency of load instructions. While many instructions have multi-cycle execution latencies, load instructions latencies have a disproportionately large effect on the performance since they have a very wide range of latencies, despite using multi-level data caches and other microarchitectural techniques such as out-of-order execution and prefetching. Instructions with very long execution latencies limit the amount of ILP by reducing the rate at which instructions are executed by the processor.

In summary, the number of instructions that can be issued and executed in parallel is primarily limited by the amount of ILP. Control and data dependences, in addition to long and variable instruction latencies, have a very large, negative effect on the processor's performance. Therefore, to improve the processor's performance, computer architects try to break the control and data dependences or reduce the effective instruction latency. The next few sub-sections introduce two techniques – **Instruction Precomputation** and the **Simplification and Elimination of Trivial Computations** – that attempt to break data dependences and reduce the execution latency.

1.3. Superscalar Performance

There are many different ways to measure a processor's performance, such as speedup, SPEC number, millions of instructions per second (MIPS), etc. [Lilja00], but one of the more meaningful metrics – and perhaps the most important – is the total execution time of the program on that particular processor. The total amount of time it takes to execute a program is approximated by Equation 1.3.1.

$$T_e = n * CPI * T_c$$

Equation 1.3.1: Formula for the Total Program Execution Time

Where

n	Total number of executed instructions
CPI	Average number of cycles needed to execute an instruction
T_c	Time per clock period (cycle)

Note that the reciprocal of CPI (cycles per instruction) is IPC (instructions per cycle).

Equation 1.3.1 shows that the execution time T_e is proportional to n, CPI, and T_c. Consequently, to reduce the program's total execution time, it is necessary to either: 1) Reduce the number of executed instructions, 2) Decrease the average execution time of each instruction, and/or 3) Decrease the clock period. However, since the number of instructions in the program cannot be reduced at run-time (i.e. by hardware) and since clock period is based on the minimum transistor width, the only viable option for computer architects to reduce the program's execution time is to reduce the CPI.

1.4. Redundant Computations

During the course of a program's execution, a processor executes many redundant computations. A redundant computation is a computation that the processor had performed earlier in the program. For example, consider the code fragment shown in Figure 1.4.1:

```
for (i=0; i < MAX; i++)
{
  for (j=0; j < MAX; j++)
  {
    …
  }
```

Figure 1.4.1: Example of a Source of Redundant Computations

For each iteration of the outer loop, the calculations for the loop index variable *j* are exactly the same. More specifically, the calculations for *j* are: 0+1, 1+1, 2+1, … , 98+1, 99+1, 0+1, 1+1, 2+1, … , 98+1, 99+1, 0+1, etc. Therefore, after the first iteration of the outer loop, all the computations to compute each value of *j* are redundant.

It is important to note that redundant computations are not limited only to add instructions. Rather, any and all computations can be redundant. Furthermore, it is also important to note that an optimizing compiler may not be able to remove these redundant computations during the compilation process since the actual input operand values may be unknown at compile time – possibly because they depend on the inputs to the program.

Redundant computations can be divided into global and local level redundant computations. The difference between the two is that the global-level redundant computations are independent of the Program Counter (PC) while local-level redundant computations are dependent on the PC. Figure 1.4.2 illustrates the difference between the two levels.

```
add r1,r1,#1  ; r1 = r1 + 1
              ; 0+1, 1+1, 2+1, 3+1, 4+1, … , 98+1, 99+1

...

add r2,r3,r4  ; r2 = r3 + r4
              ; 1+0, 1+1, 1+2, 1+3, 1+4, … , 1+98, 1+99
```

Figure 1.4.2: Example of Global and Local Level Redundant Computations

In this example, there are no local-level redundant computations since either instruction does not, itself, repeat a computation. However, at the global-level, there are 100 redundant computations since both instructions perform the same operations on the same input operand values. From this example, it is obviously that there are more redundant computations at the global-level, which is an important distinction when trying to improve the processor's performance by exploiting redundant computations.

Redundant computations affect the program's execution time in two ways. First of all, executing the instructions for redundant computations increases the program's dynamic instruction count. Secondly, these redundant computations affect the average CPI since they produce values for other instructions in the program. However, these redundant computations need to be executed to ensure correct program operation. Additionally, all of the instances of these instructions may not be redundant computations. Consequently, the hardware cannot simply disregard these computations at run-time to decrease the program’s execution time. Thus, the only recourse to improve the processor’s performance, with respect to redundant computations, is to reduce the CPI of the redundant computations.

1.5. Value Reuse: Effect and Shortcomings

Value reuse [Sodani97, Sodani98] is a microarchitectural technique that improves the processor’s performance by dynamically removing redundant computations from the processor’s pipeline. During the program’s execution, the value reuse hardware compares the opcode and input operand values of the current instruction against the opcodes and input operand values (hereafter, alternatively referred to as a **unique computation**) of all recently executed instructions, which are stored in the value reuse table (VRT). If there is match between the opcodes and input operand values, then the current instruction is a redundant computation and, instead of continuing its execution, the current instruction gets its output value from the result stored in the VRT. On the other hand, if the current instruction’s opcode and input operand values do not match those found in the value reuse table, then the instruction is not a recent redundant computation and it executes normally. After finishing execution, the value reuse hardware stores the opcode, input operand values, and output value for that instruction into the VRT.

While value reuse increases the amount of ILP – thus improving the processor’s performance – it does not necessarily effectively target the redundant computations that have the most effect on the program’s execution time. This shortcoming stems from the fact that the VRT is finite in size. Consequently, when all entries of the VRT are occupied and when the current instruction tries to store its unique computation and output value into the VRT, the contents of one entry are overwritten. However, if the current unique computation is executed less often than that of the one it replaces, then the current unique computation is less useful in increasing the amount of ILP since it occurs less often. In other words, due to dynamic replacement, the entries in the VRT are not necessarily the unique computations that have the greatest impact on the total program execution time.

To address this problem, this dissertation proposes a microarchitectural technique called **Instruction Precomputation**.

1.6. Instruction Precomputation

As described in the previous sub-section, one problem with value reuse is that a unique computation with a low frequency of execution could replace a high frequency one. However, since it is virtually impossible for hardware to determine at run-time which unique computations have the highest frequencies of execution, Instruction Precomputation uses feedback-directed optimization to first determine, at compile-time, the highest frequency unique computations. At run-time, those unique computations are then loaded into the Precomputation Table (PT), which is very similar to the VRT with the key exception that entries in the PT are not updated during the program’s execution. In other words, in value reuse, the hardware determines which unique computations should be in the VRT while in Instruction Precomputation, the compiler controls which unique computations are in the PT.

Using the compiler to statically determine which unique computations have the highest frequencies has two key advantages over using hardware. First, and most importantly, the compiler is able to determine if a unique computation is a high frequency unique computation or not. Although the compiler only profiles the program with one or two different inputs, previous work showed that the same program with different inputs sets had many high frequency unique computations in common. In other words, determining the highest frequency unique computations of a program with a specific input will most likely yield the highest frequency unique computations for the same program, but with a different input. Second, determining the highest frequency unique computations with the compiler means that fewer access ports are needed to access the PT. When the hardware is used to profile a program, additional access ports are needed to allow the hardware to write to the table. However, when using the compiler, the PT is never updated. Therefore, no additional write ports are needed. Having fewer access ports results in a lower PT access time, which means that Instruction Precomputation has a lower impact on the clock period.

1.7. Simplification and Elimination of Trivial Computations

In addition to repeatedly performing many redundant computations during the course of the program's execution, the processor also executes many trivial computations. A trivial computation can be defined as a computation where the output value is zero, one, or a shifted version of one of the input operands. For example, using definitions given in this dissertation, each of the following computations are trivial: X+0, X–X, X*2, and X/1. Since these particular computations are trivial – or more precisely, their output value is trivial – the processor can reduce the execution latency of these computations either by "calculating" the output value without using a functional unit or by simplifying the computation so it can use a functional unit with a lower execution latency. The remainder of this sub-section describes in more detail how trivial computations can be simplified or eliminated to improve the processor's performance.

As in the case of redundant computations, it is important to note that an optimizing compiler may not be able to remove these trivial computations since the actual input operand values may be unknown at compile time. As a result, removing or optimizing the execution of these trivial computations is best left to the hardware.

Trivial computations affect the program's execution time in the same two ways as do redundant computations. First of all, performing trivial computations increases the program's dynamic instruction count. Second, performing these trivial computations increases the overall, average CPI since these instructions produce input values for other instructions in the program. However, since these trivial computations need to be performed for correct program execution, to minimize their effect on the processor's performance, the only recourse is to reduce their latency.

As described in the previous sub-section, there are two methods to reduce the CPI of trivial computations. In the first method, the processor eliminates the trivial computation completely by computing the final result of the instruction without a functional unit. In the second method, the processor reduces the complexity, and consequently the execution latency, of the trivial computation by converting the operation into another operation.

For example, consider the following computations: X-X and X*4, where X is the value of one the input operands. Since the result of the first computation will be zero, regardless of the value of X, it is pointless to perform that computation using a functional unit since the result is trivial. In this particular case, by assigning the value of zero to that computation, the trivial computation hardware can reduce the execution latency of this instruction.

The result of the second computation is simply the value of X shifted to the left by two bit positions. Therefore, instead of performing this computation normally by using an integer multiply unit, the processor can perform this computation by converting this computation to a shift-left operation. And since shifts have a lower latency as compared to multiplies, dynamically changing the instruction from a multiply instruction to a shift-left instruction will decrease the CPI for this instruction, thus decreasing the overall program execution time.

In addition to reducing the execution latency of trivial computations, eliminating these trivial computations also has another key benefit: non-speculative early instruction execution. For most superscalar processors, an instruction is issued (sent to the functional units) only after it has been decoded and after it has received the values for both its input operands. However, for the trivial computations that can be eliminated, if `the instruction has received the value for the trivial input (e.g. 0 for X*0), the instruction does not need to wait for the other input operand value to arrive since the output can be computed purely as a result of the trivial one. Consequently, by exploiting these trivial computations in this way, the processor can exceed the dataflow limit (the maximum amount of ILP when given an infinite amount of hardware) non-speculatively.

The key difference between non-speculative and speculative instruction execution is that the output value of the latter is computed based on predicting what its input values might be, executing it with those input values (generating a speculative output value), and then executing any dependent instructions with that speculative output value. Therefore, before that instruction can be committed, the processor needs to verify if the input values were correctly predicted. If so, the output value of that instruction and any dependent instructions can be written to the register file. If not, then the processor needs to re-execute all dependent instructions that used the incorrect value. For those instructions that can be executed non-speculatively, the processor does not need to check if the prediction is correct (since no prediction was made) and can immediately commit its output value.

1.8. The Efficacy, Utility, and Necessity of Simulators

Simulators are the most important tool in computer architecture research. Due to cost, time, and flexibility constraints, simulators are often used to explore the design space when developing a new processor architecture or to evaluate the effectiveness of a proposed processor enhancement (hardware or software). For instance, simulators reduce the cost and development time of a new processor design by giving the architecture design team ballpark estimates of the processor's performance. Without simulators in this case, the team would have to use intuition or actually fabricate the chip to evaluate the performance of each candidate design. Consequently, without simulators, designing processors either would be too expensive or would yield very poor designs.

1.9. Deficiencies of Existing Simulation Methodologies

Despite this level of dependence on simulators, computer architects usually approach the simulation process in an ad-hoc manner. Consequently, the results that computer architects obtain from their simulations may be not completely accurate, or worse yet, may be misleading. Furthermore, an ad-hoc simulation methodology does not necessarily extract the maximum amount of information from the results.

For example, a sensitivity analysis is frequently utilized to determine the effect that different processor parameters have on a processor enhancement. To test the effect of each parameter, the computer architect will vary one or more parameters at a time while holding the other parameters at a constant value and measure the effect of the variable parameter(s) on the processor enhancement. However, before starting the simulations, several questions about sensitivity analysis itself need to be answered. For example, which parameters should be varied? What range of values should be used for those parameters? Do any of the constant parameters interact with the variable ones? What is the magnitude of those interactions? How much impact do the specific values of the constant parameters have?

Furthermore, in addition to the questions about the sensitivity analysis setup, other questions regarding the simulation setup need to be answered. For example, which benchmarks (i.e. programs) should be used in the sensitivity analysis? Which input set (to the benchmark) should be used? If the input set is relatively short, what effects does its length have as compared to a longer input set? How should those effects be mitigated?

These questions and more are the type of questions that need to be answered before starting the simulations. However, due to the sheer computational cost, it is virtually impossible to simulate all possible combinations of parameters or to fully answer all of the simulation setup questions. This situation illustrates the need for a statistically-based simulation methodology.

While the downside of using such a methodology is that it may require some additional simulations, it also has the following advantages:

1) It decreases the number of errors that are present in the simulation process and helps the computer architect detect errors more quickly. Errors include, but are not limited to, simulator modeling errors, user implementation errors, and simulation setup errors [Black98, Cain02, Desikan01, Gibson00, Glamm00].
2) It gives more insight into what is occurring inside the processor or the actual effect that a processor enhancement has on the processor.
3) It gives objective confidence to the results and provides statistical support regarding the observed behavior.

While the first and third advantages are self-explanatory, it is not obvious from the second advantage how a statistically-based methodology could improve the quality of the analysis. Since simulators are complex, it is very difficult to fully understand the effect that a design change or an enhancement may have on the processor. As a result, architects use high-level single-value metrics, such as speedup or cache miss rate, to understand the "big-picture" effects. Unfortunately for this approach, these high-level metrics sacrifice information for conciseness by discarding most of the information that is available to the simulator for a single, neat assessment of the performance. Consequently, important conclusions that are more subtle are overlooked. Furthermore, since much information is discarded, only the net effect on the final metric of two competing effects is known.

For example, suppose a new prefetching mechanism improves the processor's performance by 20%. Also suppose that in this case that this prefetching mechanism decreases the importance of the cache's associativity, but dramatically magnifies any shortage in the number of load-store queue (LSQ) entries. While this may be the case, the speedup only shows the net result of the associativity's decreased effect and the increased effect of the LSQ entries. As a result, while the overall speedup is quite good, further analysis would reveal that this prefetching mechanism also moves the performance bottleneck from the cache associativity to the LSQ entries.

Therefore, basing conclusions on a single high-level metric can be dangerous since that metric shows the "big picture" only at a distance. However, analyzing the processor from a statistical point-of-view can help the architect quantify the effects that all components have on the performance and on other important design metrics (e.g. power consumption, etc.).

More specifically, this dissertation improves the simulation methodology used by computer architects by recommending specific procedures on how to:

1) Choose the processor parameter values.
2) Select a sub-set of benchmarks.
3) Analyze the effect that an enhancement has on the processor.

The first two recommendations target the simulation setup phase of the simulation process while the last recommendation targets the analysis phase.

To illustrate the efficacy and utility of using a statistically-rigorous simulation methodology, this dissertation uses this simulation methodology when evaluating the performance of Instruction Precomputation and the Simplification and Elimination of Trivial Computations, which is described in Section 6.3.3.

1.10. Contributions of this Dissertation

This Ph.D. dissertation makes the following contributions:

1) This dissertation quantifies the amount of redundant computations at the global-level (PC-independent) and at the local-level (PC-dependent).
2) This dissertation proposes a feedback-directed optimization and hardware-based processor enhancement called Instruction Precomputation that yields speedups of 4.47% and 10.52% for a small and a large PT, respectively.
3) This dissertation defines the range of and quantifies the amount of trivial computations.
4) This dissertation proposes a set of hardware mechanisms that improve the processor's performance by Simplifying and Eliminating Trivial Computations and by using a novel non-speculative scheduling mechanism. This solution improves the processor's performance by 8.22% for a typical processor and by 6.5% for an aggressive processor.
5) This dissertation makes specific recommendations on how to improve the simulation methodology used by computer architects. Collectively, these recommendations can improve the overall quality of the simulation methodology, decrease the total number of simulations, quickly determine the processor's bottlenecks, and provide analytical insights into the impact of processor enhancements, as compared to when no rigorous simulation methodology is used.

1.11. Dissertation Organization

The remainder of dissertation is organized as follows: Chapters 2 and 3 describe Instruction Precomputation and the Simplification and Elimination of Trivial Computations in more detail while Chapter 4 does the same for statistically-based simulation methodology. Chapter 5 describes the simulator, benchmarks, and input sets that were used while Chapter 6 describes performance results for Instruction Precomputation and by exploiting trivial computations. In addition, Chapter 6 also illustrates how statistically-rigorous simulation methodology can improve the simulation quality and analysis. Chapter 7 discusses previous work related to Instruction Precomputation, exploiting trivial computations, and simulation methodology. Finally, Chapter 8 describes the future work and Chapter 9 concludes.

Chapter 2

Instruction Precomputation

As described in Chapter 1, Instruction Precomputation is a microarchitectural technique that improves the processor's performance. The remainder of this chapter describes the problem that Instruction Precomputation attempts to solve, the program characteristic that it exploits, how it operates, and what hardware and compiler additions it needs.

2.1. Problems with Existing Value Reuse Mechanisms

As described in Chapter 1, and in more depth in Chapter 7, value reuse [Sodani97, Molina99] is a hardware-based technique that dynamically removes instructions that are redundant computations by forwarding the results of those computations from the value reuse table (VRT) to that instruction. The VRT is an on-chip table that caches the input operands and the results of previously executed computations. The processor uses the program counter (PC) value for each instruction to access the VRT. An example of a four-entry VRT is shown in Figure 2.1.1. To access the VRT, the processor uses the instruction's PC and input operands (labeled Input #1 and Input #2) to the VRT. If the PCs and input operands match, the VRT sends the output value at that entry back to the processor and that instruction is removed from the pipeline.

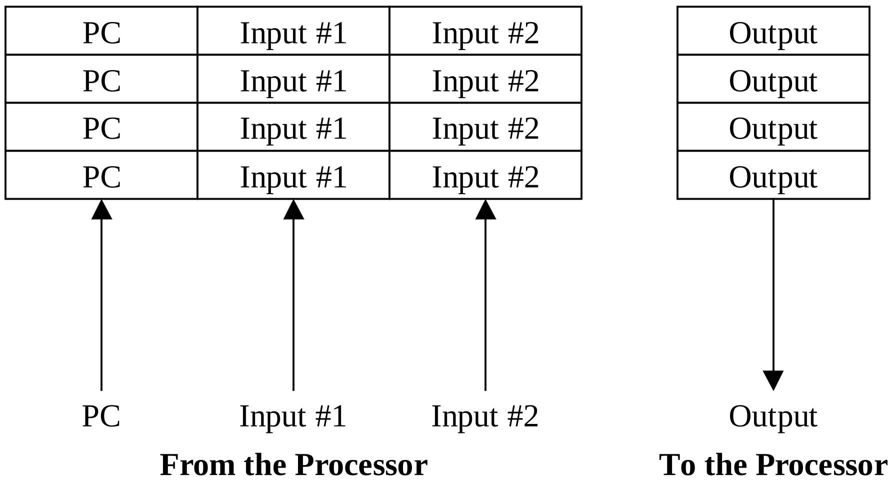

Figure 2.1.1: Four-Entry Value Reuse Table and its Processor Interface

Value reuse improves the processor's performance by decreasing the execution latency of each reused instruction and by decreasing the number of resource conflicts in the issue and execute stages of the pipeline. Decreasing the latency of a reused instruction either directly or indirectly reduces the execution time of the critical path; directly if the reused instruction is on the critical path or indirectly if the reused instruction produces the value of an input operand for an instruction that is on the critical path. Furthermore, since the reused instruction does not pass through the remaining pipeline stages, the number of resource conflicts (available issue slots, functional units, reservation station entries, etc.) decreases.

While value reuse can improve the processor's performance, two problems limit its effectiveness. First of all, since the PC is used to index the VRT, value reuse can only reuse the computations associated with each static instruction. Consequently, previous computations can only be reused if that computation has already been performed for the instruction associated with that particular PC. As a result, while another instruction of that type, but with a different PC, may have previously performed that redundant computation, the result of that computation cannot be reused since the results of the second instruction cannot be accessed by the first instruction. In other words, value reuse can be exploited only if that static instruction had previously performed that computation, even though another instruction may have performed the identical computation.

Second, since the VRT is dynamically updated during the course of the program's execution, low frequency redundant computations could eventually fill a significant percentage of the VRT's entries. Replacing a high frequency redundant computation with a low frequency one reduces the number of instructions that can reuse that computation.

Overall, the net effect of these two problems is that value reuse can be rather inefficient by not reusing previously executed computations and then evicting high frequency computations in favor of lower ones. To address these two problems, and thus improve the performance of value reuse, Instruction Precomputation uses the compiler to determine the highest frequency, PC-independent redundant computations and then does not allow those high frequency redundant computations to be replaced at run-time.

2.2. The Amount of Redundant Computations

There are two types of redundant computations. Local-level redundant computations are redundant computations that are associated with a single PC value (i.e. PC-dependent) while global-level redundant computations are PC-independent. For example, 0+0 with a PC value of 0x8000 and 0+0 with a PC value of 0x8004 are two different local-level redundant computations while they are the same global-level redundant computation. Given these two definitions, the key question is: Which definition accounts for the highest percentage of dynamic instructions, i.e. which definition affects the larger percentage of the program's instructions?

To determine the amount of redundant computation at both levels, the opcode, input operands, and PC for all the dynamic instructions have to be stored. To reduce the memory requirements for storing this information, in addition to storing the unique computation itself, the total number of times that that unique computation was executed was also stored. Recall that a unique computation is composed of the opcode and input operand values. The instruction's output value was not stored because it is a deterministic function of the opcode and input operands values.

To determine the amount of global-level redundant computation, each unique computation's PC was set to 0. As a result, unique computations that have the same opcode and input operands, but different PCs, map to the same unique computation. Meanwhile, at the local-level, the unique computation's PC was simply the instruction's PC.

To gather this information, a modified version of *sim-fast* from the SimpleScalar tool suite [Burger97] was used. *sim-fast* is a functional simulator that it is optimized for simulation speed. Consequently, it does not measure the execution time; it executes instructions serially; and it does not model a processor's pipeline, caches, etc. However, this simulator is adequate to determine the amount of global and local level redundant computation since the execution time, cache behavior, etc. are irrelevant when determining the amount of redundant computation.

Table 2.2.1: Example Unique Computations

PC	Unique Computation	Frequency
0x1000	0+1	400
0x1000	0+9	350
0x1000	1+1	500
0x1000	1+2	450
0x1000	1+3	500
0x1000	1+4	450
0x1000	1+5	450
0x1000	1+6	450
0x1000	1+7	550

The term, frequency of repetition, appears in the following paragraphs. The frequency of repetition, or frequency, is the number of times that a unique computation occurs (i.e. the number of dynamic instructions associated with that particular unique computation) in the program. Therefore, a unique computation is completely unique if it has a frequency of repetition of 1. On the other hand, unique computations with a frequency greater than one are redundant.

To illustrate how this term is used, Table 2.2.1 shows the computational history for the static instruction 0x1000.

In the program, for this PC, the computation 0+9 occurs 350 times; 0+1 400 times; 1+2, 1+4, 1+5, and 1+6 450 times each; 1+1 and 1+3 500 times each; and 1+7 550 times. The number of times that each computation occurs in the program is its frequency.

2.2.1. Global-Level Redundant Computations

Figure 2.2.1.1 shows the frequency distribution of the unique computations for selected benchmarks from the SPEC 2000 benchmark suite, using logarithmic frequency ranges. The second column in Table 5.4.1 shows the specific input sets that were used for results in this figure. After trying several different frequency range sizes, the logarithmic range size was used since it produced the most compact results without affecting the content.

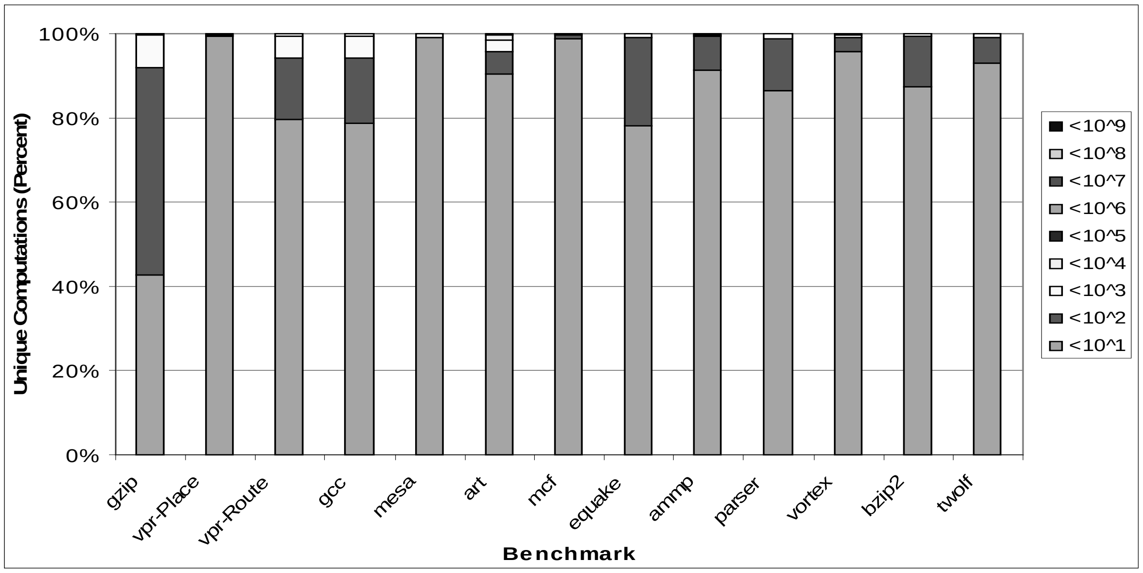

Figure 2.2.1.1: Frequency Distribution of Unique Computations per Benchmark, Global-Level, Normalized

In Figure 2.2.1.1, the height of each bar corresponds to the percentage of unique computations that have a frequency of execution within that frequency range. For example, if the unique computation 10004+11442, PC = 0x1000 executes 84 times, then it falls into the < 10^2 frequency range.

As can be seen in Figure 2.2.1.1, almost 80% of all unique computations have execution frequencies less than 10 (with the exception of *gzip*), while over 90% of all unique computations have execution frequencies less than 100. This result shows that most unique computations occur relatively infrequently in a program. Consequently, the performance benefit in reusing most of the unique computations is relatively low since most of them are only executed a few times.

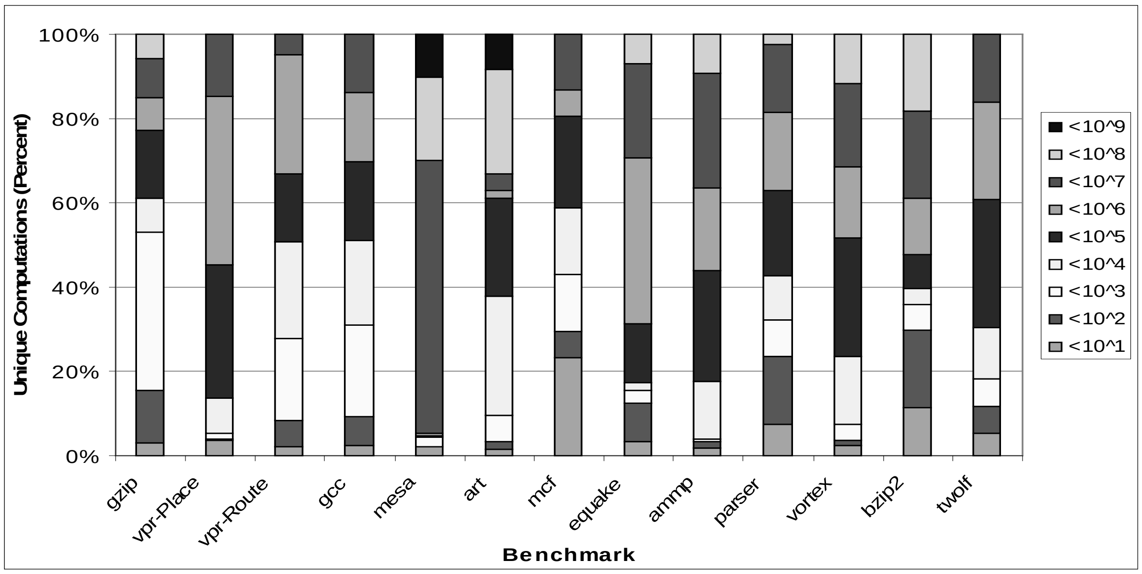

Figure 2.2.1.2: Percentage of Dynamic Instructions Due to the Unique Computations in each Frequency Range, Global-Level, Normalized

A unique computation's frequency of execution corresponds to the number of dynamic instructions that that unique computation represents. For example, if a unique computation has a frequency of execution of 2000, 2000 dynamic instructions perform that specific computation. Similarly so, if three unique computations each have a frequency of 500,000, then those unique computations are executed a total of 1,500,000 times, which corresponds to 1,500,000 dynamic instructions. Figure 2.2.1.2 shows the percentage of dynamic instructions due to the unique computations in each frequency range.

In Figure 2.2.1.2, the height of each bar corresponds to the percentage of dynamic instructions that have their unique computation in that frequency range. For each frequency range, comparing the heights of the bars in Figures 2.2.1.1 and 2.2.1.2 shows the relationship between the unique computations and dynamic instructions. For example, in *vpr-Place*, more than 99% of all unique computations represent only 3.66% of all dynamic instructions.

More than 90% of the unique computations account for only 2.29% (*mesa*) to 29.66% (*bzip2*) of the total number of instructions. Another way of stating this result is that a very large percentage of the unique computations account for a disproportionately small percentage of the total number of instructions. On the other hand, a program executes a small set of unique computations a very large number of times. This is one of the key results of this dissertation.

While a very small percentage of unique computations may account for a very large percentage of instructions, if a program has billions of unique computations, all of these high frequency unique computations may not fit into a reasonably sized on-chip table. Putting it another way, the number of unique computations that can fit into a reasonably sized on-chip table may not account for a significant percentage of the total instructions. Therefore, Table 2.2.1.1 shows the percentage of dynamic instructions that are represented by less than 2048 unique computations.

Table 2.2.1.1 shows that the top 2048 unique computations by frequency of execution, which account for a very small percentage of the total unique computations (0.002% - 0.162%), represent a significant percentage of the total dynamic instructions (14.68% - 44.49%). The conclusion from these results is that the highest frequency unique computations that can fit into a reasonably sized on-chip table still account for a significant percentage of the dynamic instructions. In other words, Table 2.2.1.1 shows that Instruction Precomputation has the potential of significantly improving the processor's performance since it targets a large percentage of the program's instructions.

Table 2.2.1.1: Characteristics of the Unique Computations for the Top 2048 Global-Level Unique Computations, by Frequency

Benchmark	% of Unique Computations	% of Total Instructions
gzip	0.024	14.68
vpr-Place	0.029	40.57
vpr-Route	0.162	23.44
gcc	0.032	26.25
mesa	0.010	44.49
art	0.010	20.24
mcf	0.005	19.04
equake	0.017	37.87
ammp	0.079	23.93
parser	0.010	22.86
vortex	0.033	25.24
bzip2	0.002	26.83
twolf	0.026	23.54

2.2.2. A Comparison of the Amount of Global and Local Level Redundant Computation

While the previous section showed that there is a significant amount of redundant computation available at the global-level, the key question is: how much more redundant computation is available (and can be exploited) at the global-level as compared to the local-level? It is important to note that, for the same number of unique computations, the global-level unique computations will account for a higher percentage of the total dynamic instruction as compared to the same number of local-level unique computations. However, since the global-level unique computations may not represent significantly more instructions, it is worthwhile to determine how many more instructions the highest frequency global-level unique computations represent before implementing Instruction Precomputation. As a result, this section compares the global and local level results for the percentage of instructions: 1) In each frequency range and 2) Represented by the top 2048 unique computations.

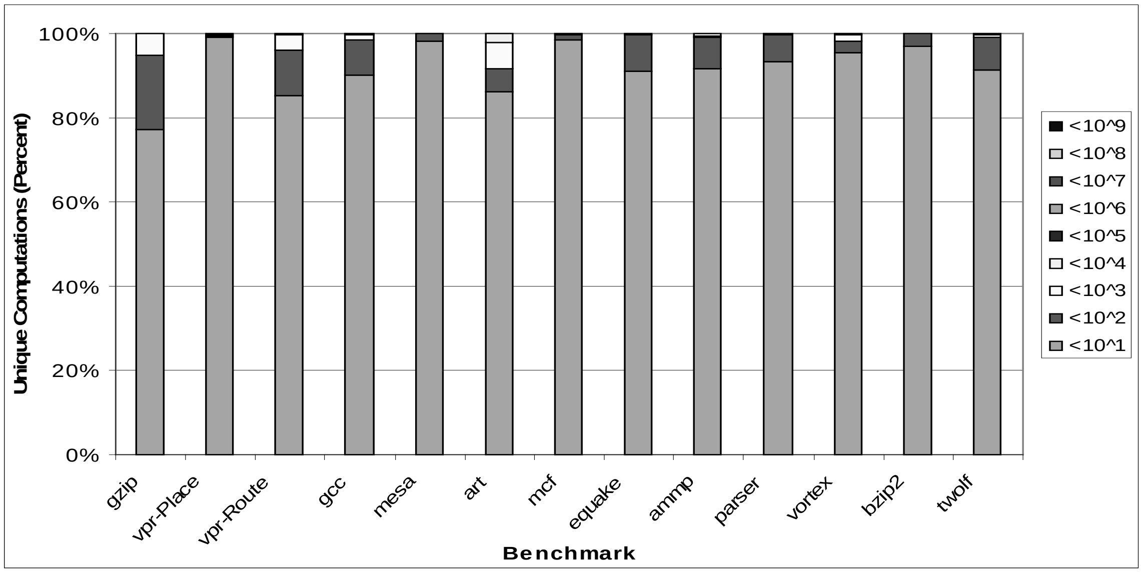

Figure 2.2.2.1: Frequency Distribution of Unique Computations per Benchmark, Local-Level, Normalized

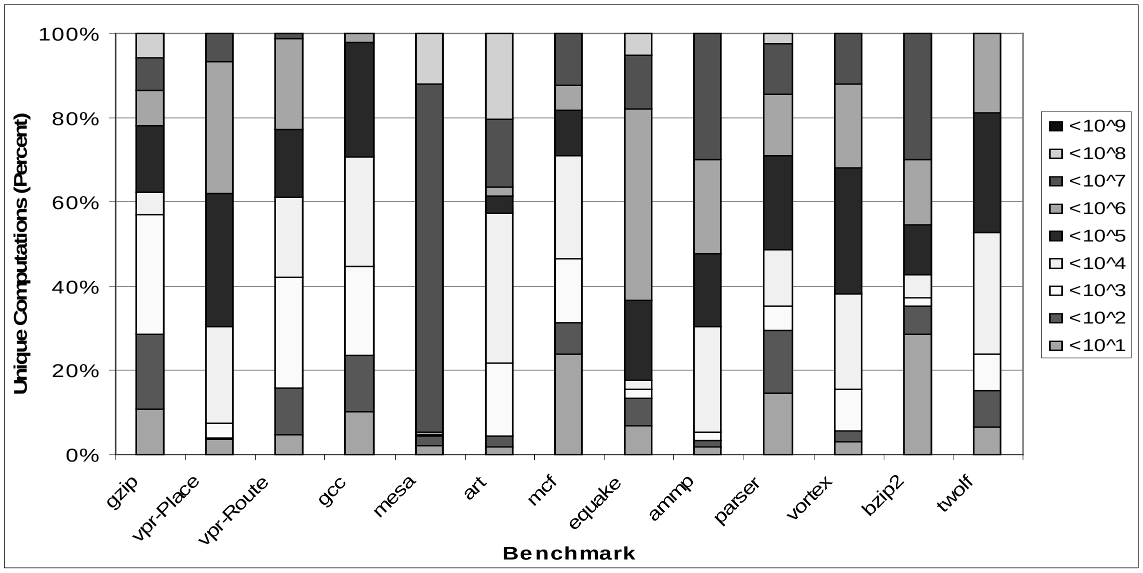

Figure 2.2.2.2: Percentage of Dynamic Instructions Due to the Unique Computations in each Frequency Range, Local-Level, Normalized

Figure 2.2.2.1 shows the frequency distribution of the unique computations at the local-level while Figure 2.2.2.2 shows the percentage of dynamic instructions due to the unique computations in each frequency range at the local-level.

Since unique computations that differ only by their PC values map to the same unique computation at the global-level while mapping to different unique computations at the local-level, there are fewer unique computations in the global-level case. Consequently, a direct comparison of Figure 2.2.1.1 with Figure 2.2.2.1 does not make sense. However, for each benchmark, global and local level unique computations can be compared by the number of instructions that they represent.

A comparison of Figures 2.2.1.2 and 2.2.2.2 shows that global-level unique computations in the higher frequency ranges represent more dynamic instructions as compared to the local-level unique computations in the same frequency ranges. This result means that that a single global-level unique computation represents a larger percentage of instructions as compared to the corresponding local-level unique computation. Therefore, using Instruction Precomputation to exploit redundant computations at a global-level should yield a larger performance benefit.

To summarize the difference between Figures 2.2.1.2 and 2.2.2.2, Table 2.2.2.1 compares the percentage of instructions that are represented by the 2048 highest frequency unique computations at both levels. The second and third columns show the percentage of dynamic instructions that are due to the top 2048 global and local level unique computations, respectively, while the fourth column is the difference of the second and third columns.

Table 2.2.2.1 shows that a global-level based Instruction Precomputation mechanism could reuse an additional 0.01% (*mesa*) to 12.59% (*gcc*) of the total number of dynamic instructions as compared to the local-level. In other words, Table 2.2.2.1 shows that global-level Instruction Precomputation has greater potential for performance improvement than local-level Instruction Precomputation.

Table 2.2.2.1: Percentage of Instructions Due to the 2048 Highest Frequency Unique Computations at the Global and Local Levels

Benchmark	Global	Local	Global - Local
gzip	14.68	11.77	2.90
vpr-Place	40.57	29.76	10.82
vpr-Route	23.44	19.88	3.55
gcc	26.25	13.66	12.59
mesa	44.49	44.48	0.01
art	20.24	13.82	6.42
mcf	19.04	13.63	5.41
equake	37.87	35.29	2.58
ammp	23.93	19.12	4.81
parser	22.86	18.59	4.27
vortex	25.24	21.67	3.57
bzip2	26.83	23.54	3.28
twolf	23.54	16.57	6.97

To determine whether the amount of redundant computations are simply the result of the benchmark itself or of the benchmark's input set, the same benchmarks were profiled with another input set. The results from the second input set were very similar to the results from the first. Therefore, the benchmarks, and not their inputs, are the cause of redundant computations.

2.3. The Mechanics of Instruction Precomputation

As described briefly in Chapter 1, Instruction Precomputation consists of two main steps: static profiling and dynamic removal of redundant computations. In the profiling step, the compiler runs the benchmark with a representative input set to determine the unique computations with the highest frequencies of execution. Instead of determining only the unique computations with the highest frequencies of execution, the compiler could also factor in the each instruction's latency to determine the unique computations with the highest frequency/latency products (F/LP). The F/LP is simply the unique computation's frequency of execution multiplied by its execution latency. Therefore, instructions that have a single-cycle execution latency have F/LP that are the same as their frequencies of execution.

Although the compiler uses a "representative" input set to profile the benchmark, the key question is: Is there a correlation between the set of unique computations and the specific input set? In other words, will the compiler assemble a very different set of unique computations for each input set? If so, then Instruction Precomputation cannot be used to improve the performance of the processor. If not, then the unique computations are a function of the benchmark, and not the input set, which means that Instruction Precomputation could significantly improve the processor's performance.

Table 2.3.1: Number of Unique Computations that are Present in Two Sets of the 2048 Highest Frequency Unique Computations from Two Different Input Sets

Benchmark	In Common	Percentage
gzip	2028	99.02
vpr-Place	527	25.73
vpr-Route	1228	59.96
gcc	1951	95.26
mesa	589	28.76
art	1615	78.86
mcf	1675	81.79
equake	1816	88.67
ammp	1862	90.92
parser	1309	63.92
vortex	1298	63.38
bzip2	1198	58.50
twolf	397	19.38

One approach to determine whether or not the highest frequency unique computations are a function of the benchmark or input set is to determine the amount of "overlap" between two sets of high frequency unique computations that were produced by different input sets. Table 2.3.1 shows the number of unique computations that are common across two sets of the top 2048 highest frequency unique computations. The second column shows the number of unique computations that are present in both sets while the third column shows that number as percentage of the total number of unique computations (2048).

Table 2.3.1 shows that with the exceptions of *vpr-Place, mesa,* and *twolf*, at least 50% of unique computations in one set are present in the other set. For *gzip, gcc*, and *ammp*, over 90% of the unique computations in one set are present in the other. While the percentage is below 50% for *vpr-Place*, *mesa*, and *twolf*, that percentage is affected by the number of unique computations in each set. The problem is partially due to limiting the set to N unique computations instead of N different frequencies. For instance, if the 2048th and 2049th unique computations in the same set have the same frequency, only the 2048th unique computation is checked against the other set of unique computations since the 2049th unique computation is not included in that set. As a result, if several unique computations have the same frequency of execution, some of them will be included in one of the two sets, but not the other. In that case, those unique computations will not appear to be in common between the two sets.

While this reason would seem to be relatively insignificant, the results show that this reason is very significant for a few benchmarks. For example, in *mesa* and *art,* 666 and 8416 unique computations, respectively, have the same frequency of execution as the 2048th highest frequency unique computation. This result shows that, for these two benchmarks, the number of unique computations that are present in two input sets is deceptively low.

However, the key conclusion from Table 2.3.1 is that for most benchmarks, a significant percentage of the unique computations are present in both sets. Consequently, the conclusion is that the highest frequency unique computations are primarily a function of the benchmark and less a function of the specific input set.

After the compiler determines the set of the highest frequency unique computations, they are compiled into the program binary. Therefore, each set is unique only to that program.

The second step for Instruction Precomputation is the removal of redundant computations at run-time. Before the program begins execution, the processor initializes the Precomputation Table (PT) with the unique computations in the program binary. Then, as the program executes, for each instruction, the PT is checked to see if there is a match between the opcodes and input operands of the PT entries and the opcode and input operands of the current instruction. If a match is found, then the Instruction Precomputation hardware forwards the value in the output field of the matching PT entry to the instruction. As a result, since that instruction does not need to continue through the remaining stages of the pipeline, it can be removed from the pipeline to await in-order commit. If a match is not found, then the instruction continues through the pipeline and executes as normal.

Figure 2.3.1 shows how the PT is integrated into the processor's pipeline.

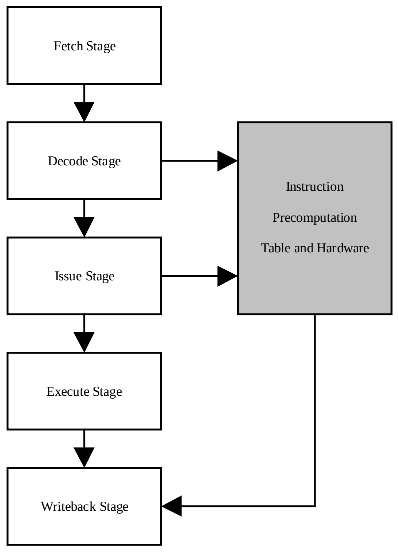

Figure 2.3.1: Operation of a Superscalar Pipeline with Instruction Precomputation

During the decode and issue stages, the opcode and input operands for each dynamic instruction are sent to the PT, when available. The Instruction Precomputation hardware then determines if there is a match between the current opcode and input operands with the unique computations in the PT. If a match is found, the Instruction Precomputation hardware sends the output value for that instruction to the writeback stage, which commits that value when the instruction is retired.

It is important to note that, unlike value reuse, Instruction Precomputation never updates the PT whenever the matching unique computation is not found. Rather, the PT is initialized just before the program starts executing.

Since instructions are removed from the pipeline only if a matching computation is found in the PT, Instruction Precomputation is a **non-speculative** optimization, i.e. the output value that is forwarded from the PT to the instruction is the correct and final value. The advantage of non-speculative optimizations is that they do not need hardware to check the correctness of their speculation.

2.3.1. How it Improves Performance

Incorporating Instruction Precomputation into a superscalar processor improves the processor's performance in two ways. First, forwarding the output value of the instruction early in the pipeline reduces the effective latency of the instruction (as opposed to normal execution). This reduces the "CPI" term in Equation 1.3.1. Instructions that are dependent on those redundant computations can also begin executing earlier (as compared to when Instruction Precomputation is not used). Second, dynamically removing the instruction from the pipeline decreases the number of resource conflicts in the later stages of the pipeline. Decreasing the number of resource conflicts makes more issue slots available for other instructions, allows other instructions to execute sooner on a functional unit, decreases the number of instructions that are on a bus, etc. Consequently, since fewer resource conflicts means that instructions can begin and finish execution faster, reducing the number of resource conflicts reduces the CPI term.

2.4. A Comparison of Instruction Precomputation and Value Reuse

Overall, Instruction Precomputation and value reuse are similar approaches. Both methods dynamically remove instructions that are redundant computations from the pipeline after forwarding the correct output value to that instruction. Both methods define a redundant computation to be one that is currently in the PT or VRT.

While these two approaches are generally similar, the key difference between the two is how a redundant computation gets into the PT or VRT. In Instruction Precomputation, only the highest frequency redundant computations – which are determined by compiler profiling – are put into the PT. Since it is likely that, for that particular input set, the highest frequency unique computations are already in the PT, there is no need for dynamic replacement.

On the other hand, in value reuse, if a unique computation is not found in the VRT, then it is added to the VRT. Therefore, even if a computation has not been redundant or will not be redundant, is still is added to the VRT, even if it replaces a high frequency redundant computation.

The one advantage that value reuse could have over Instruction Precomputation is faster table access, depending on its implementation. Instead of comparing the current instruction’s opcode and input operands against every unique computation in the VRT, using the current instruction’s PC as an index into the VRT can reduce the VRT access time by quickly selecting the matching table entry (although the input operands still need to be compared). As a result, not only does this approach require fewer comparisons, it also removes the need to compare opcodes since there can only be one opcode per PC. However, the depth of this advantage is difficult to quantify since it depends on the size of both tables, the exact implementation of both mechanisms, the processor’s architecture, and the processor’s clock frequency.

Chapter 3

Trivial Computations

As described in Chapter 1, adding hardware to simplify and eliminate trivial computations can improve the processor's performance. This chapter describes the problem that the Simplification and Elimination of Trivial Computations attempts to solve, the program characteristic that it exploits, how it operates, and the hardware that is needed.

3.1. Definition of Trivial Computations and How to Exploit Them

A significant percentage of a program's total instruction count is due to trivial computations, which are the result of the way programs are written and compiled. A trivial computation is an instruction whose output value can be determined without having to perform the specified computation by either converting the operation to a less complex one or by determining the result immediately based on the value of one or both of the inputs. An example of the former is a multiply operation where one of the input operands has a value of two. In this case, the multiply instruction can be converted to a shift-left instruction. Therefore, instead of executing the original multiply instruction with a multiply unit, that instruction can be **simplified** to a shift-left instruction which can then execute on a general integer ALU unit.

Converting the multiply instruction to a shift instruction decreases the execution latency of the instruction. Instead of computing the output value by using an integer multiply unit that requires multiple clock cycles, the same result can be computed by using a functional unit that requires only one or two clock cycles. Furthermore, since there usually are two to three times more integer ALU functional units as compared to integer multiply functional units in modern superscalar processors, simplifying the trivial computation also reduces the number of resource conflicts.

An example of the latter type is an add instruction where one of the input operands is zero. In this case, the result of the instruction is the value of the other input operand. Therefore, since the value of the output is equal to the value of the non-zero input, no computation needs to be performed. Since one of the inputs is trivial (zero), this computation can be **eliminated**.

Detecting a trivial input and then eliminating that computation improves the processor's performance in three ways. First of all, dynamically eliminating those computations obviates the need to use a functional unit, thus reducing the number of resource conflicts. Second, eliminating, instead of executing, trivial computations reduces the execution latency of those instructions. Finally, and most importantly, some trivial computations can be eliminated before both input operand values are available. As a result, not only is the result of that instruction non-speculative, it is also available earlier than is normally possible since the result is available before the value of both input operands are available.

For example, since the output value of the trivial computation X*0 is zero, regardless of what the value of X is, by using extra hardware to eliminate that trivial computation, the result of that computation will be available before the value of X is available. To illustrate the potential performance improvement, assume that the value of X is available two cycles after the zero input operand is available, that multiply instructions have a 10 cycle execution latency, and that multiply instruction has to pass through another 4 pipeline stages before it is executed. As a result, it takes 16 (2+10+4) additional cycles to finish execute this instruction normally after the zero input operand value is available, which is 15 more cycles than it would take to eliminate this trivial computation. Even if the multiply instruction takes only a single cycle, eliminating that trivial computations saves at least 6 cycles as compared to normal execution.

It is important to note that dynamically simplifying and eliminating trivial computations is a non-speculative optimization.

Finally, while it seems as though an optimizing compiler should be able to remove many of these trivial computations, it is unable to do so unless the value of the input operands is known at compile time. Furthermore, the compiler may use trivial computations, such as 0+0, for initialization purposes. Consequently, since the compiler is unable to remove these trivial computations and also since the compiler deliberately introduces additional trivial computations into the program, it is left to the hardware to simplify and eliminate these trivial computations at run-time.

3.2. The Amount of Trivial Computations

While the previous sub-section first defined what trivial computations are, how they could be exploited to improve the processor's performance by simplifying and eliminating them, and how they actually go about improving the processor's performance, that description is somewhat useless if trivial computations are not prevalent in typical programs. The goal of this sub-section is to determine percentage of dynamic instructions that are trivial computations.

Table 3.2.1 shows the types of computations that are defined as trivial in this dissertation. The first column shows the type of operation while the second column shows how the result is normally computed. The third and fourth columns show which trivial computations can be eliminated and simplified, respectively.

Table 3.2.1: List of Trivial Computations.

Operation	Normal	Can be Eliminated	Can be Simplified
Add	X+Y	X,Y=0	
Subtract	X–Y	Y=0; X=Y	
Multiply	X*Y	X,Y=0	X,Y=Power of 2
Divide	X÷Y	X=0; X=Y	Y=Power of 2
AND	X&Y	X,Y={0,0xffffffff}; X=Y	
OR, XOR	X|Y, X⊕Y	X,Y={0,0xffffffff}; X=Y	
Logical Shift	X<<Y, X>>Y	X,Y = 0	
Arithmetic Shift	X<<Y, X>>Y	X={0,0xffffffff}; Y=0	
Absolute Value	|X|	X={0, Positive}	
Square Root	$\sqrt{X}$	X=0	X=Even Power of 2

The trivial computations that can be eliminated can also be divided into two groups, those that can benefit from non-speculative, early execution and those that do not. Table 3.2.2 lists the trivial computations in the former category.

Table 3.2.2: Trivial Computations that Benefit from Non-Speculative, Early Execution

Operation	Normal	Early Execution Candidates
Add	X+Y	
Subtract	X–Y	
Multiply	X*Y	X,Y=0
Divide	X÷Y	X=0
AND	X&Y	X,Y=0
OR	X|Y	X,Y=0xffffffff
Logical Shift	X<<Y, X>>Y	
Arithmetic Shift	X<<Y, X>>Y	X={0,0xffffffff}
Absolute Value	|X|	
Square Root	$\sqrt{X}$	

For those trivial computations in Table 3.2.2, when the specific input is found to be trivial, not only is the computation also trivial, but the result is either 0 or 0xffffffff. For example, the computation X & 0 always produces an output value of 0, regardless of the value of X. Similarly so, X | 0xffffffff is always 0xffffffff, again, regardless of the value of X. The trivial computations not shown in Table 3.2.2 have to wait for both input operands before the computation is known to be trivial or before the result is known.

It is obvious as to why most of the computations in Table 3.2.1 are trivial with the possible exceptions of XOR and square roots for an even power of two. For XOR, there are three possible trivial inputs: 0, 0xffffffff, and X=Y. For $X \oplus 0$, the output value is X. For $X \oplus$ 0xfffffff, the output value is ~X (complement of X). Finally, when X = Y, the output value of $X \oplus Y$ is 0.

For a square root, if the value of X is an even power of two (e.g. 4, 16, 64), then the result can be computed by halving the value in the exponent field. As a result, the exponent needs only to be shifted to the right by one bit. For example, the exponent for 16 is 0100 (4). By applying this simplification, 0100 is right-shifted by one to produce 0010 (2). Using this new exponent, the new value of this number is 4 ($1*2^2$) which is the square root of 16.

Finally, it is important to note that the definitions for add, sub, multiply, and divide trivial computations in Table 3.2.1 are not limited only to integer operations, but are also equally applicable to floating-point operations. However, due to differences in how the number is represented (e.g. two's complement versus IEEE floating-point notation), how a floating-point computation is simplified and eliminated may differ as compared to its integer counterpart.

Given the list of the trivial computations in Table 3.2.1, Figure 3.2.1 shows the percentage of trivial computations in selected benchmarks from the SPEC 2000 and MediaBench [Lee97] benchmark suites for each instruction type and for each benchmark suite. The second column in Table 5.4.1 shows the specific SPEC input set that was used for results in this figure while the second column of Table 5.4.2 shows the specific MediaBench input set. The benchmarks in the MediaBench benchmark suite represent multimedia application programs such as compression, encryption, and encoding.

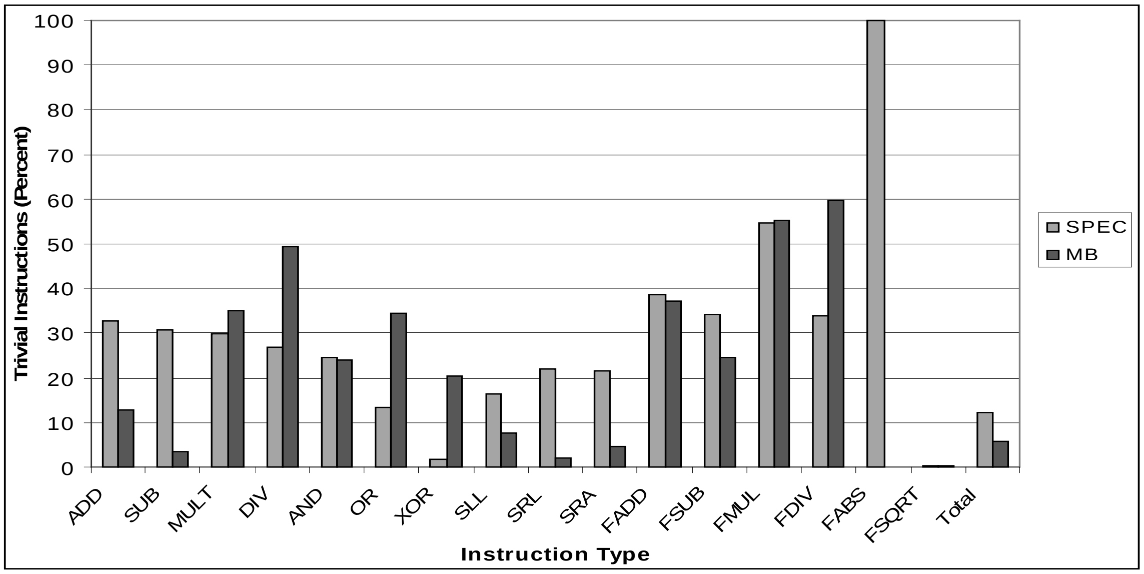

Figure 3.2.1: Percentage of Trivial Computations per Instruction Type and per Total Number of Dynamic Instructions for the SPEC and MediaBench Benchmarks

Overall, these results in Figure 3.2.1 show that trivial computations account for 12.24% and 5.73% of the total instructions in these SPEC and MediaBench benchmarks, respectively. Although some multimedia benchmarks may have a significant amount of trivial computations, the benchmarks that were selected from the MediaBench benchmark suite for this study clearly do not.

Figure 3.2.1 also shows that almost all instruction types have a significant percentage of trivial computations. However, a high percentage does not necessarily mean that those instructions will have a significant impact on the program's overall execution time since they could account for a very small percentage of the total executed instructions. For example, 100% of the absolute value instructions (FABS) are trivial, but they account for only 0.05% of the total instructions executed.

To determine whether the trivial computations are a result of the benchmark itself, or of the benchmark's input set, the same benchmarks were profiled with another input set.

The results from the second input set were very similar to the results from the first, with the exception that with the other input set, the MediaBench benchmarks had a higher percentage of trivial computations (7.43%). This result indicates that trivial computations are primarily due to the benchmark programs themselves and not due to the specific values of their inputs. However, for the MediaBench benchmarks, the input set has a larger effect on the amount of trivial computation than it does for the SPEC benchmarks.

3.3. The Mechanics of Trivial Computation Simplification and Elimination

The first step to simplify or eliminate a trivial computation is to determine if one of the input operands (X, Y) is trivial (0, 0xffffffff, positive, a power of two, or an even power of two) or if the two input operands are equal to each other. However, since different instruction types have different sets of trivial inputs, the opcode is needed to determine the possible set of trivial inputs. To determine whether or not one or more of the input values are trivial, comparators are used to compare each input operand against the set of candidate trivial input values for that operation. Since the input operand values may arrive in different clock cycles, the comparators need to check the input operand values as they arrive. Since it is possible for an instruction to be both simplified **and** eliminated, as is the case for 2*0, combinational logic is needed to determine if that is the case and then to favor eliminating the computation. Figure 3.3.1 shows the hardware that is necessary to implement this logic and how this logic fits in the processor's pipeline.

In this figure, the trivial computation hardware is placed between the reorder buffer, which stores the results of the decode stage, and the issue stage. The input operand values, X and Y, for each instruction are sent to the six trivial operand comparators. The comparators check if the input operand values are equal to zero (= 0), are all ones (All 1's), are equal to each other (X = Y), are positive (> 0), are a power of two (log2), or are an even power of two (E log2). The inputs to the trivial computation logic are the results of each comparison and the opcode for that instruction. The trivial computation logic uses those inputs to first determine if the current instruction is a trivial computation. If the current instruction is a trivial computation that can be eliminated, the output value for that instruction is sent back to the reorder buffer using the “Result” bus. If the current instruction is a trivial computation that can be simplified, the trivial computation logic generates the re-coded opcode and input operand values and sends them to the issue logic. However, if the trivial computation can be eliminated and simplified, the trivial computation logic will favor the eliminate option and simply generate the output value.

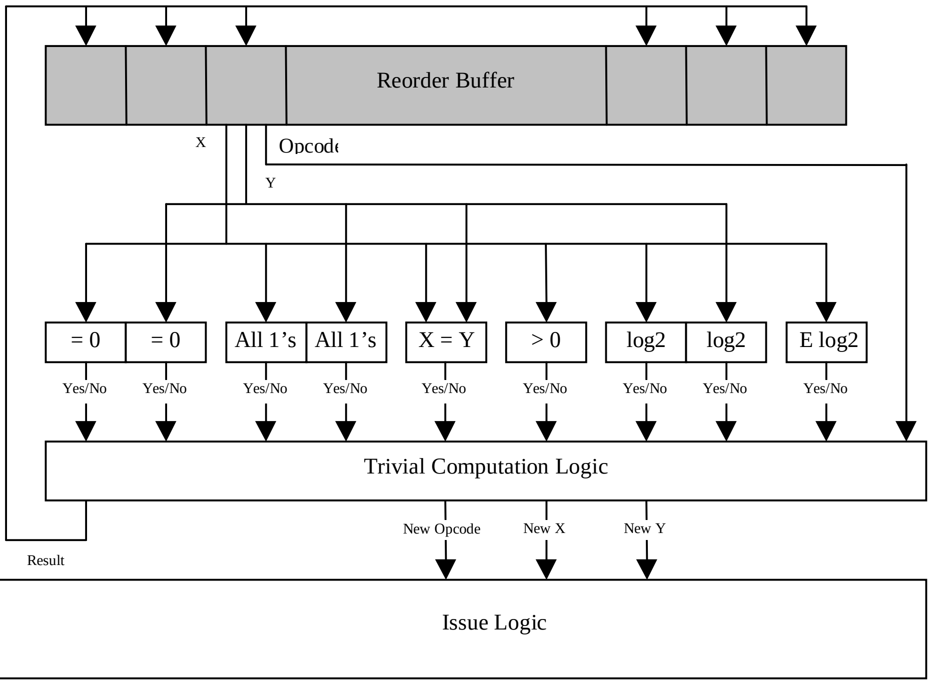

Figure 3.3.1: Trivial Computation Hardware and Its Processor Interface

After determining if the instruction is a trivial computation, then second step is to either simplify the computation or eliminate it. However, somewhat ironically, simplifying a computation is not a simple process. Simplifying an integer multiply or divide instruction means that the instruction needs to be converted to a shift-left (multiply) or shift-right (divide) instruction. Additionally, the trivial input operand (i.e. the one that is a power of two) needs to be converted into the appropriate shift amount. The non-trivial input operand needs to be shifted by the same amount as the bit-number of the only bit that is “1” in the trivial input. For instance, assuming that the computation to be simplified is X/4, the bit-pattern for 4 is: 0000 … 0000 0100. Therefore, the only bit that is “1” in that bit-pattern is in bit position “2”. Consequently, X needs to be shifted to the right by 2.

The process of simplifying floating-point multiply and divide instructions is similar with one key exception. Instead of shifting the instruction to the left or right, the shift amount is added to or subtracted from the exponent of the non-trivial input operand. As a result, instead of using a shifter, the simplified instruction should use an adder or subtractor. Since the exponent field is confined to a predetermined field of bits, an integer adder or subtractor can be used to add or subtract the shift amount.

As explained above, to simplify a square root computation that is trivial (an even power of two), only the exponent field of the input needs to be shifted to the right by one. As a result, since the format of the number is not a problem, an integer shifter can be used to simplify the computation. Alternatively, dedicated shift-right-by-one can be implemented next to the trivial input detection logic so the simplified instruction does not need to go through the pipeline.

Finally, after the instruction has been simplified by re-coding it, it can be sent to the functional units to be executed. After the simplified instruction finished executing, the result for that instruction is written to the register file and the instruction is committed, just like any other instruction.

By comparison, eliminating a trivial computation is a much simpler task than simplifying one. The four output values of trivial computations that can be eliminated are 0, 1, 0xffffffff, and X (assuming that Y is the trivial input). It is important to notice that, for the first three output values, to eliminate that trivial computation, after detecting that that instruction is a trivial computation, simple combinational logic can be used to set the output to the correct value. In the case where the output value is equal to X, combinational logic sets the output to the value of X when the value of X arrives. Then, for all four cases, after the output value is set, the instruction needs only to write its output value to the register file and be committed.

Finally, in the cases of trivial computation where non-speculative, early execution applies, the output value is set immediately, whether or not the non-trivial input operand is available. And if the value of the non-trivial input operand was not available when the output value was set, then when it is arrives, it is ignored since it does not change the output value.

3.4. Hardware Cost to Simplify and Eliminate Trivial Computations

The minimum hardware cost of exploiting trivial computations, using this dissertation's definitions of trivial computations, are the nine comparators shown in Figure 3.3.1, the trivial computation logic, a few multiplexors, a few tri-state gates, and a few extra busses. However, the final hardware cost depends on the base processor architecture and how aggressively the architect wants to exploit trivial computations. The maximum cost corresponds to when the trivial computation hardware shown in Figure 3.3.1 is replicated for each reorder buffer entry and for each reservation station entry (the input buffer for the functional units) while the minimum cost corresponds to when all reorder buffer entries share the same set of comparators and the same trivial computation logic. Specific implementations in between either extreme result in hardware costs in between the two extremes.

Chapter 4

Simulation Methodology

As described in Chapter 1, without a rigorous simulation methodology, the computer architect may make false conclusions or may not glean the maximum amount of information from the simulation results. Therefore, to improve the simulation methodology in computer architecture research, this dissertation uses a fractional multifactorial design-of-experiments, the Plackett and Burman design [Plackett46], as the foundation to improve specific stages of the simulation process. More specifically, the remainder of this chapter describes how the results of the Plackett and Burman design can be used to choose processor parameter values and benchmarks, and also how those results can be used to analyze the effect of a processor enhancement.

4.1. An Overview of Simulation Methodology

The most important tool in processor design and computer architecture research is the processor simulator. Using a simulator reduces the cost and time of a project by allowing the architect to quickly evaluate the performance of different processor configurations instead of fabricating a new processor for each configuration, a process that may take several weeks or months and is extraordinarily expensive. Additionally, a simulator is much more flexible than fabricating the processor since it can somewhat accurately determine the expected performance of a processor enhancement without having to undergo all the necessary circuit-level design steps. Note that the term “processor enhancement” includes both microarchitectural and compiler enhancements.

Since simulators are more cost-effective, flexible, and efficient than fabricating a processor, computer architects use their results to guide design decisions, determine what points to explore in the design space, and to quantify the efficacy of a processor enhancement. Consequently, since misleading simulation results can severely affect the final design of the processor or lead to erroneous conclusions, the accuracy of the simulator’s results is extremely important. Therefore, to minimize the amount of error in the simulation results, computer architects need to do two things. First, they should try to minimize the amount of error inherent to the simulator (as compared to a hardware version of the processor the simulator models). Second, they should try to reduce the amount of “error” that the user introduces when running simulations. A user may introduce additional error into the simulation results by choosing a poor set of processor parameters or benchmarks that over or under inflate the processor’s performance, or power consumption, reliability, etc. While current research also focuses on decreasing the processor’s power consumption and improving its reliability, for brevity, the remainder of this section assumes that the computer architect is only trying to improve the processor’s performance. However, the techniques that are described in this chapter are equally applicable to power consumption reduction and reliability improvement.

Furthermore, since the simulation results are used to make design decisions, it is also important to glean the maximum amount of information from each set of simulation results so that accurate conclusions can be drawn. For example, while the speedup metric measures the overall performance impact of a processor enhancement, what it does not reveal is “how” the processor enhancement got that final result. For instance, while the processor enhancement may relieve one bottleneck (such as the cache miss rate), the same enhancement may exacerbate another (such as the amount of memory traffic), which then could be the performance-limiting factor. Consequently, the speedup represents the net effect that the processor enhancement has on these two bottlenecks. Higher speedups may be possible by redesigning the enhancement such that the second bottleneck is less of a limiting factor.

In spite of this dependence on simulators, relatively little research has focused on decreasing the amount of error in simulation results by improving the accuracy of simulators and by improving simulation methodology. In fact, current simulation methodology is, at best, ad-hoc. Therefore, to decrease the amount of error in the simulation results and also to improve the overall quality of the simulation methodology, this dissertation introduces rigorous, statistically-based simulation methodology.

4.1.1. Principal Steps of the Simulation Process

In computer architecture research, the simulation process is the sequence of steps that architects must perform to run their simulations and to analyze their simulation results. Most architects start with a publicly available simulator, such as SimpleScalar, and then add their own code to the simulator to model their enhancement. This dissertation divides the simulation process into six major steps. They are:

1) Simulator Design and Validation
2) Processor Enhancement Implementation and Verification
3) Processor Parameter Selection
4) Benchmark and Input Set Selection
5) Simulation
6) Analysis of an Enhancement’s Effect

In each of the following paragraphs, a short description of each step is given along with a short description of the some of the potential errors that the user could make.

In the first step, the simulator is designed, implemented, and verified. The most important design decision when implementing a simulator is how much detail to incorporate into the simulator. For example, if the simulator does not fully model the memory traffic within the memory hierarchy, the performance of the simulated processor will be higher-than-should-be-expected since the effect of bus traffic and other resource limited hazards are removed. While adding more detail into the simulator improves its accuracy, it comes with the price of a slower simulation speed, which results in longer simulation times. In summary, the goal of this step is to produce an accurate simulator that fully models all of the key components, i.e. the components that have a significant effect on the simulated performance.

In the second step of the simulation process, computer architects implement their processor enhancement into the simulator or into the compiler. In the former case, the processor enhancement is a hardware-based solution while the latter is software-based. After implementing the processor enhancement, it is very important for the architect to verify if their implementation functions correctly and if it is sufficiently detailed to accurately model the enhancement. In summary, the goal of this step is to create an accurate representation of the processor enhancement.

In the third step of the simulation process, the computer architect chooses values for the user-configurable processor parameters. Typical processor parameters include cache size, associativity, and block size; reorder buffer size; branch predictor type and size; and the number and type of each functional unit. In this dissertation, the generic term "processor parameters" includes parameters in the processor core and key memory parameters. Choosing the set of values for the processor parameters is important since a poorly chosen set may result in creating artificial bottlenecks or minimizing real bottlenecks that are not present in commercial processors.

The fourth step in the simulation process is similar to the third. In this step, the computer architect chooses a sub-set of benchmarks from the benchmark suite and a set of inputs for those benchmarks. Since processor enhancements either aim to improve the processor's performance for a wide range of programs (i.e. general-purpose computing) or for a specific set of programs and since the purpose of simulations is to quantify the effect that a processor enhancement has, the computer architect needs to carefully choose a sub-set of benchmarks and inputs that is appropriate. After choosing the benchmarks, the architect should compile them with a state-of-the-art compiler (with the appropriate compilation options), if the simulator models a state-of-the-art processor. Choosing an unrepresentative set of benchmarks and inputs can under or overstate the expected performance of the processor which, in effect, introduces additional "error" into the simulation process. In summary, the goal of this step is to choose an appropriate set of benchmarks and inputs that will yield an accurate estimate of the expected performance. Poorly choosing a set of benchmarks and inputs could inadvertently introduce additional error into the simulation results.

In the fifth step, the computer architect performs the simulations. Although the setup phase of the simulation process is concluded with step four, decisions on how actually to perform the simulation still need to be made. For instance, to reduce the simulation time even more, the computer architect may wish to employ one of two options. First, to reduce the simulation time of the initialization phase of the program, which is less interesting than the remainder of the program, the computer architect may decide to "fast-forward" (functional execution without any timing information) through most of the initialization phase. The upside of this approach is that the architect can test the performance of the enhancement on a more interesting part of the program while limiting the total simulation time. The downside is that the performance results may be misleading since they do not account for the execution time of the initialization section.

In the other option, the simulation is terminated after a certain user-chosen number of instructions. The upside of this approach is the assumption that after that number of instructions, the processor has already executed some or all of the most interesting aspects of the program. Therefore, simulating the program to the completion may not yield any additional information about what effect the enhancement has on the processor. However, the downside is that skipping parts of the program may introduce some amount of error into the simulation results.

In all of the first five steps of the simulation process, the computer architect may fall into several pitfalls that will affect the accuracy of the simulation results. And since the architect makes conclusions based on the simulation results, inaccurate simulation results may lead to inaccurate conclusions. However, since the magnitude and net effect of each of the errors is currently unknown, it is impossible to say whether or not the simulation results even produce an indicative trend. Consequently, it is very important to decrease the amount of error in some or all of these steps.

The sixth and final step of the simulation process is analyzing the simulation results. One goal of this step is to thoroughly understand the effect that the enhancement has on the processor and how the enhancement interacts with the processor. Understanding these two effects are central to understanding what limits the performance improvement of the enhancement. To accomplish that goal, the computer architect should look beyond single-valued metrics, such as the speedup or the cache miss rate, to multi-valued metrics, such as the distribution of the performance bottlenecks, which form a more complete picture of the enhancement's effect.

4.1.2. Focus of this Dissertation

This dissertation focuses on improving the simulation methodology of the third step, processor parameter selection; the fourth step, benchmark selection; and the last step, the analysis of an enhancement's effect. This dissertation excludes the first step (simulator design and validation) for two reasons. First, since most computer architects do not implement their own simulator, but rather use publicly available simulators as their base simulator, focusing on this step benefits only a small number of computer architects. Second, as will be described in Chapter 7, there have been a few papers that have focused on improving the accuracy of simulators. For similar reasons, this dissertation does not focus on the second (processor enhancement implementation and verification) and fifth (simulation) steps.

4.2. Fractional Multifactorial Design of Experiments

As described in Chapter 1, in a sensitivity analysis, the values of some parameters are varied while the values of the remaining parameters are fixed. Then, by measuring the change in the output value in response to changes in the values of the variable parameters, a computer architect can determine the effect that each variable parameter, or combination of parameters, has on the output.

Multifactorial designs are statistical methods that allow the user to determine the specific effect that a variable parameter has on the output [Montgomery91]. However, one key difference between a multifactorial design, such as the analysis of variance (ANOVA), and a sensitivity analysis is the level of detail that can be extracted from each. In ANOVA, the user can determine the percentage effect that each parameter and interaction (combination of parameters) have on the change in the output value. For example, the user may calculate that parameter A accounts for 25% of the total amount of change (variation) in the output while parameter B accounts for 70%. In this particular example, interaction AB accounts for the remaining 5%. On the other hand, a sensitivity analysis may only reveal that parameter B has more of an effect on the output than does parameter A and that their interaction has even less. In other words, the conclusions that are derived from a sensitivity analysis are inherently more limited while the multifactorial design conclusions are more analytical.

While the outputs of a multifactorial design, such as ANOVA, may be more detailed than the outputs of a sensitivity analysis, using the same multifactorial design for all situations it not necessarily the best solution. The basic difference between multifactorial designs is the trade-off between the level of detail and the required number of simulations. In general, the multifactorial designs that yield the highest level of detail also require the largest number of simulations. While having a higher level of detail gives the user more information from which to form a conclusion, the required simulation time to produce that level of detail may be prohibitively high. (The total simulation time is equal to the average time per simulation multiplied by the total number of simulations. The total number of simulations depends on the number of variables, the number of values per variable, and the specific multifactorial design.) The appropriate multifactorial design is the one that exceeds the minimum level of detail needed while having a tractable amount of simulation time.

In this dissertation, to improve the simulation methodology in computer architecture research, the minimum level of detail that is required is the quantitative effect of each individual parameter and the quantitative effect of all the significant interactions. (There are 41 individual, user-configurable parameters in the SimpleScalar simulator that is used in this dissertation.) The maximum amount of simulation time is limited by the total number of processors that can be used to run the simulations and their availability.

4.2.1. Comparison of Statistical Designs

To determine the effect of the individual parameters and the most significant interactions, three different multifactorial designs were considered for use: the "one-at-a-time" design, the ANOVA design, and the Plackett and Burman design. For reasons that will be explained below, the ANOVA design is an example of a full multifactorial design while the Plackett and Burman design is an example of a fractional multifactorial design. Each of the following three sub-sections describes the mechanics of each design, as well as its advantages and disadvantages.

4.2.1.1. One-at-a-Time Design

In the one-at-a-time experimental design, the value of one parameter is varied while the other parameters are set to constant values. For example, if each parameter can take one of two values, the parameter that is being varied is set to its "high" value while all of the other parameters are set to their respective "low" values. The high and low values for each parameter represent the range of values that that parameter can take. Then, the user can determine the effect of each parameter by comparing the output value when each parameter is at its high value, while the other parameters are at their low value, with the configuration where all parameters are at their low values. The latter case is the baseline case. Consequently, for N parameters and for two values for each parameter, this design requires only N+1 simulations. This design requires N simulations to vary each parameter to its high value and one simulation for the baseline case. (If each parameter can be set to X different values, these designs require X*N+1 simulations.)

The advantage of this design is that it requires the absolute minimum number of simulations. As a result, the amount of time required to execute this design and determine the effect of each parameter is relatively low.

However, these designs should be avoided because they are vulnerable to masking important effects that exist due to interactions between the parameters. Suppose the interaction between parameters A and B is very significant, i.e. that interaction has a large effect on the output value. However, since the effect of interaction AB cannot be quantified with this approach, whatever effect that interaction has will appear to be the effect of a single parameter. Additionally, a constant parameter can be set to such an extreme value that it dominates the results and overshadows the effect of the parameters under test. For instance, setting the low value of a buffer's size to be too small can cause this unit to become the performance bottleneck, thereby masking the effect due to another parameter.

Overall, although the number of simulations required for this design is very low, this advantage is more than offset by the low quality of the results. Consequently, since it is difficult to have confidence in these results, this option was discarded in favor of one of the following two options.

4.2.1.2. Full Multifactorial Design: Analysis of Variance (ANOVA) Design

To determine the effect of all single parameters and interactions, a full multifactorial design varies the parameters to account for every possible configuration. For example, for a design with three parameters that can take two possible values, the values of parameters A, B, and C will be: LLL, LLH, LHL, LHH, HLL, HLH, HHL, and HHH, where H and L represent the high and low values, respectively, for each parameter.

This design has the advantage of being able to precisely quantify the effect that any parameter or interaction has on the variation in the output. From a statistical point-of-view, a parameter that has a large effect on the output value will account for a large percentage of the total variation in the output. As a result, by computing the percentage effect that each parameter and interaction has on the variation in the output, the user can determine: 1) Which parameters and interactions are the most significant (have the most effect on the output value) and 2) The relative important of parameters and interactions with respect to each other.

However, the disadvantage of this design is that it is extremely computationally expensive. For example, assume that a computer architect wants to determine the percent effect that all user-configurable parameters and their interactions have on the processor's performance. Also assume that the architect is using a simulator that has N user-configurable parameters and decides to set those N parameters to b different values. Finally, assume also that the each simulation takes an average of X hours. The total amount of time to simulation all of these test cases is $X*b^N$ hours.

For computer architects using the superscalar processor simulator in the SimpleScalar tool suite and 10 benchmarks from the SPEC 2000 benchmark suite, N=41 and X=12. Simulating all 2.2 **trillion** test cases requires a total simulation time of $10*12*2^{41}$ hours, or 30.1 **billion** years! (This is three times longer than the expected lifetime of the sun.)

While these numbers are somewhat facetious, the point of this example is that, for simulation-based computer architecture research, it is virtually impossible to simulate all the test cases for values of N larger than 15. Consequently, the utility of full multifactorial designs is limited to small values of N.

One potential solution to this problem is to fix the values of some parameters while performing the full multifactorial design for the remaining parameters. Ideally, this compromise drastically reduces the simulation time while still determining the effect of the most important parameters. For example, the computer architect using SimpleScalar may choose to fix the values of the 30 **least** important parameters while varying the values of the 11 most important. However, the problem with this approach is that the values of the constant parameters may have a significant effect on the results. Since interactions between constant and variable parameters may have a significant, but unknown effect, on the results, the user cannot have a high level of confidence in the results of this approach.

Overall, although full multifactorial designs give the most information about the effects of parameters and interactions, this advantage is more than offset by the computational cost. Consequently, since this cost was too high given the available computing resources, this option was discarded in favor of the third option, fractional multifactorial designs.

4.2.1.3. Fractional Multifactorial Design: Plackett and Burman Design

In a fractional multifactorial design, all N parameters are varied simultaneously over approximately N+1 simulations, which is the logically minimal number of simulations required to estimate the effect of each of the N parameters. However, unlike the one-at-a-time design where only one parameter is at its high value, in a fraction multifactorial design, half of the values are at their high value. Furthermore, each parameter is set to its high value for half of the simulations.

One well-established fractional multifactorial design is the Plackett and Burman design [Plackett46]. The base Plackett and Burman design requires X simulations, where X is the next multiple of four greater than N. For example, if N=3, then X=4. However, if N=16, then X=20. An improvement on the base Plackett and Burman design is the Plackett and Burman design with foldover [Montgomery91]. This doubles the number of required simulations to 2*X.

A Plackett and Burman design with foldover can accurately quantify the effect that single parameters and selected, two-parameter interactions have on the output variable. Therefore, they cannot quantify the effect of interactions that are composed of three or more parameters. While this may appear to be major problem for computer architects, it is not. The results in [Yi02-1] showed that the most important interactions are due to dominant parameters. Therefore, if the effect of an interaction is significant, it is composed of at least one dominant parameter.

Additionally, [Yi02-1] shows that almost all of the most significant interactions are two parameter interactions. As a result, since a Plackett and Burman design can quantify the effects of all single parameters and all two parameter interactions, and since all other significant interactions are the result of significant single parameters, the Plackett and Burman design is able to capture all of the significant effects. Consequently, a computer architect can use a Plackett and Burman design to characterize the effect that processor parameters have on the performance with a high degree of confidence.

4.2.2. Mechanics of Plackett and Burman Designs

For a Plackett and Burman design, the value of each parameter in each configuration, is given by the Plackett and Burman design matrix. For most values of X, the Plackett and Burman design matrix is simple to construct. Each row of the design matrix specifies if the parameter is set to its high or low value for that configuration. For a Plackett and Burman design with foldover, there are 2*X configurations, or rows. (Only X configurations are needed when using a Plackett and Burman design without foldover.) Each column specifies the values that a parameter is set to across all configurations. For a Plackett and Burman design, with or without foldover, there are X-1 columns. When there are more columns than parameters (i.e. $N < X-1$), then the extra columns are simply “dummy parameters” and have no effect on the simulation results.

For most values of X, the first row of the design matrix is given in [Plackett46]. The value of each entry in the design matrix is either "+1" or "-1" where +1 corresponds to the parameter's high value and -1 corresponds to its low value. The next X-2 rows are formed by performing a circular right shift on the preceding row. Finally, row X, the last row of the design matrix (without foldover), is a row of minus ones, which corresponds to the base case. The gray-shaded portion of Table 4.2.2.1, Rows 1-8, illustrates the construction of the Plackett and Burman design matrix for X=8, a design appropriate for investigating four to seven parameters.

When using foldover, X additional rows are added to the matrix. The signs of the values in each of these additional rows are the opposite of the corresponding entries in the original matrix. The corresponding row for each of these foldover rows is X rows above that row. Consequently, the last row, Row 2*X, is a row of plus ones. Table 4.2.2.1 shows the complete Plackett and Burman design matrix with foldover. Note that rows 9-16 specifically show the additional foldover rows.

Table 4.2.2.1: Plackett and Burman Design Matrix with Foldover (X=8)

A	B	C	D	E	F	G	Execution Time
+1	+1	+1	-1	+1	-1	-1	9
-1	+1	+1	+1	-1	+1	-1	11
-1	-1	+1	+1	+1	-1	+1	2
+1	-1	-1	+1	+1	+1	-1	1
-1	+1	-1	-1	+1	+1	+1	9
+1	-1	+1	-1	-1	+1	+1	74
+1	+1	-1	+1	-1	-1	+1	7
-1	-1	-1	-1	-1	-1	-1	4
-1	-1	-1	+1	-1	+1	+1	17
+1	-1	-1	-1	+1	-1	+1	76
+1	+1	-1	-1	-1	+1	-1	6
-1	+1	+1	-1	-1	-1	+1	31
+1	-1	+1	+1	-1	-1	-1	19
-1	+1	-1	+1	+1	-1	-1	33
-1	-1	+1	-1	+1	+1	-1	6
+1	+1	+1	+1	+1	+1	+1	112
191	19	111	-13	79	55	239	

For a Plackett and Burman design, the high and low values are slightly different than the high and low values for the one-at-a-time design. A "+1", or high value, for a parameter represents a value that is slightly higher than the range of normal values for that parameter while a "-1", or low value, represents a value that is slightly lower than the range of normal values. By comparison, the high and low values for the one-at-a-design represent the range of values that the parameter normally can take.

It is important to note that the high and low values are not restricted to only numerical values. For example, in the case of branch prediction, the high and low values could be perfect and 2-level branch prediction, respectively. It is also important to note that choosing high and low values that yield too large a range can artificially inflate the parameter's apparent effect. On the other hand, too small a range for a parameter means that the parameter will have very little or no effect on the output. However, having too large a range is better than having too small range because that ensures that that parameter has an effect on the output variable. In any case, the user should carefully choose the high and low values for each parameter that are just outside of the "normal" range of values.

After determining the configurations and performing the simulations, the effect of each parameter is computed by the multiplying the parameter's Plackett and Burman value (+1 or -1) for that configuration by the output variable (e.g. execution time) for that configuration and summing the resulting products across all configurations. For example, given the execution times in the rightmost column in Table 4.2.2.1, the effect of parameter A is computed as follows:

$$\text{Effect}_A = (1 * 9) + (-1 * 11) + (-1 * 2) + \ldots + (-1 * 33) + (-1 * 6) + (1 * 112) = 191$$

By performing the same computation for each parameter in Table 4.2.2.1, the results show that the parameters that have the most effect on the execution time are parameters G, A, and C, in order of their overall impact on performance. Only the magnitude of the effect is important; the sign of the effect is meaningless.

After computing the magnitude of the effect for each parameter, the parameters were ranked based on their magnitudes (1 = most important, X-1 = least important). Since the execution time (in cycles), for the same processor configuration, can be very different across benchmarks, the magnitudes of the effects reflect those differences. Consequently, ranking the parameters by significance allows for comparisons between benchmarks, which would not otherwise be possible due to the large differences in the execution times.

Collectively, the ranks of all parameters form a vector of ranks, one for each benchmark. These vectors are used as the basis for improving the simulation methodology. The following three sections explain how these vectors can be used to improve the way in which processor parameters are chosen, benchmarks are chosen, and the effect of an enhancement is analyzed.

4.3. Processor Parameter Selection

Choosing the processor parameter values for simulation is the third step in the simulation process as described in Section 4.1.1. Choosing a "good" set of values is very important since improperly choosing the value of even a single parameter can significantly affect the simulated speedup of a processor enhancement. For example, simply increasing the reorder buffer size can change the **speedup** of value reuse [Yi02-1] from approximately 20% to approximately 30%.

However, choosing a good set of parameters is extremely difficult since many of the important parameters may interact, thereby compounding the error of selecting a single poor value. Determining which parameters interact requires performing a sensitivity analysis on all of the parameters simultaneously or choosing a select few parameters for detailed study. The problem with the former approach is that simulating all possible combinations is a virtual impossibility. The problem with the latter approach is that in studying only a few parameters, the other parameters have to have constant values. Therefore, if one of the constant parameters significantly interacts with one of the free parameters, then the results of the sensitivity analysis will be distorted.

A Plackett and Burman design solves this problem by quantifying the effect of all single parameters. And since the most significant interactions are the by-product of the most significant single parameters, the user needs only to determine the most important single parameters.

To determine which single parameters are the most significant ones, for each parameter, the rank of that parameter across all benchmarks is first summed together and then the sums are sorted in ascending order. Consequently, the parameter with the lowest average sum-of-ranks corresponds to, across all benchmarks, the parameter that has the most effect on the variation in the execution time. Then, by examining the average sum-of-ranks for each parameter, the computer architect can determine which parameters have the most effect on the execution time and can then carefully choose values for those parameters.

More formally, this dissertation recommends using the following steps as guide when choosing processor parameter values:

1) Determine the most significant processor parameters using a Plackett and Burman design.
    a) Choose low and high values for each of the parameters.
    b) Run and analyze the Plackett and Burman simulations to determine the critical parameters.
2) Iteratively perform sensitivity analyses for each critical parameter using the ANOVA design.
3) Choose final values for the significant parameters based on the results of the sensitivity analyses.
4) Choose the final values for the remaining parameters based on commercial processor values, or some other appropriate source.

Of these four steps, the most important step, by far, is the first step. In this step, the computer architect uses a Plackett and Burman design to determine the most significant parameters. The second step is optional depending on the results of the first step. For example, in Table 4.2.2.1, the two most significant parameters are clearly parameters A and G. Consequently, using an ANOVA design to compare the relative effect of parameters A and G (and any related interactions) with parameter C is a waste of time since parameters A and G are clearly much more important than parameter C. Finally, in the third and fourth steps, the computer architect chooses the final values for all parameters, based on the results of the first two steps and based on commercial parameter values.

Detailed results demonstrating the efficacy of this approach are given in Section 6.3.1.

4.4. Benchmark Selection

Choosing the benchmarks and inputs for simulation is the fourth step in the simulation process. Improperly choosing benchmarks and input sets may affect the results, the conclusions that are drawn, or both. First of all, if a computer architect chooses a set of benchmarks that does not accurately reflect the applications that the proposed processor enhancement targets, then the apparent speedup due to that enhancement may be misleading enough to affect the conclusion that the architect forms. Although the results of those simulations are not wrong, they could still be misleading.

For example, if the proposed enhancement is a prefetching mechanism that seeks to improve the performance of the memory hierarchy, choosing a set of benchmarks that have very regular memory access patterns (as is the case for scientific programs), could produce speedup results that are higher than the speedup results for a more memory-intensive set of benchmarks. In that case, the architect may form a misleading conclusion. Note that it is also possible for a poorly chosen set of benchmarks to understate the speedup results.

However, how does the architect know if two benchmarks are similar or dissimilar? One option is to relying on existing classifications, such as integer versus floating-point, computationally-bound versus memory-bound, or by application type. The problem with this approach is that two benchmarks that are classified differently may have the same characteristics, such as having the same performance bottlenecks in the processor. On the other hand, two benchmarks that are classified to be in the same group may have very dissimilar characteristics. Therefore, simply relying on existing classifications, without verifying the similarity of benchmarks within and across classification groups may still result in a poor choice of benchmarks.

The solution proposed in this dissertation approaches this problem from a different direction. Instead of classifying benchmarks based on their intrinsic characteristics, benchmarks are classified based on what effect they have on the processor. Different benchmarks have different processor-related performance limiting factors (i.e. different bottlenecks). Therefore, two benchmarks that have a similar effect on the processor have most of the same performance bottlenecks and consequently should be grouped together. Since the results of the Plackett and Burman design show which parameters are the most important (or in other words, are the biggest performance bottlenecks), comparing the Plackett and Burman design results of two benchmarks indicates how similar the two benchmarks are, in terms of their performance bottlenecks. Benchmarks that are similar are put into the same group. After grouping all the benchmarks into different groups, selecting the final set of the benchmarks is easy since the architect needs only to select one benchmark from each group.

Detailed results demonstrating the efficacy of this approach are given in Section 6.3.2.

4.5. Analysis of a Processor Enhancement

The last step in the simulation process is analyzing the effect of an enhancement. For most computer architects, the analysis extends only to calculating the speedup of the processor enhancement or, measuring the amount of contention or the decrease in the power consumption, or performing a sensitivity analysis of the key parameters. While these approaches give the architect a high-level picture of the enhancement's effect, it shows only the net effect.

For example, suppose that a processor enhancement yields a speedup of 25%. Also suppose that two parameters (A and B) are the primary performance bottlenecks in the processor. One case is that the enhancement relieves both bottlenecks by about the same amount. Therefore, the bottlenecks due to both parameters still exist, albeit to a lesser degree. However, another case is that the enhancement relieves the bottleneck due to parameter A, but exacerbates the bottleneck due to parameter B. While both cases could result in the same speedup, the two cases arrive at that speedup by different ways. Therefore, understanding what effect the enhancement has on the performance bottlenecks is a crucial step in trying to improve the performance of the enhancement. In other words, high-level metrics such as speedup only show what the enhancement did and not how it got there. Since the "how" affects the "what", it is important to determine the effect that an enhancement has to a greater depth than just with high-level metrics.

Therefore, as a complement to the high-level metrics, this dissertation proposes using the Plackett and Burman design to quantify the effect of an enhancement. The results of a Plackett and Burman design can be used to measure the significance of all processor parameters for the processor with and without the enhancement. Since the significance of a parameter is an indication of how much of a performance bottleneck that parameter is, a change in the significance of a parameter means that that parameter is more or less of a performance bottleneck with that enhancement.

Therefore, to determine what effect an enhancement has on the each parameter, the difference in the average sum-of-ranks for each parameter is computed. Consequently, any parameter that experiences a large change in its average sum-of-ranks after an enhancement is applied has become more of a bottleneck (Before-After < 0) or less of a bottleneck (Before-After > 0).

Detailed results demonstrating the efficacy of this approach are given in Section 6.3.3.

4.6. Summary

This chapter describes three solutions, based on a Plackett and Burman design, which can improve the quality of simulation methodology by adding statistical rigor to the simulation process. The first solution attempts to improve how processor parameter values are chosen, which is the third step of the simulation process. This solution uses two steps. The first step identifies and chooses values for the most significant processor parameters. In the second step, the architect chooses values for the remaining parameters.

The second solution targets the fourth step of the simulation process, benchmark selection. This solution helps the architect select a set of benchmarks by using the Plackett and Burman design results to categorize the benchmarks into different groups. Since benchmarks in the same group have a similar set of performance bottlenecks, to get a wide range of benchmarks, the architect then needs to select only one benchmark from each group.

Finally, the third and final solution seeks to improve the analysis phase of the simulation process. By using the Plackett and Burman design results to rank and then sum the ranks of each parameter, the architect can compare the average sum-of-ranks for each parameter before and after the enhancement is applied to the processor. If the average sum-of-ranks is higher after the enhancement is implemented in the processor, that means that that enhancement decreases the significance of that parameter. In other words, this parameter is now less of a performance bottleneck with this enhancement. A lower average sum-of-ranks means that that parameter is now more of a performance bottleneck.

Chapter 5

Experimental Framework

This chapter describes the experimental framework that was used for the results given in Chapter 6. There are four main sections in this chapter. The first section describes the architecture of *sim-outorder*, the superscalar simulator from the SimpleScalar tool suite [Burger97]. The second and third sections describe the processor and memory configurations that were used to evaluate the performance of Instruction Precomputation, to evaluate the performance of Trivial Computation Simplification and Elimination, and to generate the Plackett and Burman design vector of ranks. Finally Section 5.4 describes the benchmarks and input sets that were used in this dissertation.

5.1. The SimpleScalar Superscalar Simulator

sim-outorder is the base superscalar simulator from the SimpleScalar 3.0 tool suite. *sim-outorder* is an execution-driven, cycle-accurate simulator that models a five-stage processor pipeline (fetch, decode and dispatch, issue and execute, writeback, and commit). Although *sim-outorder* has a relatively low number of pipeline stages as compared to commercial processors, it models a longer pipeline by offering a user-configurable parameter that sets the number of cycles that it takes to flush the pipeline on a branch misprediction and restart instruction execution from the first correct-path instruction. In a superscalar processor, the number of pipeline stages between the branch prediction and the branch execution determines the number of cycles it takes to detect a branch misprediction (plus an additional few cycles to flush the instructions following the mispredicted branch, and to start fetching and executing on the correct path). Consequently, using a variable parameter to set the number of cycles that it takes to detect and recover from a branch misprediction roughly models the effect of a longer pipeline.

In addition to branch prediction, *sim-outorder* also has the following features: multiple instruction fetch and execution, a monolithic reorder buffer, fully-pipelined functional units, a load-store queue, store forwarding, and a two-level cache hierarchy.

Finally, although *sim-outorder* fairly accurately models most processor components, one problem with this version of SimpleScalar is that its memory hierarchy does not fully model two aspects of the memory hierarchy. First, instead of allowing only a limited number of loads to access the memory hierarchy, *sim-outorder* allows an unlimited number of loads to access memory. Also, instead of allowing only a limited of traffic within the memory hierarchy, *sim-outorder* allows an unlimited amount of traffic between the L1 caches and the L2 Cache. The net effect of these two differences is that the memory hierarchy in *sim-outorder* is less of a performance bottleneck than it normally would be in a superscalar processor. As a result, since Instruction Precomputation and the Simplification and Elimination of Trivial Computations attempt to improve the processor's performance (and not the memory hierarchy's performance), the somewhat unrealistic memory performance most likely overestimates the performance improvement of these two enhancements.

5.2. Instruction Precomputation and Trivial Computation Parameters

To evaluate the performance of Instruction Precomputation or the Simplification and

Elimination of Trivial Computations, each proposed technique was added to the base *sim-outorder* simulator. For both techniques, the base processor was a four-way issue width machine.

Table 5.2.1: Key Processor and Memory Parameters for the Performance Evaluation of Instruction Precomputation and the Simplification and Elimination of Trivial Computations

Parameter	Value
Branch Predictor	Combined
Number of Branch History Table Entries	8192
Return Address Stack (RAS) Entries	64
Branch Misprediction Penalty	3 Cycles
Instruction Fetch Queue (IFQ) Entries	32
Reorder Buffer (ROB) Entries	64
Number of Integer ALUs	2
Number of FP ALUs	2
Number of Integer Multipliers	1
Number of FP Multipliers	1
Load-Store Queue (LSQ) Entries	32
Number of Memory Ports	2
L1 D-Cache Size	32 KB
L1 D-Cache Associativity	2-Way
L1 D-Cache Block Size	32 Bytes
L1 D-Cache Latency	1 Cycle
L1 I-Cache Size	32 KB
L1 I-Cache Associativity	2-Way
L1 I-Cache Block Size	32 Bytes
L1 I-Cache Latency	1 Cycle
L2 Cache Size	256 KB
L2 Cache Associativity	4-Way
L2 Cache Block Size	64 Bytes
L2 Cache Latency	12 Cycles
Memory Latency, First Block	60 Cycles
Memory Latency, Following Block	5 Cycles
Memory Bandwidth (Bytes/Cycle)	32
TLB Latency	30 Cycles

Table 5.2.1 shows the values of the key processor and memory parameters that were used for the performance evaluations of both techniques. These parameter values are similar to those found in the Alpha 21264 [Kessler98, Kessler99, Leiholz97, Matson98] and the

MIPS R10000 [Yeager96].

5.3. Plackett and Burman Parameters

As was the case for the performance evaluation of Instruction Precomputation and the Simplification and Elimination of Trivial Computations, the base simulator was *sim-outorder*. *sim-outorder* was modified to include user configurable instruction latencies and throughputs. The value of the instruction throughput is the number of cycles that must separate the start of two instructions on the same functional unit. For example, if the instruction throughput for the floating-point ALUs is two cycles and if instruction A starts executing on a floating-point ALU at cycle 1000, instruction B cannot start executing on that floating-point ALU until cycle 1002. However, instruction B can start executing on any other floating-point ALU immediately.

It is important to mention that *sim-outorder* was used instead of the validated Alpha 21264 simulator [Desikan01] for three reasons. The first reason is that this is a methodology study and not an architecture or performance-only study. Consequently, since the simulation results serve only to illustrate certain key points, the choice of a specific simulator does not affect the point that is being made. The second reason is that the Alpha simulator contains many parameters that are specific to the Alpha architecture while the basic SimpleScalar simulator models a vanilla superscalar processor. Therefore, to avoid the risk of producing results that are particular to the Alpha 21264 processor, *sim-outorder* was chosen to be the base simulator. The third reason is that the SimpleScalar simulator is itself a widely used simulator. Therefore, using this simulator has the extra benefit of producing results that are beneficial to the SimpleScalar community.

As stated in Chapter 4, the parameter values that were used in these simulations should be slightly higher and lower than the normal range of values to allow the Plackett and Burman design to work most efficiently. As a result, the final values for each parameter are not values that would be actually present in commercial processors nor are they supposed to represent a potential value. **Rather, the values were deliberately chosen to be values that were slightly higher and lower than the range of "reasonable" values.**

Table 5.3.1: Processor Core Parameters and Their Plackett and Burman Values

Processor Core Parameter	Low Value	High Value
Instruction Fetch Queue (IFQ) Entries	4	32
Branch Predictor	2-Level	Perfect
Branch Predictor Misprediction Penalty	10 Cycles	2 Cycles
Return Address Stack (RAS) Entries	4	64
Branch Target Buffer (BTB) Entries	16	512
Branch Target Buffer (BTB) Associativity	2-Way	Fully-Associative
Speculative Branch Update	In Commit	In Decode
Decode, Issue, and Commit Width	4-Way	
Reorder Buffer (ROB) Entries	8	64
Load-Store Queue (LSQ) Entries	0.25 * ROB	1.0 * ROB
Memory Ports	1	4

Table 5.3.2: Functional Units Parameters and Their Plackett and Burman Values

Functional Unit Parameter	Low Value	High Value
Integer ALUs	1	4
Integer ALU Latencies	2 Cycles	1 Cycle
Integer ALU Throughputs	1	
FP ALUs	1	4
FP ALU Latencies	5 Cycles	1 Cycle
FP ALU Throughputs	1	
Integer Mult/Div Units	1	4
Integer Multiply Latency	15 Cycles	2 Cycles
Integer Divide Latency	80 Cycles	10 Cycles
Integer Multiply Throughput	1	
Integer Divide Throughput	Equal to the Integer Divide Latency	
FP Mult/Div Units	1	4
FP Multiply Latency	5 Cycles	2 Cycles
FP Divide Latency	35 Cycles	10 Cycles
FP Square Root Latency	35 Cycles	15 Cycles
FP Multiply Throughput	Equal to the FP Multiply Latency	
FP Divide Throughput	Equal to the FP Divide Latency	
FP Square Root Throughput	Equal to the FP Square Root Latency	

Choosing values in this way allows the Plackett and Burman design to more accurately determine the effect of each parameter on the processor's performance.

Table 5.3.3: Memory Hierarchy Parameters and Their Plackett and Burman Values

Memory Hierarchy Parameter	Low Value	High Value
L1 I-Cache Size	4 KB	128 KB
L1 I-Cache Associativity	1-Way	8-Way
L1 I-Cache Block Size	16 Bytes	64 Bytes
L1 I-Cache Replacement Policy	Least Recently Used (LRU)	
L1 I-Cache Latency	4 Cycles	1 Cycle
L1 D-Cache Size	4 KB	128 KB
L1 D-Cache Associativity	1-Way	8-Way
L1 D-Cache Block Size	16 Bytes	64 Bytes
L1 D-Cache Replacement Policy	Least Recently Used (LRU)	
L1 D-Cache Latency	4 Cycles	1 Cycle
L2 Cache Size	256 KB	8192 KB
L2 Cache Associativity	1-Way	8-Way
L2 Cache Block Size	64 Bytes	256 Bytes
L2 Cache Replacement Policy	Least Recently Used (LRU)	
L2 Cache Latency	20 Cycles	5 Cycles
Memory Latency, First Block	200 Cycles	50 Cycles
Memory Latency, Following Blocks	0.02 * Memory Latency, First Block	
Memory Bandwidth	4 Bytes	32 Bytes
I-TLB Size	32 Entries	256 Entries
I-TLB Page Size	4 KB	4096 KB
I-TLB Associativity	2-Way	Fully-Associative
I-TLB Latency	80 Cycles	30 Cycles
D-TLB Size	32 Entries	256 Entries
D-TLB Page Size	Same as I-TLB Page Size	
D-TLB Associativity	2-Way	Fully-Associative
D-TLB Latency	Same as I-TLB Latency	

Several parameters in these three tables are shaded in gray. For these parameters, the low and high values cannot be chosen completely independently of the other parameters due to the mechanics of a Plackett and Burman design. The problem occurs when one of the shaded parameters is set to its high or low value and the parameter it is related to is set to the opposite value. In those configurations, the combination of values for those parameters leads to a situation that either does not make sense or would not actually occur in a real processor. For example, if the number of LSQ entries were chosen independently of the number of ROB entries, some of the configurations would have an 8-entry reorder buffer and a 64-entry LSQ. Since the total number of in-flight instructions cannot exceed the number of reorder buffer entries, the maximum number of filled LSQ entries could never exceed eight. Therefore, to avoid the above situation and other similar ones, the specific values used in the simulations for all gray-shaded parameters are based on their related parameter. This allows the gray-shaded parameters to have a meaningful effect on the output, instead of being artificially constrained by another parameter.

Finally, all parameter values were based on a four-way issue processor. While the issue width is a very important parameter, the issue width was fixed at four for two reasons. The first reason is the same as the reason given above for the parameters shaded in gray. That is, if the issue width were set to its low value while the number of functional units were set to their high values, then some of the functional units would never be used since simulator allows only four new instructions to start executing per cycle. Second, several four-way issue commercial processors exist and these processors are fairly well documented. Therefore, to obtain a good range of values for each parameter, the issue width was chosen to reflect the issue width of the processors with good documentation. However, fixing the issue width to a constant value does not affect the conclusions drawn from these simulations in any way. It merely removes the issue width as one of variable parameters.

5.4. Benchmarks and Input Sets

To evaluate the performance of Instruction Precomputation, 12 benchmarks from the SPEC CPU 2000 benchmark suite [Henning00], shown in Table 5.4.1, were used. The leftmost column in Table 5.4.1 gives the benchmark name. The second and fourth columns show the two input sets that were used for each benchmark, while the third and fifth columns show the dynamic instruction count, in millions of instructions, for that benchmark and input set combination. In the second and fourth columns, the specific file is listed when there is more than one input set of that type. The input set in the second and third columns is arbitrarily named "Input Set A" while the other input set is likewise named "Input Set B". Chapter 6 explains why two different input sets are used to evaluate the performance of Instruction Precomputation.

Table 5.4.1: Selected SPEC CPU 2000 Benchmarks and Input Sets (Dynamic Instruction Count in Millions of Instructions)

Benchmark	Input Set A Name	Instr. (M)	Input Set B Name	Instr. (M)
gzip	Small (*smred.log*)	526.4	Medium (*mdred.log*)	531.4
vpr-Place	Medium	216.9	Small	17.9
vpr-Route	Medium	93.7	Small	5.7
gcc	Medium	451.2	Test	1638.4
mesa	Large	1220.6	Test	3239.6
art	Large	2233.6	Test	4763.4
mcf	Medium	174.7	Small	117.0
equake	Large	715.9	Test	1461.9
ammp	Medium	244.9	Small	68.1
parser	Medium	459.3	Small	215.6
vortex	Medium	380.3	Large	1050.0
bzip2	Large (*lgred.source*)	1553.4	Test	8929.1
twolf	Test	214.6	Large	764.9

These benchmarks were chosen because they were the only ones that had MinneSPEC [KleinOsowski02] large reduced input sets available at the time. Since the benchmark *vpr* uses two "sub-input" sets, *Place* and *Route*, the results for each are listed separately.

To evaluate the performance of Trivial Computation Simplification and Elimination, the same 12 benchmarks from the SPEC CPU 2000 benchmark suite were used. But instead of using two input sets, as was the case for Instruction Precomputation, only one input set, Input Set A, was used. The MediaBench benchmarks [Lee97] listed in Table 5.4.2 were also used to evaluate the performance that can be achieved by Simplifying and Eliminating Trivial Computations.

Finally, to evaluate the efficacy of the Plackett and Burman design in generating a vector of ranks that is useful in improving simulation methodology, the same 12 benchmarks from the SPEC CPU 2000 benchmark suite were used. However, in this case, instead of using small, medium, large, and test input sets, to reduce the differences in the total dynamic instruction count, only large input sets were used. Table 5.4.3 shows the benchmarks, input sets, and their dynamic instruction count.

Table 5.4.2: Selected MediaBench Benchmarks and Input Sets (Dynamic Instruction Count in Millions of Instructions)

Benchmark	Input Set Name	Instr. (M)
adpcm-Decode	clinton.adpcm	5.4
adpcm-Encode	clinton.pcm	6.5
epic-Compress	test_image.pgm	52.7
epic-Uncompress	test.image.pgm.E	6.8
g721-Decode	clinton.g721	269.4
g721-Encode	clinton.pcm	276.9
mpeg2-Decode	options.par	170.9
mpeg2-Encode	mei16v2.m2v	1133.8
pegwit-Decrypt	pegwit.dec	18.2
pegwit-Encrypt	pgptest.plain	31.8
pegwit-Pub-Key	my.sec	12.7

Table 5.4.3: Selected SPEC CPU 2000 Benchmarks with the Large Input Set (Dynamic Instruction Count in Millions of Instructions)

Benchmark	Input Set Name	Instr. (M)
gzip	Large (*lgred.graphic*)	1364.2
vpr-Place	Large	1521.7
vpr-Route	Large	881.1
gcc	Large	4040.7
mesa	Large	1217.9
art	Large	2181.1
mcf	Large	601.2
equake	Large	713.7
ammp	Large	1228.1
parser	Large	2721.6
vortex	Large	1050.2
bzip2	Large (*lgred.graphic*)	2467.7
twolf	Large	764.6

In this dissertation, all benchmarks were compiled at optimization level -O3 using the SimpleScalar version of the gcc compiler (version 2.6.3). All benchmarks were compiled to target the PISA instruction-set, which is a MIPS-like instruction-set. Finally, the benchmarks ran to completion without fast-forwarding.

Chapter 6

Performance Evaluation

The results in this chapter are divided into three main sections that reflect the contents of Chapters 2, 3, and 4. Section 6.1 presents the performance results for Instruction Precomputation while Section 6.2 does the same for the Simplification and Elimination of Trivial Computations. Finally, the results in Section 6.3 show how the Plackett and Burman design can be used to improve selected steps of the simulation process.

6.1. Instruction Precomputation Performance Results

The performance results for Instruction Precomputation are divided into five groups. The results of the first group present the upper-bound performance results of Instruction Precomputation; the upper-bound occurs when the same input set is used for both profiling and performance simulation. To accomplish this, the benchmark was first profiled with Input Set A to find the highest frequency unique computations and then evaluated the performance of Instruction Precomputation with that benchmark by again using Input Set A. The shorthand notation is Profile A, Run A.

However, since it extremely unlikely that the same input set that is used to profile the benchmark will also the same input set that will be used to run the benchmark, Section

6.1.2 presents the results for when two different input sets are used, i.e. Profile B, Run A. This group of results represents the typical case.

Since the highest frequency unique computations from two different input sets may be very different from each other and since this could affect the performance of Instruction Precomputation, one solution is to combine the two sets of unique computations together. In other words, instead of profiling only a single input set, this solution uses the superset of profiling results from two input sets. This test case is known as Profile AB, Run, A.

As described in Chapter 2, one of the ways that Instruction Precomputation improves a processor's performance is by reducing the execution latency of the instructions that match a unique computation in the Precomputation Table (PT). Consequently, dynamically removing instructions with the longest execution latencies yields the largest performance gain. Therefore, choosing unique computations purely by their frequency of execution may not yield the largest performance gain since some low frequency unique computations may have longer execution latencies. As a result, the results in Section 6.1.4 compare the performance of Instruction Precomputation when using the highest frequency unique computations and when using the unique computations with the highest frequency/latency product.

The final group of results compares the performance improvement of Instruction Precomputation against the performance improvement of value reuse. These results are given in Section 6.1.5.

Note: When not explicitly stated, any specific results are assumed to be from the Profile B, Run A test case.

6.1.1. Upper-Bound – Profile A, Run A

Since Instruction Precomputation consists of two key steps – profiling and the dynamic removal of redundant computations – using the same input set for both represents the upper-bound on performance. This case is the upper-bound because using the same input set to profile and run a benchmark will determine the set of unique computations that account for the largest possible percentage of instructions for that benchmark and input set. Therefore, since removing the largest possible number of instructions can minimize the execution time, using the same input set to profile and run the benchmark is the upper-bound.

The following figure, Figure 6.1.1.1, shows the speedup due to Instruction Precomputation, for 16 to 2048 PT entries, when Input Set A is used both for profiling and for execution. For comparison, the speedup due to using a L1 D-Cache that is twice as large as the L1 D-Cache of the base processor is included. This result, labeled "Big Cache", represents the alternative of using the chip area for something other than the PT. The total capacity of this cache is 64 KB.

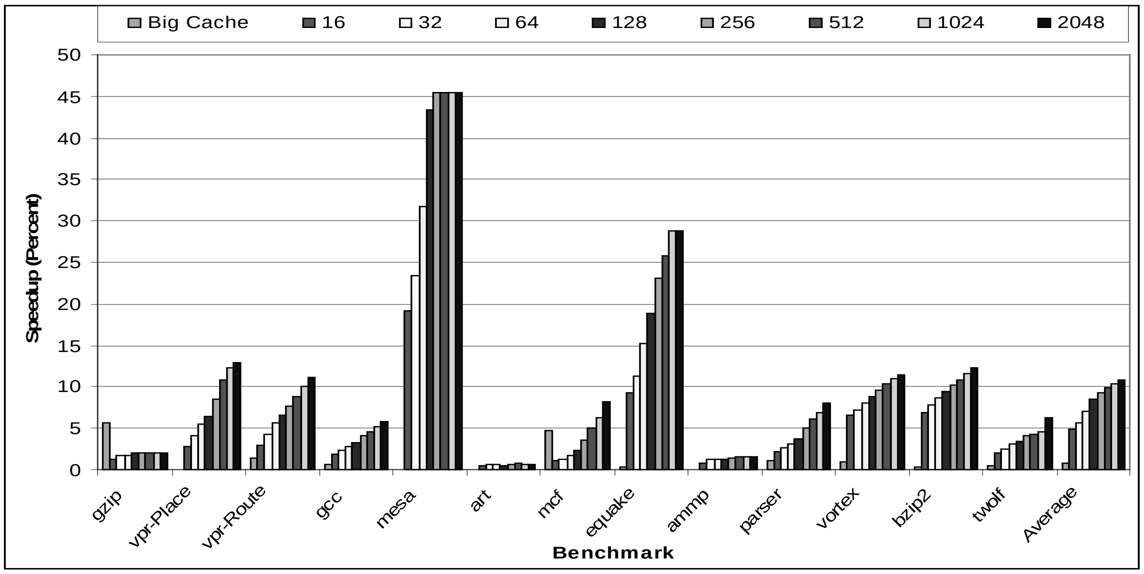

Figure 6.1.1.1: Speedup Due to Instruction Precomputation; Profile Input Set A, Run Input Set A, Frequency

Figure 6.1.1.1 shows that the average upper-bound speedup due to using a 16-entry PT is 4.82% for these 13 benchmarks (counting *vpr-Place* and *vpr-Route* separately) while the speedup due to using a 2048-entry PT is 10.87%. Across all benchmarks, the range of speedups for a 2048-entry PT is 0.69% (*art*) to 45.05% (*mesa*). The average speedup results demonstrate that the upper-bound performance improvement due to Instruction Precomputation is fairly good for all table sizes.

Instruction Precomputation is very effective in decreasing the execution time for two benchmarks, *mesa* and *equake*, even for very small PTs. For *mesa*, the speedup for a 16-entry PT is 19.19% while the speedup for a 2048-entry PT is 45.03%. For *equake*, the speedups range from 9.35% to 28.71% for the same PT sizes. The reason that Instruction Precomputation is particularly effective in reducing the execution time of these benchmarks is because the Top 2048 unique computations account for a very large percentage of the total dynamic instructions. Table 3.2.1.1 shows that the 2048 highest frequency unique computations account for 44.49% and 37.87% of the total dynamic instructions count in *mesa* and *equake*, respectively.

For *mesa*, the speedup levels off for Precomputation Tables larger than 128-entries. For *equake*, the speedup levels off after 1024-entries. The reason for this is that the unique computations that are in the bottom "half" of the Precomputation Table account for a lower percentage of dynamic instructions than the unique computations in the top half. For example, in *mesa*, the 64 highest frequency unique computations account for 34.96% of the total dynamic instructions while the next 64 highest frequency unique computations account for only an additional 7.69%. As a result, doubling the Precomputation Table size yields smaller and smaller performance gains.

The average speedup due to using the larger L1 D-Cache is 0.74%. By comparison, the upper-bound speedup when using a 2048-entry Precomputation Table (approximately 26 KB, assuming a one byte opcode, two four byte input operands, and a four byte output value) averages 10.87%. In other words, using approximately the same chip area for a Precomputation Table instead of for a larger L1 D-Cache improves the performance of the base processor by an additional 10%.

6.1.2. Different Input Sets – Profile B, Run A

While the previous sub-section showed that the upper-bound speedup due to Instruction Precomputation was approximately 10% for a 2048-entry PT, the actual, i.e. achievable, speedup may be much lower if the highest frequency unique computations from two different input sets are very different. Therefore, for Instruction Precomputation to be useful, the highest frequency unique computations in one input set have to be among the highest frequency unique computations in another input set. If not, then Instruction Precomputation may not be a very practical microarchitectural enhancement.

Figure 6.1.2.1 shows the speedup due to Instruction Precomputation when using Input Set B for profiling and Input Set A for execution. The figure shows the speedup using 16-entry to 2048-entry PT tables that hold the highest frequency unique computations.

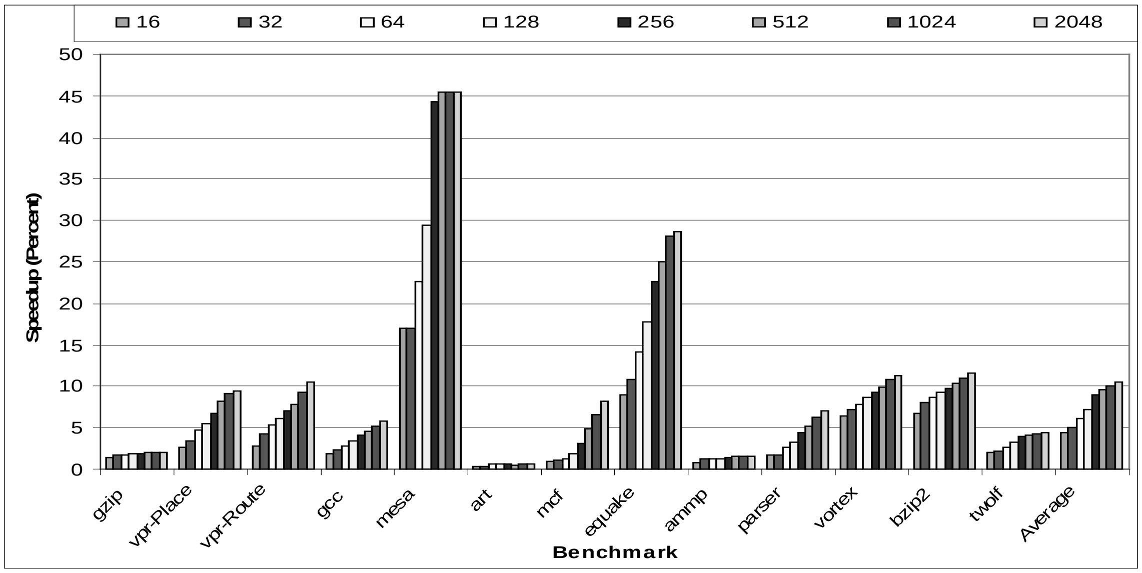

Figure 6.1.2.1: Speedup Due to Instruction Precomputation; Profile Input Set B, Run Input Set A, Frequency

As shown in Figure 6.1.2.1, the average speedup ranges from 4.47% for a 16-entry PT to 10.53% for a 2048-entry PT. By comparison, the speedup for Profile A, Run A ranges from 4.82% to 10.87% for the same table sizes. These results show that the average speedups for Profile B, Run A are very close to the upper bound speedups for the endpoint PT sizes. In addition, with one key exception, *mesa*, the speedups for each benchmark, for a given PT size, are similar.

Table 6.1.2.1 shows the speedups for *mesa* when using Input Set A for execution while using Input Sets A and B for profiling.

Table 6.1.2.1: Speedup Due to Instruction Precomputation for *mesa*; Profile Input Set A, Run Input Set A versus Profile Input Set B, Run Input Set A, Frequency

PT Entries	Profile A, Run A	Profile B, Run A	Difference
16	19.19	16.91	2.28
32	23.49	16.92	6.56
64	31.77	22.59	9.18
128	43.40	29.45	13.95
256	45.40	44.37	1.03
512	45.40	45.40	0.00
1024	45.40	45.40	0.00
2048	45.40	45.40	0.00

The largest difference between the two sets of speedups is for the 32-entry, 64-entry, and 128-entry PTs. Those differences completely disappear for PT sizes larger than 256 entries. The reason for the speedup differences and their subsequent disappearance is that the highest frequency unique computations for Input Set B, do not have as high a frequency of execution for Input Set A. Therefore, until the highest frequency unique computations for Input Set A are included in the PT (for PT sizes larger than 128), the speedup for Profile B, Run A for *mesa* will be much lower than the upper-bound speedup.

On the other hand, since the Profile B, Run A speedup is very significant for the smaller table sizes, some unique computations have a fairly high frequency of execution for both input sets.

Finally, for a few benchmarks and table sizes, the speedup is actually slightly higher than the "upper-bound" for Profile B, Run A. While these differences in speedups are fairly small (less than 0.3% difference), this result shows that the highest frequency unique computations for one input set may have an even higher frequency of execution in another input set.

The key conclusion of this sub-section is that the performance of Instruction Precomputation is generally not affected by the specific input set since the Profile B, Run A speedups are very close to the upper-bound speedups. This conclusion is not particularly surprising since Table 2.3.1 showed that a large number of the highest frequency unique computations occur for multiple input sets. Therefore, although different input sets may be used for profiling and execution, since the same high frequency unique computations occur for multiple input sets, Instruction Precomputation is an effective method of improving the processor's performance.

6.1.3. Combination of Input Sets – Profile AB, Run A

While the performance of Instruction Precomputation is generally not affected by the specific input set, the speedup when different input sets are used for profiling and execution affects at least one benchmark (*mesa*). Although the difference in speedups disappeared for larger PT sizes, sufficient chip area may not exist to allow for a larger table. One potential solution to this problem is to combine two sets of unique computations – which are the product of two different input sets – to form a single set of unique computations that may be more representative of all input sets.

To form this combined set of unique computations, unique computations were selected from the 2048 highest frequency unique computations from Input Set A and Input Set B. Excluding duplicates, the unique computations that were chosen for the final set were the ones that accounted for the largest percentage of dynamic instructions for their input set. In other words, when combining sets of unique computations, only the unique computations that represent the largest percentage of instructions are chosen for the final set.

Figure 6.1.3.1 shows the speedup due to Instruction Precomputation when using Input Sets A and B for profiling and Input Set A for execution. The figure shows the speedup using 16-entry to 2048-entry PT tables that hold the highest frequency unique computations from each input set.

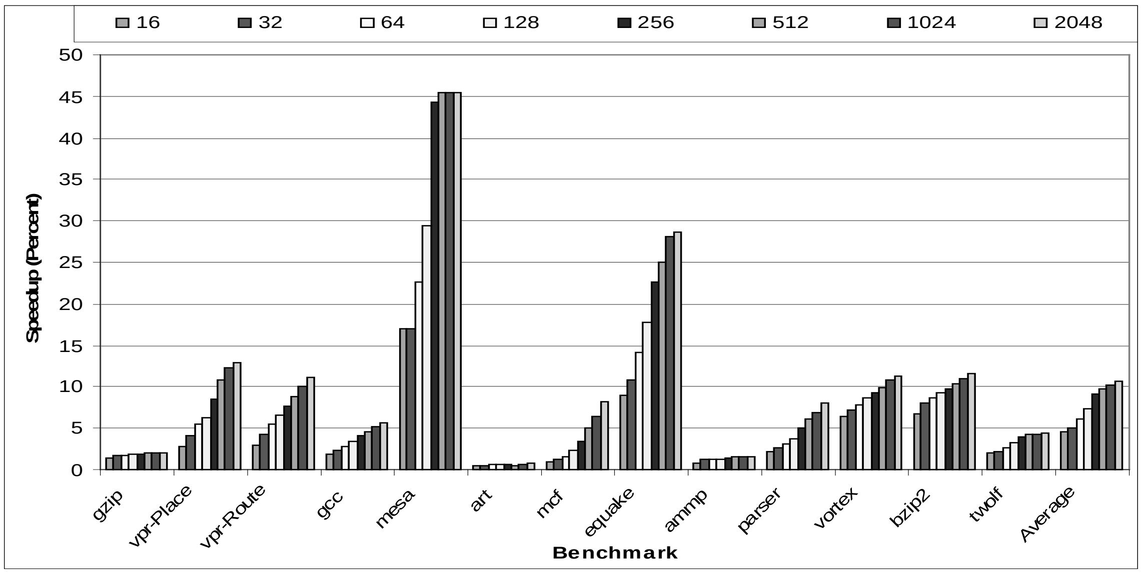

Figure 6.1.3.1: Speedup Due to Instruction Precomputation; Profile Input Set AB, Run Input Set A, Frequency

As shown in Figure 6.1.2.1, the average speedup ranges from 4.53% for a 16-entry PT to 10.71% for a 2048-entry PT. By comparison, the speedup for Profile A, Run A ranges from 4.82% to 10.87% for the same table sizes, while the speedup for Profile B, Run A ranges from 4.47% to 10.53%. Therefore, while the average speedups for Profile AB, Run A are closer to the upper bound speedups, using the combined set of unique computations provides only a slight, but measurable, performance improvement over the Profile B, Run A speedups.

The main reason that the speedups for Profile AB are only slightly higher than the speedups for Profile B is that the highest frequency unique computations from Input Set A are very similar to their counterparts from Input Set B, for most benchmarks. Table 2.3.1 (given below for convenience) shows that with the exceptions of *vpr-Place, mesa*, and *twolf*, more than half of the unique computations are among the highest frequency unique computations for Input Set A and Input Set B. Therefore, since most of the unique computations in the final set are common to both input sets, it is not surprising to see that the speedup is only slightly higher.

Table 6.1.3.1: Number of Unique Computations that are Present in Two Sets of 2048 of the Highest Frequency Unique Computations from Two Different Input Sets

Benchmark	In Common	Percentage
gzip	2028	99.0%
vpr-Place	527	25.7%
vpr-Route	1228	60.0%
gcc	1951	95.3%
mesa	589	28.8%
art	1615	78.9%
mcf	1675	81.8%
equake	1816	88.7%
ammp	1862	90.9%
parser	1309	63.9%
vortex	1298	63.4%
bzip2	1198	58.5%
twolf	397	19.4%

In the case of *vpr-Place*, although the percentage of unique computations is relatively low (25.7%), the average speedup using the 2048 PT entries for the Profile B, Run A case is still fairly high (9.41%). However, after combining the two sets of unique computations together, the average speedup using the 2048 PT entries for the Profile AB, Run is significantly higher (12.97%). These results show that combining two sets of unique computations together to form a single set, which is more representative of all input sets, produces a higher speedup.

For *mesa*, the percentage of common unique computations is even lower (28.8%) than the speedup for Profile B, Run A (45.40%) using 2048 PT entries. However, the 256 highest frequency unique computations account for 44.3% of the dynamic instructions in Input Set A; by comparison, the 2048 highest frequency unique computations account for only 44.5%. Therefore, for this benchmark, very few unique computations need to be common to both input sets for a significant percentage of the dynamic instructions to be accounted for.

For *twolf*, the speedups for the Profile B, Run A case and the Profile AB, Run A case are very close (4.38% for Profile B, Run A and 4.40% for Profile AB, Run A), which would seem to indicate that the highest frequency unique computations are generally dissimilar. This conclusion is confirmed by the fact that *twolf* has the lowest percentage of unique computations that are common across both input sets. In other words, the reason that only 19.4% of the top 2048 highest frequency unique computations are common across input sets is because the highest frequency unique computations are generally different. One by-product of this characteristic for *twolf* is that combining sets of unique computations does not significantly improve the performance.

Finally, although Profile AB yields slightly higher speedups, the downside of this approach is that the compiler needs to profile two input sets. Therefore, from a cost-benefit point-of-view, an additional 0.29% (16 PT entries) to 0.15% (2048 PT entries) average speedup does not offset the cost of profiling two input sets and combining their unique computations together.

6.1.4. Frequency versus Frequency and Latency Product

The last three sub-sections presented the speedup results for Instruction Precomputation based on choosing the highest frequency unique computations. Although the set of the highest frequency unique computations represents the largest percentage of dynamic instructions, those instructions could have a lower impact on the execution time than their numbers would suggest since many of those dynamic instructions have a single-cycle execution latency. Therefore, instead of simply choosing unique computations based only on their frequency of execution, choosing the unique computations that have the highest frequency/latency product (F/LP) could yield a larger performance gain. Since the execution latency of a unique computation is strictly determined by its opcode, to compute the F/LP for a unique computation, one need only to multiply the frequency of that unique computation by its execution latency. Choosing unique computations based on their F/LP, instead of solely by their frequency, may yield a larger performance gain since the highest F/LP unique computations may potentially account for more execution cycles than the highest frequency unique computations.

Figure 6.1.4.1 shows the speedup due to Instruction Precomputation when using Input Set B for profiling and Input Set A for execution. The figure shows the speedup using 16-entry to 2048-entry PT tables that hold the unique computations with the highest F/LP. The F/LP is computed by multiplying the frequency of execution by the execution latency for that instruction (fixed latencies for each opcode).

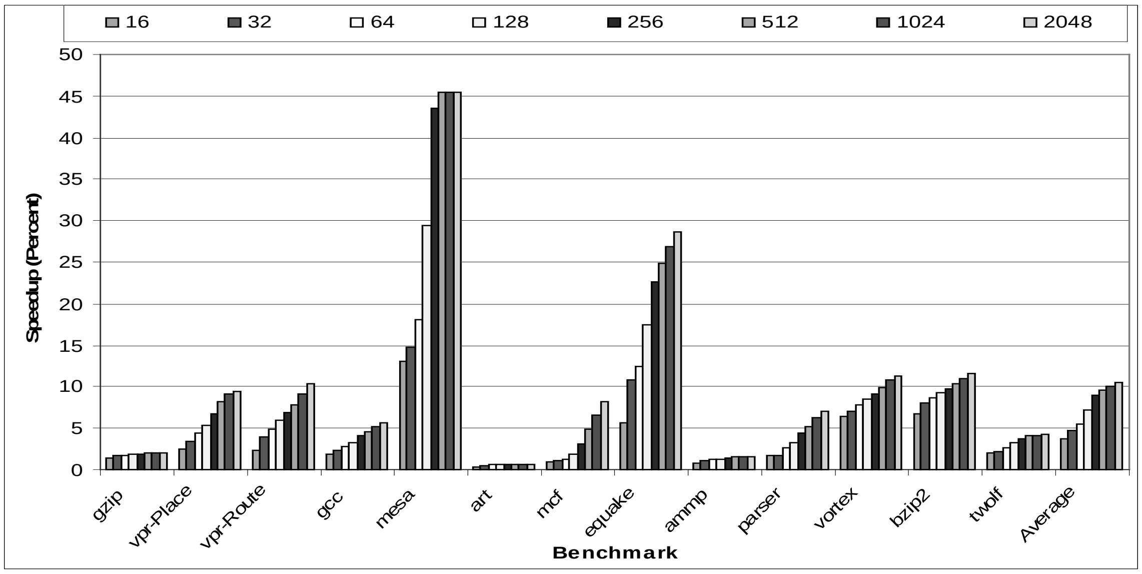

Figure 6.1.4.1: Speedup Due to Instruction Precomputation for the Highest Frequency and Latency Product Unique Computations; Profile B, Run A

As shown in Figure 6.1.4.1, the average speedup ranges from 3.85% for a 16-entry PT to 10.49% for a 2048-entry PT. In most cases, the speedup when using the highest F/LP unique computations is slightly lower than the speedups when using the highest frequency unique computations. While this result may seem a little counterintuitive, the explanation for this result is because the processor can issue and execute instructions out-of-order. By issuing and executing instructions out-of-order, the processor is able to hide the latency of longer-latency instructions by issuing or executing other instructions.

While the processor is able to tolerate the effect of longer execution latencies, it is somewhat limited by the number of functional units. By using the highest F/LP unique computations, fewer instructions are dynamically eliminated (as compared to when using the highest frequency unique computations), thus increasing the number of instructions that require a functional unit. As a result, any performance improvements gained by using the highest F/LP unique computations are partially offset by the higher amount of functional unit contention.

6.1.5. Performance of Instruction Precomputation versus Value Reuse

As described in Chapters 1 and 2, value reuse is a microarchitectural technique that dynamically removes redundant computations from the processor's pipeline by forwarding their output values from the value reuse table (VRT) [Sodani97, Sodani98]. The key difference between value reuse and Instruction Precomputation is that value reuse dynamically updates the VRT while the Precomputation Table is statically managed by the compiler. Since the two approaches are quite similar, this sub-section compares the speedup results of Instruction Precomputation with the speedup results for value reuse. Figure 6.1.5 shows the average speedup results for value reuse when executing Input Set A.

The configuration of the base processor is the same as the base processor configuration for Instruction Precomputation. The number of value reuse table entries varies from 16 to 2048 entries. Each entry holds the opcode, input operands, and output value of a redundant computation. When the program begins execution, all entries of the VRT are invalid. During program execution, the opcode and input operands of each dynamic instruction are compared to the opcodes and input operands in the VRT. As with Instruction Precomputation, when the opcodes and input operands match, the VRT forwards the output value to that instruction and it is removed from the pipeline. Otherwise, the instruction executes normally. Entries in the VRT are replaced only when the VRT is full. In that event, the least-recently used (LRU) entry is replaced.

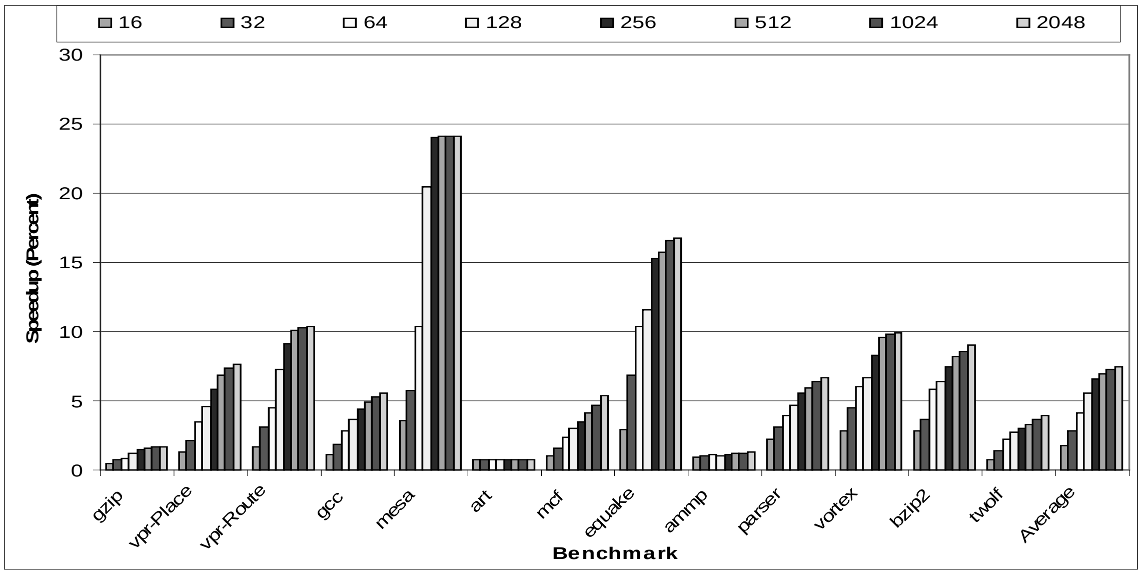

Figure 6.1.5.1: Speedup Due to Value Reuse; Run A

As shown in Figure 6.1.5.1, the average speedup ranges from 1.82% for a 16-entry VRT to 7.43% for a 2048-entry VRT while the speedup for Instruction Precomputation (Profile B, Run A) ranges from 4.47% to 10.53% for the same number of table entries. Therefore, for all table sizes, Instruction Precomputation has a higher speedup. This difference is especially noticeable for the 16-entry tables.

Since value reuse constantly replaces the LRU entry with the opcode and input operands of the latest dynamic instruction, the VRT can easily be filled with low frequency unique computations when it is very small. By contrast, Instruction Precomputation is most effective when the table size is small since each entry in the PT accounts for a large percentage of dynamic instructions. The reason that the VRT could be filled with lower frequency unique computations while the PT is filled with highest frequency ones is because Instruction Precomputation selects the highest frequency unique computations based on profiling while the value reuse hardware effectively assumes that the frequency of recently executed computations is higher than the frequency of the LRU entries.

In conclusion, while value reuse is very limited by the VRT size, Instruction Precomputation is especially effective. Overall, the average speedup for Instruction Precomputation is significantly higher than the average speedup for value reuse for all table sizes. This result shows that using the compiler to choose the highest frequency unique computations is better than using hardware to do the same, especially for smaller table sizes.

6.1.6. Summary

The results in this section show that Instruction Precomputation can significantly improve the processor's performance, by an average of 10.53% and up to 45.40% for a 2048-entry table. For all table sizes, the average speedups for the Profile B, Run A; Profile AB, Run A; and the F/LP configurations are quite close to the upper bound speedup. Finally, the results in the last sub-section showed that the average speedup due to Instruction Precomputation was much higher than the average speedup for value reuse, especially for the smaller table sizes.

6.2. Performance Results for Exploiting Trivial Computations

This section presents the speedup results when execution of the trivial computations in the benchmark are simplified or eliminated. Section 6.2.1 gives the speedup results for the baseline processor configuration, shown in Table 5.2.1. Section 6.2.2 gives the speedup results when additional functional units are added to the baseline processor configuration to remove any functional unit constraints. In each section, the speedup results for the SPEC and MediaBench benchmarks are given in separate figures.

6.2.1. Realistic Processor Configuration

This processor configuration is labeled as the “realistic” processor configuration since it closely resembles the configurations of the MIPS R10000 and Alpha 21264. Figures 6.2.1.1 and 6.2.1.2 show the performance improvement by simplifying and eliminating trivial computations for the SPEC and MediaBench benchmarks, respectively. The rightmost bar in each figure shows the average speedup across the benchmarks from that benchmark suite.

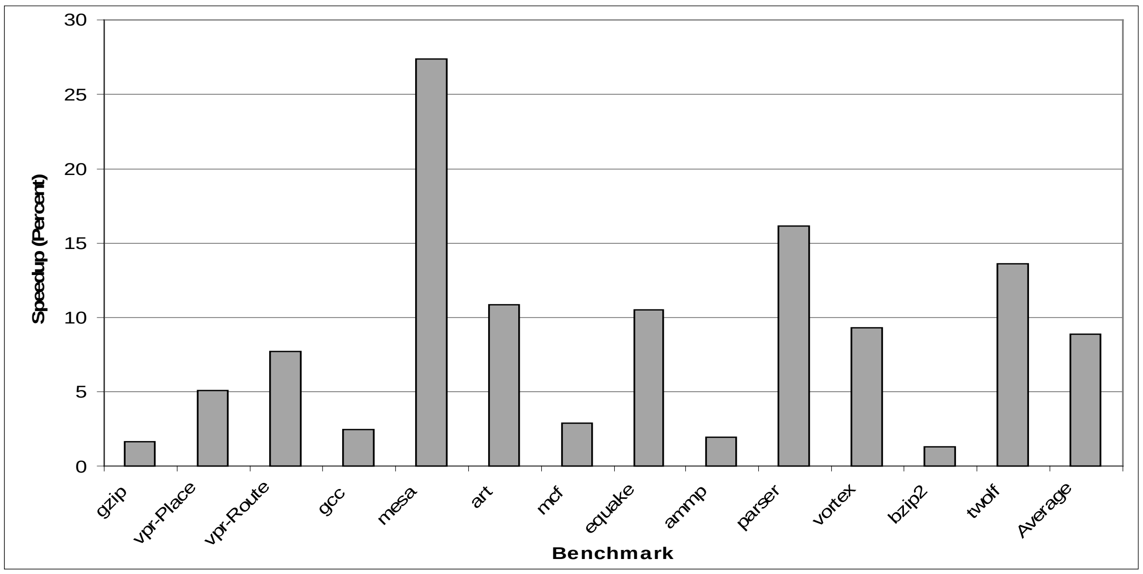

Figure 6.2.1.1: Speedup Due to the Simplification and Elimination of Trivial Computations for Selected SPEC 2000 Benchmarks, Realistic Processor Configuration

Figure 6.2.1.1 shows that simplifying and eliminating trivial computations yields speedups of 1.31% (*bzip2*) to 27.36% (*mesa*), with an average of 8.86% for the 13 SPEC benchmarks. The speedups for the MediaBench benchmarks are 2.97% (*epic-Compress*) to 13.97% (*epic-Uncompress*), with an average of 4.00%. These results show that exploiting trivial computations can significantly decrease the execution time of the SPEC benchmarks while moderately decreasing the execution time of the MediaBench benchmarks.

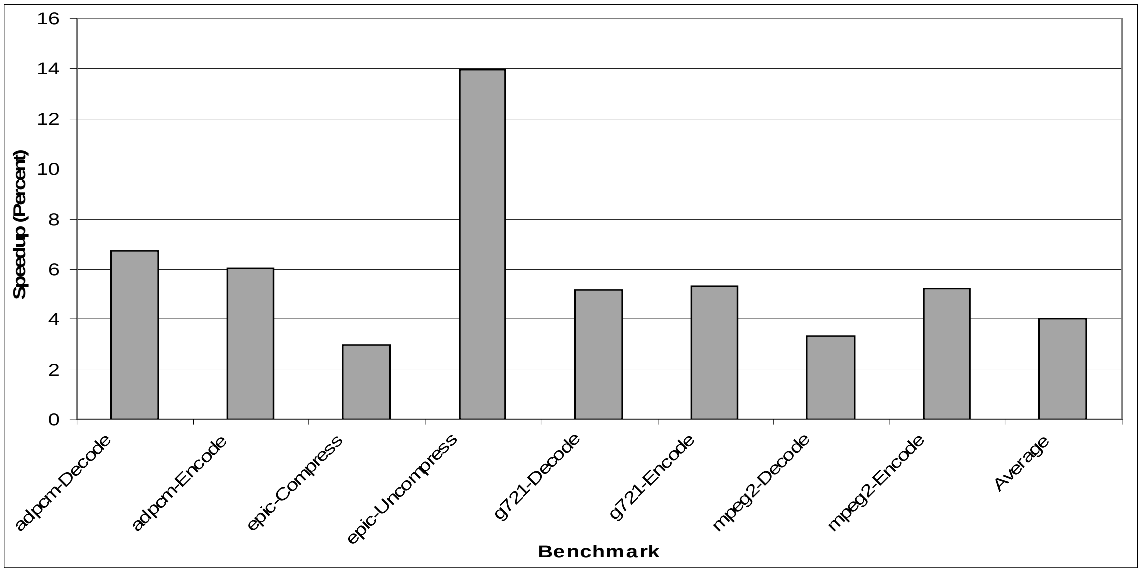

Figure 6.2.1.2: Speedup Due to the Simplification and Elimination of Trivial Computations for Selected MediaBench Benchmarks, Realistic Processor Configuration

The SPEC benchmarks have a higher average speedup than the MediaBench benchmarks for two reasons. First, the SPEC benchmarks have a higher percentage of instructions that could be trivial computations. Second, in the SPEC benchmarks, a higher percentage of those eligible instructions are trivial computations. The net effect of these two reasons is that, in the SPEC benchmarks, a higher percentage of the total instructions are trivial computations.

For each benchmark, in general, the speedup due to simplifying and eliminating trivial computations is correlated to the percentage of the total instructions that are trivial computations. For example, *mesa* has the highest speedup (27.36%) and also the largest percentage of instructions that are trivial computations (24.74%). On the other hand, *gzip* has the lowest percentage of instructions that are trivial computations (1.87%) while having the second-lowest speedup (1.64%).

However, the correlation between the percentage of instructions that are trivial computations and the speedup resulting from exploiting these trivial computations is not universally true. *epic-Uncompress*, at 13.97%, has the fourth highest speedup across all of these benchmarks, although it has the third lowest percentage of instructions that are trivial computations. There are at least two reasons as to why this correlation does not hold for all benchmarks. First of all, eliminating a trivial computation affects the performance more than simplifying a trivial computation since eliminating a trivial computation reduces that instruction's latency to zero cycles while simplifying it reduces it to two cycles (1 cycle to issue the instruction and another to execute it). Also, although simplifying a trivial computation could dramatically reduce its execution latency, the simplified instruction still uses a functional unit, albeit a different one, which increases the amount of functional unit contention. Finally, by definition, only trivial computations that can be eliminated benefit from early non-speculative instruction execution.

Second of all, although a benchmark may have a relatively large percentage of trivial computations, if very few of those trivial computations on are the program's critical path, then simplifying or eliminating most of the benchmark's trivial computations will not significantly improve the processor's performance. (Determining which computations are critical and which are not is virtually impossible since it requires storing all dynamic instructions in memory.)

6.2.2. Enhanced Processor Configuration

One potential criticism of simplifying and eliminating trivial computations is that the trivial computation hardware merely functions as an additional functional unit or two. In other words, the processor's performance could be similarly improved by adding a couple of functional units. To determine the validity of this criticism, Figures 6.2.2.1 and 6.2.2.2 show the speedup due to simplifying and eliminating trivial computations for a base processor without any functional unit constraints (there are four functional units of each type, which matches the maximum issue width).

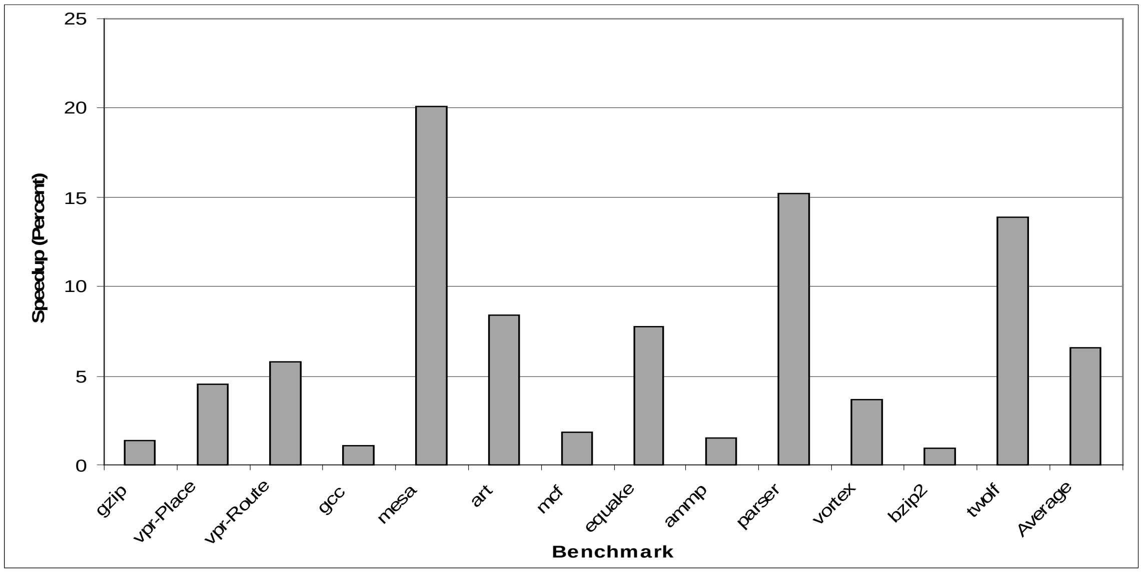

Figure 6.2.2.1: Speedup Due to the Simplification and Elimination of Trivial Computations for Selected SPEC 2000 Benchmarks, Enhanced Processor Configuration

For the SPEC benchmarks, Figure 6.2.2.1 shows that the speedups range from 0.95% (*bzip2*) to 20.09% (*mesa*), with an average of 6.60%. For the MediaBench benchmarks, Figure 6.2.2.2 shows that the speedups range from 1.86% (*adpcm-Decode*) to 10.04% (*mpeg2-Encode*), with an average of 2.92%. These results show that even for a processor without any functional unit constraints, exploiting trivial computations can either moderately (SPEC) or slightly (MediaBench) decrease the execution time. Therefore, the criticism that the trivial computation hardware simply functions as a *de facto* functional unit is generally incorrect.

Although the average speedup for the SPEC benchmarks is fairly impressive, two benchmarks, *mesa* and *vortex*, have significantly lower speedups when the base processor uses the enhanced processor configuration. The speedup for *mesa* decreases from 27.36% when using the realistic processor configuration to 20.09% when using the enhanced processor configuration while the speedup for *vortex* decreases from 9.34% to 3.66%. The speedup of *epic-Uncompress* also exhibits a similar change (13.97% to 4.05%) when using those two configurations. The reason that the speedups decrease is because the enhanced processor configuration relieves much of the functional unit contention, which previously limited the processor's performance and allowed the trivial computation hardware to have more effect.

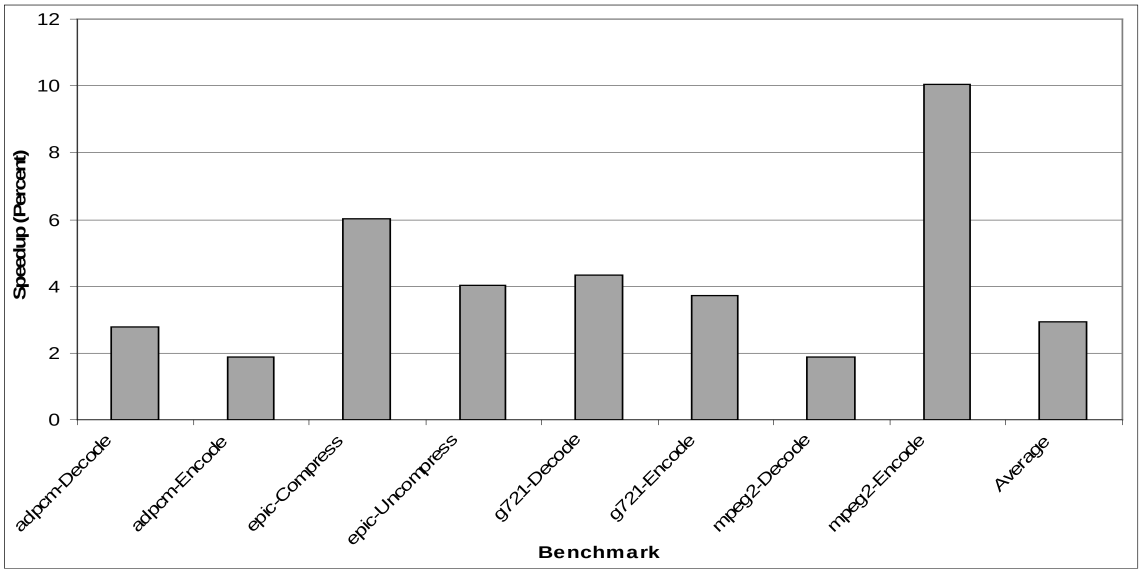

Figure 6.2.2.2: Speedup Due to the Simplification and Elimination of Trivial Computations for Selected MediaBench Benchmarks, Enhanced Processor Configuration

Therefore, for these three benchmarks, since the speedups sharply decreases when additional functional units are added to the base processor, the criticism of exploiting trivial computation hardware is somewhat accurate. However, simplifying and eliminating trivial computations still decreases the execution time of these benchmarks (and the other benchmarks) by reducing the execution latency of some instructions and by early non-speculative instruction execution.

The speedups for three other benchmarks, *twolf*, *epic-Compress*, and *mpeg2-Encode*, increased from 13.62% to 13.92%; 2.97% to 6.02%; and 5.24% to 10.04%; respectively, when using the enhanced processor configuration. Since the base execution time when using the enhanced configuration is lower than the base execution time when using the realistic configuration, the performance impact of early non-speculative instruction execution and reducing the instruction's execution latency have a larger effect. In other words, the decrease in the number of cycles due to dynamically simplifying and eliminating trivial computations accounts for a larger percentage of the total execution time, which then yields a larger speedup.

6.2.3. Summary

The results in this section show that the Simplification and Elimination of Trivial Computations can significantly improve the processor's performance. For a realistic processor configuration, adding hardware to exploit trivial computations yields an average speedup of 8.86% for the SPEC benchmarks and 4.00% for the MediaBench benchmarks. Even for a processor without any functional unit constraints, this enhancement still yields average speedups of 6.60% and 2.92% for the SPEC and MediaBench benchmarks, respectively. This last result illustrates the efficacy that reducing an instruction's execution latency and non-speculatively executing instructions early can have on the processor's performance.

6.3. The Results of Applying a Statistically Rigorous Simulation Methodology

As described in Chapter 4, a computer architect can use the results of a Plackett and Burman design in three ways to improve simulation methodology. First, when choosing processor parameter values, the architect can use the Plackett and Burman design to determine which parameters have the most effect on the performance. Second, when choosing the set of benchmarks, architect can use the Plackett and Burman design to determine the similarity between benchmarks. Finally, the architect can use the Plackett and Burman design as an analysis tool to finely examine the effect that an enhancement has on the processor.

Using the Plackett and Burman design, Section 6.3.1 determines the most significant SimpleScalar processor parameters while Section 6.3.2 chooses a set of statistically different benchmarks. Finally, Section 6.3.3 analyzes the effect that Instruction Precomputation and Simplifying and Eliminating Trivial Computations have on the processor.

6.3.1. Analysis of Processor Parameters for Parameter Value Selection

The third step of the simulation process is parameter value selection. The key to minimizing the amount of error is to understand which parameters have the most effect on the processor's performance. Table 6.3.1.1 presents the parameters in descending order of significance.

Table 6.3.1.1 shows the results of a Plackett and Burman design with foldover (X=44) for a superscalar processor with the parameter values shown in Tables 5.3.1, 5.3.2., and 5.3.3. After simulating all 88 (2*X) configurations, the Plackett and Burman design results were calculated by first assigning a rank to each parameter based on its significance (1 = most important). Then the ranks of each parameter were averaged across all benchmarks and the resulting averages sorted in ascending order; the rightmost column shows the average sum-of-ranks. Averaging the ranks across benchmarks reveals the most significant parameters across all of the benchmarks. Consequently, the parameters with the lowest averages represent the parameters that have the most effect across all benchmarks.

This table shows several key results. First, only the first ten parameters are significant across all benchmarks. This conclusion is drawn by examining the large difference between the average sum-of-ranks of the 10th parameter, LSQ size, which has an average sum-of-ranks of 12.6, and the average sum-of-ranks of the 11th parameter, Speculative

Branch Update, which has an average sum-of-ranks of 18.2. Furthermore, while the ranks of the top ten parameters for each benchmark are completely different, two parameters, ROB Entries and L2 Cache Latency are significant across all of the benchmarks since those two parameters invariably have one of the lowest ranks for every benchmark. Stating it differently, this means that the ROB and the L2 Cache Latency are the two biggest bottlenecks in the processor across all of the benchmarks tested in this dissertation. Therefore, of all the user-configurable simulator parameters, the architect needs to be especially careful when choosing parameter values for the number of reorder buffer entries and the L2 Cache Latency.

Table 6.3.1.1: Plackett and Burman Design Results for All Processor Parameters; Ranked by Significance and Sorted by the Average Sum-of-Ranks

Parameter	*gzip*	*vpr-Place*	*vpr-Route*	*gcc*	*mesa*	*art*	*mcf*	*equake*	*ammp*	*parser*	*vortex*	*bzip2*	*twolf*	Ave
ROB Entries	1	4	1	4	3	2	2	3	6	1	4	1	4	**2.8**
L2 Cache Latency	4	2	4	2	2	4	4	2	13	3	2	8	2	**4.0**
Branch Predictor	2	5	3	5	5	27	11	6	4	4	16	7	5	**7.7**
Int ALUs	3	7	5	8	4	29	8	9	19	6	9	2	9	**9.1**
L1 D-Cache Latency	7	6	7	7	12	8	14	5	40	7	5	6	6	**10.0**
L1 I-Cache Size	6	1	12	1	1	12	37	1	36	8	1	16	1	**10.2**
L2 Cache Size	9	35	2	6	21	1	1	7	2	2	6	3	43	**10.6**
L1 I-Cache Block Size	16	3	20	3	16	10	32	4	10	11	3	22	3	**11.8**
Memory Latency First	36	25	6	9	23	3	3	8	1	5	8	5	28	**12.3**
LSQ Entries	12	14	9	10	13	39	10	10	17	9	7	4	10	**12.6**
Speculative Branch Update	8	17	23	28	7	16	39	12	8	20	22	20	17	**18.2**
D-TLB Size	20	28	11	23	29	13	12	11	25	14	25	11	24	**18.9**
L1 D-Cache Size	18	8	10	12	39	18	9	36	32	21	12	31	7	**19.5**
L1 I-Cache Associativity	5	40	15	29	8	34	23	28	16	17	15	9	21	**20.0**
FP Multiply Latency	31	12	22	11	19	24	15	23	24	29	14	23	19	**20.5**
Memory Bandwidth	37	36	13	14	43	6	6	29	3	12	19	12	38	**20.6**
Int ALU Latencies	15	15	18	13	41	22	33	14	30	16	41	10	16	**21.8**
BTB Entries	10	24	19	20	9	42	31	20	22	19	20	17	34	**22.1**
L1 D-Cache Block Size	17	29	34	22	15	9	24	19	28	13	32	28	26	**22.8**
Int Divide Latency	29	10	26	16	24	32	41	32	20	10	10	43	8	**23.2**
Int Mult/Div	14	20	29	31	10	23	27	24	33	36	18	26	15	**23.5**
L2 Cache Associativity	23	19	14	19	32	28	5	39	37	18	42	21	12	**23.8**
I-TLB Latency	33	18	24	18	37	30	30	16	21	32	11	29	18	**24.4**
Instruction Fetch Queue Entries	43	13	27	30	26	20	18	37	9	25	23	34	14	**24.5**
Branch Misprediction Penalty	11	23	42	21	6	43	20	34	11	22	39	37	23	**25.5**
FP ALUs	34	11	31	15	34	17	40	22	26	37	13	42	13	**25.8**
FP Divide Latency	22	9	35	17	30	21	38	15	43	38	17	39	11	**25.8**
I-TLB Page Size	42	39	8	37	36	40	7	17	12	26	28	14	39	**26.5**
L1 D-Cache Associativity	13	38	17	34	18	41	34	33	14	15	35	15	42	**26.8**
I-TLB Associativity	24	27	37	25	17	31	42	13	29	30	21	33	22	**27.0**
L2 Cache Block Size	25	43	16	38	31	7	35	27	7	35	38	13	40	**27.3**
BTB Associativity	21	21	36	32	11	33	17	31	34	43	27	35	25	**28.2**
D-TLB Associativity	40	32	25	26	22	35	26	26	18	33	26	30	35	**28.8**
FP ALU Latencies	32	16	38	41	38	11	22	30	23	27	30	40	29	**29.0**
Memory Ports	39	31	41	24	27	15	16	41	5	42	29	41	27	**29.1**
I-TLB Size	35	34	28	35	20	37	19	18	31	34	34	27	31	**29.5**
Dummy Factor #2	27	42	21	39	35	14	13	35	41	28	43	18	30	**29.7**
FP Mult/Div	41	22	43	40	40	19	28	38	27	31	31	19	20	**30.7**
Int Multiply Latency	30	41	39	36	14	26	29	21	15	41	37	32	41	**30.9**
FP Square Root Latency	38	30	40	33	33	5	25	42	42	24	24	38	37	**31.6**
L1 I-Cache Latency	26	26	32	42	28	38	21	40	38	40	36	25	33	**32.7**
Return Address Stack Entries	28	33	33	27	42	25	36	25	39	39	33	36	32	**32.9**
Dummy Factor #1	19	37	30	43	25	36	43	43	35	23	40	24	36	**33.4**

Second, the effect that each benchmark has on the processor can be clearly seen. The "effect" that a benchmark has on the processor can be defined as the performance bottlenecks that are present in the processor when running that program. For example, for a compute intensive benchmark, the number of functional units will probably be a performance bottleneck for that processor. On the other hand, for a memory intensive benchmark, the sizes of the L1 D-Cache and the L2 Cache may be the performance bottlenecks.

In this case, for *mesa*, since the ranks for the L1 I-Cache size, associativity, and block size are lower than or similar to the ranks for the L1 D-Cache size, associativity, and block size, respectively, the performance of the instruction cache is more of a limiting factor than the performance of the data cache. The miss rates for the L1 I-Cache and the L1 D-Cache validate this result. When using a 32-byte cache block, the miss rate of the L1 I-Cache is similar to or higher than the miss rate of the L1 D-Cache. Therefore, it is not surprising to see that the L1 I-Cache parameters are generally more significant.

Third, several parameters have surprisingly low ranks in some benchmarks. For example, the FP square root latency in *art* has a rank of five. Since *art* does not have a significant number of FP square root instructions, its rank does not appear to be consistent with its intuitive significance. However, what the rank does not show is that the magnitude of the effect for this parameter is much smaller than magnitudes of the effects for the four most significant parameters. In other words, while ranking the parameters for each benchmark provides a basis for comparison across benchmarks, it cannot be used as the sole arbiter in concluding the significance of a parameter's impact since the rank does not represent the magnitude of the effect.

Finally, Table 6.3.1.1 shows that the L1 D-Cache parameters (size, associativity, block size, and latency) are not as significant as one would expect. The lowest ranks for the L1 D-Cache size, associativity, block size, and latency are 7 (*twolf*), 13 (*gzip*), 9 (*art*), and 5 (*vortex*), respectively. Given the amount of effort that the computer architecture community has put into improving memory performance, one would expect that the L1

D-Cache parameters would have much lower ranks. Therefore, the key question is: What are not the L1 D-Cache parameters more significant?

One potential reason is that the specific input set that is used for these benchmarks does not adequately stress the memory hierarchy. When using the large input set, the L1 D-Cache miss rates for those benchmarks are much lower than when using the reference input set [Yi02-2] since the reference input set usually has a much larger memory footprint. Therefore, since the high and low values for the L1 D-Cache parameters were based on the values present in commercial processors and were not downsized to account for the smaller memory footprint, the cache miss rates are subsequently lower. One consequence of the lower-than-expected cache miss rates is that the L1 D-Cache parameters have less impact on the performance (i.e. higher rank).

To minimize the effect of using an input set that produces a smaller-than-expected memory footprint, one solution is to use smaller values for the L1 D-Cache and L2 Cache size and associativity. Therefore, to determine how much of an effect that the lower cache miss rates have on ranking of the cache parameters, the low value of the L1 D-Cache size was reduced from 4 KB to 1 KB while the low value of the L2 Cache size was reduced from 256 KB to 64 KB. The associativities of both caches were not reduced since they were at the absolute minimum value (1-way).

Table 6.3.1.2 shows parameters in descending order of significance when using the reduced cache size configurations.

Table 6.3.1.2 show that reducing the sizes of the L1 D-Cache and the L2 Cache to account for using a reduced input set significantly changes the effect that the memory parameters have on the performance. The sums-of-ranks for the L1 D-Cache size, L1 D-Cache associativity, the L2 Cache size, and the L2 Cache associativity decreases by 71, 86, 86, and 137, respectively. Accordingly, the importance of these four parameters also increases. The L1 D-Cache increases from 13th most important parameter to the 12th, the L1 D-cache associativity changes from 29th to 17th, the L2 Cache size increases from 7th to 2nd, and the L2 Cache associativity changes from 22nd to 8th. In other words, these four parameters have a much larger effect on the performance when the cache sizes are downsized to compensate for the smaller memory footprint.

Table 6.3.1.2: Plackett and Burman Design Results for All Processor Parameters; Ranked by Significance and Sorted by the Average Sum-of-Ranks; Reduced Cache Sizes

Parameter	*gzip*	*vpr-Place*	*vpr-Route*	*gcc*	*mesa*	*art*	*mcf*	*equake*	*ammp*	*parser*	*vortex*	*bzip2*	*twolf*	Ave
ROB Entries	2	4	2	7	3	3	3	3	6	3	7	1	5	**3.8**
L2 Cache Size	1	8	1	2	19	1	1	5	2	1	3	2	6	**4.0**
L2 Cache Latency	6	2	5	3	2	16	5	2	12	4	2	7	2	**5.2**
Memory Latency First	4	9	3	4	21	2	2	6	1	2	4	4	7	**5.3**
Int ALUs	5	10	6	9	4	33	6	8	19	8	11	3	10	**10.2**
L1 I-Cache Size	7	1	15	1	1	27	17	1	36	17	1	25	1	**11.5**
L1 D-Cache Latency	11	6	10	8	10	17	30	7	42	7	9	5	8	**13.1**
L2 Cache Associativity	10	20	9	6	26	4	9	10	38	5	8	13	14	**13.2**
L1 I-Cache Block Size	19	3	41	5	15	8	26	4	10	12	5	22	4	**13.4**
Branch Predictor	9	7	8	11	5	34	25	9	4	6	35	12	9	**13.4**
Memory Bandwidth	13	30	13	10	38	7	4	12	3	9	13	10	17	**13.8**
L1 D-Cache Size	8	5	4	14	41	6	20	19	26	16	6	14	3	**14.0**
LSQ Entries	20	13	12	13	14	23	35	15	17	27	10	6	11	**16.6**
Instruction Fetch Queue Entries	17	18	18	22	24	9	11	27	9	14	40	18	24	**19.3**
FP Multiply Latency	26	12	21	12	20	28	33	11	23	26	14	24	13	**20.2**
L1 D-Cache Associativity	12	37	19	24	16	12	24	14	14	10	31	11	39	**20.2**
L1 I-Cache Associativity	3	33	14	17	9	43	19	35	16	24	12	9	32	**20.5**
L2 Cache Block Size	14	38	7	31	27	36	7	23	7	18	29	8	33	**21.4**
L1 D-Cache Block Size	16	42	34	20	17	10	29	13	29	11	42	27	34	**24.9**
BTB Entries	18	22	31	27	8	31	39	16	22	21	36	16	40	**25.2**
I-TLB Page Size	33	31	11	42	34	18	15	28	13	41	21	19	21	**25.2**
Dummy Factor #2	23	36	16	23	32	26	22	26	43	20	25	17	23	**25.5**
Int Mult/Div	25	16	27	39	11	15	40	34	33	29	18	41	15	**26.4**
Dummy Factor #1	22	43	29	26	29	13	21	18	35	13	38	21	35	**26.4**
FP ALUs	32	11	23	33	33	11	36	43	25	32	16	39	12	**26.6**
D-TLB Size	40	41	36	36	28	14	10	17	28	22	26	15	36	**26.8**
D-TLB Associativity	28	17	25	25	23	32	42	33	18	38	19	31	20	**27.0**
FP ALU Latencies	21	24	32	15	39	21	23	41	24	15	30	29	41	**27.3**
FP Square Root Latency	31	26	43	43	36	5	12	29	41	19	17	26	31	**27.6**
Int Divide Latency	34	14	42	29	25	29	41	25	20	23	20	42	16	**27.7**
I-TLB Size	30	25	17	16	22	37	27	40	32	43	15	38	19	**27.8**
FP Mult/Div	27	35	33	18	37	22	28	31	27	25	22	20	37	**27.8**
Branch Misprediction Penalty	41	40	24	30	6	24	8	39	11	36	27	35	42	**27.9**
Speculative Branch Update	15	27	40	40	7	41	38	22	8	37	32	30	26	**27.9**
Memory Ports	24	39	20	37	30	20	14	38	5	30	39	28	43	**28.2**
Int Multiply Latency	39	32	28	21	13	40	31	42	15	42	24	37	18	**29.4**
Return Address Stack Entries	43	28	37	19	43	38	16	24	39	31	23	33	22	**30.5**
FP Divide Latency	37	15	22	32	35	35	32	20	40	35	34	36	28	**30.8**
L1 I-Cache Latency	38	19	30	38	31	19	43	32	37	28	28	32	27	**30.9**
I-TLB Associativity	29	34	35	34	18	30	34	21	31	33	41	34	29	**31.0**
BTB Associativity	42	29	26	35	12	25	18	37	34	34	37	43	38	**31.5**
Int ALU Latencies	36	21	38	28	42	42	13	36	30	39	43	23	25	**32.0**
I-TLB Latency	35	23	39	41	40	39	37	30	21	40	33	40	30	**34.5**

Another reason that the L1 D-Cache parameters are not more significant is that the memory hierarchy of *sim-outorder* tends to overestimate the memory performance since it does not model memory contention. In addition, *sim-outorder* has a shorter-than-normal pipeline, does not partition the execution core, does not replay traps, and has fewer pipeline flushes. The net effect of these factors is that the average IPC "error" of SimpleScalar for eight selected SPEC 2000 benchmarks is 36.7% [Desikan01]. Given this rather large margin of error, the unrealistic memory behavior, and the smaller-than-expected memory footprint, it is not too surprising to see that the ranks of the L1 D-Cache parameters are not as low as expected.

The results in this section illustrate how a Plackett and Burman design can be used to identify the key processor parameters, which is very useful when the computer architect is trying to select processor parameters values. In this example, there are 10 key processor parameters (out of 41), with the number of ROB entries and the L2 Cache Latency being the two most important.

6.3.2. Analysis of Benchmarks for Benchmark Selection

The fourth step of the simulation process is benchmark selection. As described in Chapter 4, a potential pitfall in choosing benchmarks is that the architect may inadvertently choose a set of the benchmarks that are not representative of the target applications. One way of avoiding this problem is to understand the effect that each benchmark has on the processor in greater detail and then to select benchmarks that are dissimilar. Two benchmarks are defined to be similar if they have a similar effect on the processor.

Starting with the results of the Plackett and Burman design, the first step in determining whether two benchmarks have similar effects on the processor is to calculate the Euclidean distance between all possible pair-wise combinations of benchmarks. Since the Plackett and Burman design results for each benchmark is simply a vector of ranks, where each value in the vector corresponds to the rank for that parameter, the formula for computing the Euclidean distance is simply:

$$\text{Distance} = [(x_1-y_1)^2 + (x_2-y_2)^2 + \ldots + (x_{n-1}-y_{n-1})^2 + (x_n-y_n)^2]^{½}$$

In this formula, n is the number of parameters while $X = [x_1, x_2, \ldots, x_{n-1}, x_n]$ and $Y = [y_1, y_2, \ldots, y_{n-1}, y_n]$ are the vector of ranks that represent benchmarks X and Y, respectively.

For example, the Euclidean distance between *gzip* and *vpr-Place,* using the ranks from Table 6.3.1.1, is as follows:

$$\text{Distance} = [(1-4)^2 + (4-2)^2 + \ldots + (28-33)^2 + (19-37)^2]^{½} = [8058]^{½} = 89.8$$

In the second step, the benchmarks were clustered together based on their Euclidean distances. And in the third step, the final clustering tree is plotted. Figure 6.3.2.1 shows the output of the cluster analysis for the benchmarks and input sets given in Table 6.4.3.

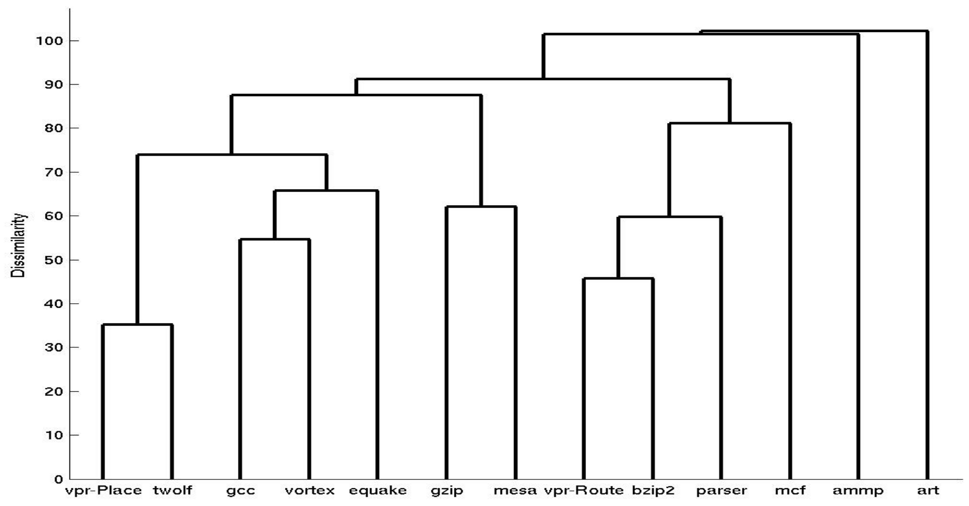

Figure 6.3.2.1: Cluster Analysis Results (i.e. Dendrogram) for the Large MinneSPEC Input Set

In Figure 6.3.2.1, the benchmarks are arranged along the x-axis while the y-axis represents the level of dissimilarity between any two benchmarks (or group of benchmarks). Whenever two benchmarks are connected by a horizontal line, that means at that level of dissimilarity and higher, those two benchmarks are considered to be similar. The level of dissimilarity is simply the Euclidean distance. For example, since *vpr-Route* and *twolf* are connected together at a dissimilarity of 35.19, for dissimilarities (or Euclidean distances) less than 35.19, those two benchmarks are categorized into separate groups. However, when the level of dissimilarity exceeds 35.19, they are categorized into the same group. All benchmarks in the same group are considered to be similar.

The first step in selecting a final group of benchmarks to simulate from the dendrogram is to draw a horizontal line at a dissimilarity of 0. Then the horizontal line should be moved up until it the number of vertical lines that it intersects matches the maximum number of benchmarks that can be simulated. The number of intersecting vertical lines represents the number of groups that the benchmarks have been classified into. At that level of dissimilarity, all benchmarks within the same group are considered to be similar while any benchmark in another group is considered to be dissimilar. The final step in the benchmark selection process is to select one benchmark from each group to form the final set of benchmarks.

For example, assume that the architect can simulate a maximum of eight benchmarks. Therefore, the 13 benchmarks need to be categorized into eight different groups. From a dissimilarity of 0 to dissimilarity of 35.18, the 13 benchmarks are in 13 different groups. From 35.19 to 45.73, the 13 benchmarks are in 12 different groups because *vpr-Place* and *twolf* are categorized into the same group. From 45.74 to 54.58, the 13 benchmarks are in 11 different groups after *vpr-Route* and *bzip2* are categorized together. This process continues until the 13 benchmarks are categorized into desired number of different groups. Table 6.3.2.1 shows the final categorization with eight groups.

Table 6.3.2.1: Example of Benchmark Selection, Choosing Eight Benchmarks from Thirteen

Group	Benchmarks	Final Set
I	*gzip, mesa*	*gzip*
II	*vpr-Place, twolf*	*vpr-Place*
III	*vpr-Route, parser, bzip2*	*vpr-Route*
IV	*gcc, vortex*	*gcc*
V	*art*	*art*
VI	*mcf*	*mcf*
VII	*equake*	*equake*
VIII	*ammp*	*ammp*

The middle column of Table 6.3.2.1 shows the benchmarks in each group while the rightmost column shows the benchmark that were selected from each group to form the final set.

The final set of benchmarks consists of five integer benchmarks (*gzip*, *vpr-Place*, *vpr-Route*, *gcc*, and *mcf*) and three floating-point benchmarks (*art*, *equake*, and *ammp*). In addition, two of the benchmarks (*art* and *mcf*) have very high cache miss rates (over 20% for a 32 KB, 2-way associative cache) while the other six have comparatively low miss rates (less than 5% for a 32 KB, 2-way cache). Therefore, the final set of benchmarks consists of benchmarks that would come from different groups when categorizing benchmarks using existing methods (integer versus floating-point, etc.).

Finally, it is important to note that it may be useful to consider other factors when selecting a benchmark from each group. In this example, one reason to choose *gzip* instead of *mesa* from Group I, is because *gzip* has a much lower instruction count although those two benchmarks are statistically similar. Similarly, one reason to choose *vpr-Route* over *parser* and *bzip2* from Group III is to match the choice of *vpr-Place* from Group II.

The results in this section illustrate how a Plackett and Burman design can be used to help the computer architect select a set of statistically different benchmarks. In particular, these results illustrated how eight benchmarks could be selected from the candidate list of 13 SPEC 2000 benchmarks by first classifying them into eight groups and then selecting one benchmark from each group.

6.3.3. Analysis of the Effect of Processor Enhancements

The sixth and final step of the simulation process is analyzing the effect of an enhancement. To illustrate how a Plackett and Burman design can be used in this way, this technique was used to analyze the effects that Instruction Precomputation and

Simplifying and Eliminating Trivial Computations had on the processor. Table 6.3.3.1 presents the Plackett and Burman design results for Instruction Precomputation while Table 6.3.3.2 does the same for Simplifying and Eliminating Trivial Computations.

Table 6.3.3.1 shows the results for Instruction Precomputation for Profile B, Run A using a 128-entry PT. Table 6.3.3.1 represents the "after" case while Table 6.3.1.1 represents the "before" case, that is, the unenhanced processor.

Table 6.3.3.1: Plackett and Burman Design Results for All Processor Parameters When Using Instruction Precomputation; Ranked by Significance and Sorted by the Average Sum-of-Ranks

Parameter	*gzip*	*vpr-Place*	*vpr-Route*	*gcc*	*mesa*	*art*	*mcf*	*equake*	*ammp*	*parser*	*vortex*	*bzip2*	*twolf*	Ave
ROB Entries	1	4	1	4	3	2	2	3	6	1	4	1	4	**2.8**
L2 Cache Latency	4	2	4	2	2	4	4	2	13	3	2	8	2	**4.0**
Branch Predictor	2	5	3	5	5	28	11	8	4	4	16	7	5	**7.9**
L1 D-Cache Latency	7	6	5	7	11	8	14	5	40	7	5	4	6	**9.6**
L1 I-Cache Size	5	1	12	1	1	12	38	1	36	8	1	15	1	**10.2**
Int ALUs	6	8	8	9	8	29	9	13	20	6	9	3	9	**10.5**
L2 Cache Size	9	35	2	6	22	1	1	6	2	2	6	2	43	**10.5**
L1 I-Cache Block Size	15	3	20	3	14	10	32	4	10	11	3	20	3	**11.4**
Memory Latency First	35	25	6	8	18	3	3	7	1	5	7	6	27	**11.6**
LSQ Entries	13	14	9	10	15	40	10	9	17	9	8	5	10	**13.0**
D-TLB Size	21	28	11	24	25	13	12	10	25	14	25	10	24	**18.6**
Speculative Branch Update	8	20	25	29	7	16	39	11	8	20	21	22	19	**18.8**
L1 I-Cache Associativity	3	41	15	28	6	34	23	28	16	17	11	9	21	**19.4**
L1 D-Cache Size	18	7	10	12	42	19	8	35	32	21	13	32	7	**19.7**
FP Multiply Latency	31	12	22	11	19	24	15	22	24	28	14	24	18	**20.3**
Memory Bandwidth	33	36	13	14	43	6	6	31	3	12	20	11	38	**20.5**
BTB Entries	10	23	19	20	9	41	31	20	22	19	19	16	34	**21.8**
Int ALU Latencies	16	15	18	13	40	22	33	14	31	16	41	12	16	**22.1**
L1 D-Cache Block Size	17	30	34	22	16	9	24	19	26	13	33	25	26	**22.6**
Int Divide Latency	30	10	26	17	24	33	40	33	19	10	10	41	8	**23.2**
L2 Cache Associativity	23	19	14	19	33	27	5	39	37	18	42	21	12	**23.8**
Int Mult/Div	14	21	30	31	12	23	27	23	33	37	18	27	15	**23.9**
I-TLB Latency	32	17	24	18	34	30	30	16	21	33	12	29	17	**24.1**
Instruction Fetch Queue Entries	43	13	27	30	23	20	19	37	9	25	23	34	14	**24.4**
Branch Misprediction Penalty	11	24	41	21	4	43	20	32	11	22	39	35	23	**25.1**
FP Divide Latency	20	9	36	16	28	21	37	15	43	38	17	38	11	**25.3**
FP ALUs	34	11	31	15	38	17	41	24	27	36	15	43	13	**26.5**
I-TLB Page Size	42	38	7	38	39	39	7	17	12	26	28	14	39	**26.6**
L1 D-Cache Associativity	12	39	17	35	17	42	34	34	14	15	36	17	42	**27.2**
L2 Cache Block Size	25	43	16	37	31	7	35	27	7	35	38	13	40	**27.2**
I-TLB Associativity	26	27	38	25	20	31	42	12	29	30	22	33	22	**27.5**
BTB Associativity	22	18	35	32	10	32	17	30	34	43	27	36	25	**27.8**
D-TLB Associativity	40	32	23	26	27	35	25	26	18	32	26	28	35	**28.7**
Memory Ports	39	31	39	23	26	15	16	40	5	42	30	40	29	**28.8**
FP ALU Latencies	37	16	37	41	37	11	21	29	23	27	29	42	28	**29.1**
I-TLB Size	36	34	28	34	21	37	18	18	30	34	34	30	32	**29.7**
Dummy Factor #2	28	42	21	39	32	14	13	36	42	29	43	18	30	**29.8**
Int Multiply Latency	29	40	42	36	13	26	29	21	15	41	35	31	41	**30.7**
FP Mult/Div	41	22	43	40	41	18	28	38	28	31	31	19	20	**30.8**
FP Square Root Latency	38	29	40	33	35	5	26	43	41	24	24	39	37	**31.8**
Return Address Stack Entries	27	33	33	27	36	25	36	25	39	40	32	37	31	**32.4**
L1 I-Cache Latency	24	26	32	42	29	38	22	41	38	39	37	26	33	**32.8**
Dummy Factor #1	19	37	29	43	30	36	43	42	35	23	40	23	36	**33.5**

Comparing these two tables yields two conclusions about the effect that Instruction Precomputation has on the processor. First of all, the same parameters that were significant for the base processor are also significant for the processor with Instruction

Precomputation. While Instruction Precomputation changes the relative ordering of the significant parameters, with respect to each other, it does not change which parameters have the greatest significance.

Second, of the significant parameters, the parameter that has the biggest change in its overall effect (defined as the biggest change in its average sum-of-ranks) is the number of integer ALUs. Instruction Precomputation changes its average sum-of-ranks from 9.1 in the base processor to 10.5. This result is intuitively reasonable since most of the instructions that Instruction Precomputation eliminates would have executed on the integer ALUs. Therefore, by using Instruction Precomputation, the impact of the number of integer ALUs on the processor's performance decreases in significance.

Although these results show that Instruction Precomputation improves the processor's performance by reducing functional unit contention, it also improves the processor's performance by decreasing the execution latency of redundant computations. However, since the base SimpleScalar processor has a short, fixed-length pipeline, this latter effect appears to be relatively unimportant.

Table 6.3.3.2 shows the results for Simplifying and Eliminating Trivial Computations when using the realistic base processor configuration. Tables 6.3.1.1 and 6.3.3.2 represent the before and after cases, respectively.

Simplifying and Eliminating Trivial Computations has a similar effect on all processor parameters. That is, the performance bottlenecks in the base processor do not get substantially better or worse when hardware to exploit trivial computations is added to the processor. There are two reasons to support this conclusion. First, the order of the ten most significant parameters is the same as the base processor. Since their sums-of-ranks are nearly identical, this enhancement has a very similar on the most important processor parameters.

Second, there is relatively little difference between the average sums-of-ranks for the other parameters. The maximum difference between the average sums-of-ranks for a parameter with and without adding the trivial computation exploitation hardware is 1.3. Although this difference rivals the change in the average sum-of-ranks for the Number of Integer ALUs when Instruction Precomputation is added to the base processor, this difference is less meaningful because it is a smaller percentage of the average sum-of-ranks for that parameter. In other words, since those parameters are very insignificant to being with, large changes in their average sums-of-ranks do not imply that the enhancement has a large effect on that parameter.

Table 6.3.3.2: Plackett and Burman Design Results for All Processor Parameters When Simplifying and Eliminating Trivial Computations; Ranked by Significance and Sorted by the Average Sum-of-Ranks

Parameter	*gzip*	*vpr-Place*	*vpr-Route*	*gcc*	*mesa*	*art*	*mcf*	*equake*	*ammp*	*parser*	*vortex*	*bzip2*	*twolf*	Ave
ROB Entries	1	4	1	4	3	2	2	3	6	1	4	1	4	**2.8**
L2 Cache Latency	4	2	4	2	2	4	4	2	13	3	2	8	2	**4.0**
Branch Predictor	2	5	3	5	5	27	11	8	4	4	16	7	5	**7.8**
Int ALUs	6	8	7	8	6	29	9	6	19	7	9	2	9	**9.6**
L1 D-Cache Latency	7	6	5	7	12	8	14	5	40	6	5	6	6	**9.8**
L1 I-Cache Size	5	1	12	1	1	11	38	1	36	8	1	16	1	**10.2**
L2 Cache Size	9	40	2	6	21	1	1	7	2	2	6	3	43	**11.0**
L1 I-Cache Block Size	16	3	20	3	15	10	32	4	10	11	3	21	3	**11.6**
Memory Latency First	36	28	6	9	20	3	3	9	1	5	7	5	27	**12.2**
LSQ Entries	12	16	9	10	14	43	10	10	17	10	8	4	10	**13.3**
Speculative Branch Update	8	18	26	29	7	16	39	13	8	20	21	20	18	**18.7**
L1 D-Cache Size	18	7	11	12	43	18	8	25	31	21	12	33	7	**18.9**
D-TLB Size	20	27	10	23	30	13	12	12	25	14	27	11	25	**19.2**
FP Multiply Latency	31	11	22	11	19	24	15	17	24	26	14	24	17	**19.6**
Memory Bandwidth	37	41	13	14	39	6	6	22	3	12	20	12	39	**20.3**
L1 I-Cache Associativity	3	38	16	28	8	34	23	43	16	16	13	9	21	**20.6**
BTB Entries	10	22	18	20	9	42	30	15	22	18	19	17	34	**21.2**
Int Divide Latency	29	10	24	17	25	33	40	21	20	9	10	43	8	**22.2**
L1 D-Cache Block Size	17	29	31	22	16	9	24	14	27	13	34	28	26	**22.3**
Int ALU Latencies	15	14	19	13	42	21	33	18	30	17	43	10	16	**22.4**
L2 Cache Associativity	23	15	14	19	33	28	5	38	37	19	39	22	13	**23.5**
Branch Misprediction Penalty	11	23	43	21	4	40	20	19	11	22	42	36	23	**24.2**
Int Mult/Div	14	24	30	31	11	23	27	34	33	37	17	26	14	**24.7**
FP Divide Latency	21	9	34	16	26	22	37	11	43	38	18	38	11	**24.9**
Instruction Fetch Queue Entries	43	13	27	30	27	20	18	40	9	25	23	35	15	**25.0**
I-TLB Latency	32	20	23	18	37	30	31	24	21	33	15	27	19	**25.4**
L1 D-Cache Associativity	13	39	15	34	18	41	34	23	14	15	37	15	41	**26.1**
FP ALUs	34	12	33	15	31	17	41	32	26	36	11	41	12	**26.2**
BTB Associativity	22	19	37	32	10	32	17	20	34	43	26	34	24	**26.9**
I-TLB Page Size	42	35	8	38	36	39	7	26	12	27	29	14	38	**27.0**
I-TLB Associativity	25	25	36	25	17	31	42	16	29	29	22	32	22	**27.0**
L2 Cache Block Size	26	42	17	37	32	7	35	41	7	35	36	13	40	**28.3**
Memory Ports	40	30	41	24	28	14	16	29	5	42	31	40	29	**28.4**
D-TLB Associativity	39	33	25	26	22	35	25	37	18	32	25	29	36	**29.4**
FP ALU Latencies	33	21	38	41	38	12	21	28	23	30	30	42	28	**29.6**
Dummy Factor #2	28	36	21	39	34	15	13	42	41	28	40	18	32	**29.8**
FP Mult/Div	41	17	42	40	40	19	28	39	28	31	32	19	20	**30.5**
I-TLB Size	35	31	28	35	23	36	19	27	32	34	33	30	33	**30.5**
FP Square Root Latency	38	32	39	33	35	5	26	33	42	24	24	39	37	**31.3**
Int Multiply Latency	30	43	40	36	13	26	29	31	15	41	38	31	42	**31.9**
Dummy Factor #1	19	37	29	43	24	37	43	30	35	23	41	23	35	**32.2**
L1 I-Cache Latency	24	26	35	42	29	38	22	36	38	39	35	25	31	**32.3**
Return Address Stack Entries	27	34	32	27	41	25	36	35	39	40	28	37	30	**33.2**

The results in this section illustrate how a Plackett and Burman design can be used to help the computer architect analyze the effect that a processor enhancement (hardware or software) has on the processor. The two examples that were given examined the effect that Instruction Precomputation and that Simplifying and Eliminating Trivial Computations had on the processor. In particular, the results show that Instruction Precomputation improves the processor's performance by decreasing the amount of functional unit contention while adding hardware to exploit trivial computations does not significantly create or relieve any performance bottlenecks.

6.3.4. Summary

The results in this section illustrate how a Plackett and Burman design can be used to improve simulation methodology. More specifically, a computer architect can use a Plackett and Burman design to identify the most significant processor parameters, which is important to know when choosing parameter values. Those results can also be used categorize benchmarks into groups, which is helpful when choosing benchmarks for simulations. Finally, the architect can use a Plackett and Burman design to determine the effect of a processor enhancement by comparing the sums-of-ranks for each processor parameter with and without the processor enhancement.

Chapter 7

Related Work

The scope of previous work that is related to this dissertation extends from work that has focused on value locality, value reuse and prediction, and improving processor performance by exploiting trivial computations to simulation methodology and the use of statistics in computer architecture research.

7.1. Value Locality

Value locality is the "likelihood of the recurrence of a previously seen value within a storage location" in a processor [Lipasti96-1]. In other words, value locality is the probability that an instruction produces the same output value.

Since their output value is constant, redundant computations exhibit value locality. As defined in Chapter 1, a redundant computation is a computation that the processor had performed earlier in the program. However, in this particular case, since their input values are constant (which along with the operation produces a constant output value), these instructions specifically exhibit input value locality. The difference between input and "normal" (output) value locality is that in the former, repetitive and constant input values are the reason for the repetitive output values while in the latter, the input values may differ while the output value is repetitive.

Section 7.1.1 summarizes the previous work that examined the amount of redundant computations that are present in SPEC benchmarks while Sections 7.2.1 and 7.2.2 describe solutions from the two major approaches of exploiting value locality: value reuse and value prediction.

7.1.1. Redundant Computations

Sodani and Sohi analyzed the amount of instruction repetition (amount of redundant computation at the local-level) in the integer benchmarks of the SPEC 95 benchmark suite [Sodani98]. Their results showed that 56.9% (*compress*) to 98.8% (*m88ksim*) of the dynamic instructions were repeated (i.e. had the same inputs – and, of course, produced the same result – as an earlier instance of the instruction). Therefore, in the case of *m88ksim*, almost all of the dynamic instructions were redundant with another dynamic instruction. This shows that in typical programs, such as those from the SPECint95 benchmark suite, a very large percentage of the computations are redundant.

Their results also showed that, of the static instructions that execute more than once, most of the repetition in the dynamic instructions is due to a small sub-set of the dynamic instructions. More specifically, with the exception of *m88ksim*, less than 20% of the static instructions that execute at least twice are responsible for over 90% of the dynamic instructions that are redundant. For *m88ksim*, those static instructions are responsible for over 50% of the instruction repetition.

However, it is important to reiterate that this paper only analyzed the amount of redundant computation present at the local-level.

Gonzalez *et al* [Gonzalez98] analyzed the amount of instruction repetition in the integer and floating-point benchmarks of the SPEC 95 benchmark suite. Like [Sodani98], their results were for instruction repetition at only the local-level. Their results showed that 53% (*applu*) to 99% (*hydro2d*) of the dynamic instructions were repeated. Furthermore, the geometric means of all the benchmarks, the integer benchmarks only, and the floating-point benchmarks only were 87%, 91%, and 83%, respectively. Consequently, there is not a significant difference in the amount of instruction repetition between the integer and floating-point benchmarks. Overall, their results confirmed the key conclusion from [Sodani98], that there is a significant amount of instruction repetition (local-level redundant computation) available in typical programs.

7.2. Value Reuse and Prediction

The two major techniques of exploiting value locality are value reuse (also called instruction reuse and instruction memorization [Citron98]) and value prediction.

7.2.1. Value Reuse

As explained in Section 1.4, value reuse exploits redundant computations by storing the opcode, input operand values, and output value for redundant computations into the value reuse table (VRT). When the current instruction's opcode and input operand values match an opcode and input operand value in the VRT, the processor can bypass the execution of that instruction and simply retrieve its output value from the VRT.

When exploiting local-level redundant computations, the PC is used as an index into the VRT. However, when exploiting global-level redundant computations, some combination of the opcode and input operand values are needed.

Sodani and Sohi [Sodani97] implemented a dynamic value reuse mechanism that exploited local-level only value reuse and tested it with selected SPEC 92 and 95 benchmarks. Since their value reuse mechanism exploited local-level redundant computations, they consequently indexed their value reuse table with each instruction's PC.

Their value reuse mechanism produced speedups of 0% to 17%, 2% to 26%, and 6% to 43% for a 32 entry, a 128 entry, and a 1024 entry, respectively, value reuse table. While the speedups in that paper are comparable to those for Instruction Precomputation, given in Section 6.1 of this dissertation, their approach needs additional hardware to dynamically update the VRT and does not exploit global-level redundant computations, as compared to Instruction Precomputation.

By contrast, Molina *et al* [Molina99] implemented a dynamic value reuse mechanism that exploited value reuse at the both the global and local-levels. To test the performance of their value reuse mechanism, they simulated selected integer and floating-point benchmarks from the SPEC 95 benchmark suite. Their approach is very area-intensive since it uses three separate value reuse tables to reuse global and local level computations and memory instructions. As can be expected, their speedups are somewhat correlated to the area used. For instance, their value reuse mechanism produced speedups of 3% to 25% with an average of 10% when using a 221 KB table. When the table area is reduced to a more realistic 36 KB, the speedups dropped to a range of 2% to 15% with an average of 7%. While their speedups are comparable to those presented in Chapter 6 of this dissertation, to achieve a similar speedup, their approach requires approximately ten times the area that Instruction Precomputation does (221 KB versus 26 KB).

Citron *et al* [Citron98, Citron00-1, Citron00-2] proposed using distributed value reuse tables that are accessed in parallel with the functional units. This approach, called memoziation, is best suited to bypass the execution of long latency instructions, e.g. integer divide; floating-point multiply, divide, and square root. Since their mechanism reduces the execution time of redundant computations to a single cycle, targeting only long latency instructions maximizes the performance gain due to this approach. As a result, although this mechanism produces speedups up to 20%, it is best suited for benchmarks with a significant percentage of high-latency instructions, such as the

MediaBench benchmark suite. There are two key differences between this approach and Instruction Precomputation. The first difference is that this approach uses distributed memorization tables, each of are which are located adjacent next to a functional unit, instead of a monolithic Precomputation Table. While smaller tables have a faster access time, all entries across all tables may not be used. Consequently, some tables may be full (and in the process of replacing entries) while other tables have empty entries. The other key difference is that the processor can access the memoization tables only in the execute stage while the VRT is usually accessed in the decode or issue stage. As a result, this approach cannot reuse the computations for single cycle latency instructions since there is no difference between the memoization table access latency and the execution latency for those instructions. The effect of this difference is that there are fewer instructions for this approach to reuse, as compared to value reuse, where all arithmetic, logical, and load instructions can be reused.

Huang and Lilja [Huang98, Huang99] introduced basic block reuse, which is value reuse at the basic block level. This approach uses the compiler to identify basic blocks where the inputs and outputs were are relatively constant and stable. Then, at run-time, the processor caches the inputs and outputs for those compiler-identified basic blocks after they finish executing. Subsequently, before the next execution of that basic block, the current inputs of that basic block are compared with all cached entries. If a match is found, then the register file and memory are updated with the correct results. Otherwise, the processor executes the basic block normally.

They showed that the average reused basic block ranged in size from 4.14 instructions (*wordcount*) to 5.95 instructions (*ijpeg*). This approach produced speedups of 1% to 14% with an average of 9%. The key difference between this approach and Instruction Precomputation is in the level of granularity. Basic block reuses multiple instructions at a time while Instruction Precomputation reuses one a time. However, the corresponding hardware cost is higher; each basic block reuse table entry requires at least 60 bytes of storage (as compared to 13 bytes for Instruction Precomputation).

In summary, the previous approaches produce comparable or lower speedups, as compared to Instruction Precomputation, while consuming either a little more area or, in some cases, an order of magnitude more area and while probably having a higher table access time. The other key difference is that the previous approaches allow for dynamic table replacement while Instruction Precomputation does not.

Azam *et al* [Azam97] proposed adding a dynamic reuse buffer and an extra pipeline stage (to access the reuse buffer) to decrease the processor's power consumption. Their results showed that an eight-entry reuse buffer decreased the power consumption by up to 20% while a 128-entry reuse buffer decreased the power consumption by up to 60%. While one of the goals of this solution is to decrease the power consumption while maintaining the performance (i.e. the same execution time) of the base processor, since performance numbers were not given in this paper, it is not clear if the performance goal was met.

Weinberg and Nagle [Weinberg98] proposed using value reuse to reduce the latency of pointer traversals by caching the elements of the pointer chain. This approach reduced the execution latency by up to 11.3%. However, this approach differs with Instruction Precomputation in three respects: 1) It only targets pointers, 2) It uses dynamic replacement, and 3) It consumes a very large amount of area (approximately 600 KB).

Finally, Gonzalez *et al* [Gonzalez98] measured the maximum performance potential of local-level value reuse, given an infinite VRT, a zero-cycle reuse latency (the number of cycles need to access the VRT and forward the output value to the redundant instruction), and a processor without any structural hazards. Their results showed that, for these benchmarks and for the given hardware assumptions, the average overall speedup is 18.7 (i.e. 1770%). The maximum possible speedup ranges from 1.5 (*applu*) to 2231 (*turb3d*).

7.2.2. Value Prediction

Value prediction is another microarchitectural technique that exploits value locality. The value predication hardware predicts the output values of future instances of each static instruction based on its past values. After the predicting the output value, the processor forwards that predicted value to any dependent instructions and then speculatively executes the dependent instructions based on the predicted value. To verify the prediction, the processor has to execute the predicted instruction normally. If the prediction is correct (i.e. the predicted value matches the actual value), then the processor resumes normal (non-speculative) execution and can commit the values of the speculatively executed dependent instructions. If the prediction is incorrect, then all of the dependent instructions need to be squashed and re-executed. This is the key difference between value reuse and value prediction; namely, that value prediction is speculative while value reuse is not.

7.2.2.1. Simple Value Predictors

Lipasti *et al* [Lipasti96-1, Lipasti96-2] introduced the concept of value locality and a means – last value prediction – to exploit it. Last value prediction stores the last output value of each static instruction into the value prediction table. Upon encountering the next instance of that static instruction, the processor uses the last output value as the predicted value. They showed that the average speedup for the last value prediction of load instructions is 6%, with a maximum of 17% [Lipasti96-1]. Meanwhile, the average speedup for arithmetic and load instructions is 7%, with a maximum of 54% [Lispasti96-2].

However, the accuracy of last value prediction is very poor when trying to predict the values of computations such as incrementing the loop induction variable. Therefore, to improve the prediction accuracy of last value prediction for these and similar computations, Gabby and Mendelson [Gabbay98] proposed another value predictor: the stride-value value predictor. By adding the difference of the last two output values (i.e. the stride) for that instruction to the last value for that instruction, this value predictor is able to accurately predict the output values for loop induction variables. Note that when the stride value equals to zero, the stride value predictor functions as a last value predictor.

For the integer benchmarks of the SPEC 95 benchmark suite, they showed that, for ALU instructions, the prediction accuracy of the last and stride value predictors were 52.8% and 61.1%, respectively. The improved prediction accuracy subsequently increases the amount of ILP significantly. For example, for *m88ksim*, stride value prediction increases the amount of ILP from 7 IPC to 34 IPC while last value prediction only increases the amount of ILP to 13 IPC. (Note: Speedup results were not given in this paper, only the increase in the amount of ILP.)

Although stride value prediction produces a higher prediction accuracy and a larger amount of ILP, as compared to last value prediction, the two predictors are fundamentally the same. Consequently, for more complex output value patterns such as 1, 4, 7, 9, 1, 4, 7, 9, … , 1, 4, 7, 9, etc., both value predictors have very poor performance. To address this shortcoming, Sazeides and Smith [Sazeides97] proposed the finite-context method (FCM) predictor. This two-level predictor stores the last n output values into the first level of the predictor while the hardware in the second level to chooses between those values. Consequently, this predictor is able to capture and accurately predict more complex, but regular patterns.

Their results showed that the prediction accuracy of the last value predictor is 23% to 61%, with an average of 40%, while the prediction accuracy of the stride value predictor is 38% to 80%, with an average of 56% for selected SPECint95 benchmarks. By comparison, the prediction accuracy of the FCM predictor is 56% to 90%, with an average of 78%. Since higher prediction accuracies translate into higher performance, combining two predictors together, such as the stride and FCM value predictors, should yield even higher prediction accuracy, and subsequently, performance. These hybrid value predictors are the subject of the next sub-section.

7.2.2.2. Complex (Hybrid) Value Predictors

For their hybrid value predictor, Rychlik *et al* [Rychlik98] combined an enhanced stride value predictor together with a FCM value predictor. They enhanced the base stride value predictor by adding second stride field. The first stride field stores the difference between the last two output values while the second stride field stores the last stable stride and is used in conjunction with the last output value to generate the predicted value. The first stride field only updates the stable stride field if the last two stride values are the same. For each instruction, both predictors make a prediction. The prediction from the predictor with the higher confidence counter value is chosen as the predicted value. In the case of a tie, the default choice is the FCM predictor.

For the SPEC 95 benchmarks, this hybrid value predictor achieved prediction accuracies of 74% to 83% and speedups of 9% to 23%, for a realistic machine. Although these prediction accuracies are not as high as those given in [Sazeides97], this hybrid value predictor reduces the total number of predictions by only allowing "useful" instructions into the value prediction table. A useful instruction is one which produces a value for a dependent instruction when both instructions are in the instruction window. Furthermore, these speedup results were enhanced by using a selective re-issue core, which only re-issues dependent instructions (instead of all instructions after the dependent one).

The hybrid value predictor proposed by Wang and Franklin [Wang97] is very similar to the predictor in [Rychlik98]. In this paper, the prediction from the FCM predictor is chosen if its confidence counter is higher than the prediction threshold. If not, then the prediction from the stride predictor is chosen, also if its confidence counter is higher than the prediction threshold. If not, then no prediction is made. The prediction accuracy and speedup results for this hybrid value predictor as similar to those shown by [Rychlik98].

7.3. Simplification and Elimination of Trivial Computations

The only previous work that focused directly on trivial computation is found in [Richardson92]. In this paper, Richardson restricted the definition of trivial computations to the following eight types: multiplications by 0, 1, and –1; certain divisions (X ÷ Y with X = {0, Y, -Y}), and square roots of 0 and 1. To exploit these trivial computations, Richardson proposed hardware that would eliminate the computations simply by setting the output value to the appropriate value (0, 1, -1, or –X). The latency of this bypass was assumed to be one cycle.

For the benchmarks from the SPECfp92 and Perfect Club benchmark suites, his results showed that 0% to 7.3% of the instructions were trivial in these benchmarks. His results showed that by eliminating these trivial computations, his proposed solution could improve the processor's performance by 2.1% for the SPEC benchmarks and 4.4% for the Perfect Club benchmarks.

The three key differences between this work and this dissertation are the types of benchmarks that were used, the scope of the definition of trivial computations, and how the trivial computations were exploited. The first difference is that Richardson restricted the definition of trivial computations to the above eight types while 26 types were defined in Chapter 3. The second difference is that Richardson studied only floating-point benchmarks (SPEC 92 and Perfect Club) while the results given in Chapter 6 are for a mix of integer, floating-point, and multimedia benchmarks from the SPEC 2000 and MediaBench benchmark suites. The third difference is that Richardson did not appear to use a simulator to determine the performance of his proposed solution. As a result, his speedup results do not account for pipeline effects. Instead of simplifying **and** eliminating the trivial computations, Richardson only eliminated them because his definition of trivial computations did not include the simplifiable ones. Furthermore, even for the trivial computations that could be directly eliminated, he did not take advantage of the early non-speculative scheduling.

Since Richardson did not increase the scope of trivial computations, the effects of the first and third differences result in a lower percentage of speedup. On the other hand, using only floating-point benchmarks and not including the effect of the processor's pipeline has the effect of somewhat over-inflating the speedup results. Despite this, for similar processor configurations, the average speedup of 2% that he reported was much lower than the 8.86% given in Chapter 6.

Finally, Richardson asserted that the lack of previous work on trivial computation was not due to its novelty, but due to a lack of knowledge as to how often trivial computations would occur.

Brooks and Martonosi [Brooks99] proposed two methods of reducing the operation’s bit width to improve the processor’s performance or decrease its power consumption. They noticed that for the SPEC and MediaBench benchmarks, more than 70% of all 64-bit arithmetic operations required 32-bits or less. To decrease the processor’s power consumption, 64-bit operations were converted into 32-bit operations. To improve the processor’s performance, the functional units were modified to allow two 32-bit operations to execute simultaneously.

Their results showed that by reducing the bit-width of the operation from 64 to 32 bits, the power consumption of the integer arithmetic units decreased by over 50%. On the other hand, executing two narrow-width operations on the same functional unit yields speedups of 4.3% to 6.2% for the SPEC 95 benchmarks and 8.0% to 10.4% for the MediaBench benchmarks.

7.4. Prior Work in Simulation Methodology

The related work in this section is divided into the following four categories: simulator validation, reducing the simulation time, benchmark and input set characterization, and processor parameter analysis.

7.4.1. Simulator Validation

The authors of several papers described their experiences when trying to validate the performance of a simulator against a reference machine or instruction set architecture (ISA). Black and Shen [Black98] iteratively improved the accuracy of their performance model by comparing the cycle count of their simulator, which targeted a specific architecture (in this case the Power PC 604), against the cycle count of the actual hardware. Their results show that modeling, specification, and abstraction errors were still present in their simulation model, even after a long period of debugging. In fact, some of these errors could be revealed only after comparing the performance model to the actual processor. As a result, their work showed the need for extensive, iterative validation before the results from a performance model can be trusted.

Desikan *et al* [Desikan01] measured the amount of error, as compared to the Alpha 21264 processor, that was present in an Alpha version of the SimpleScalar simulator. They defined the amount of error to be the difference in the simulated execution time and the execution time of the processor itself. Their results showed that the simulators that model a generic machine (such as SimpleScalar) generally report higher IPCs than simulators that are validated against a real machine. In other words, a simulator that does not target a specific architecture will generally report higher IPCs for the same benchmarks as compared to a validated simulator that targets a specific architecture. This result is not particularly surprising since it is likely that unvalidated, generic-architecture simulators will tend to underestimate the complexity of the implementing certain microarchitectural features that affect the clock period. On the other hand, unvalidated simulators that targeted a specific machine usually **underestimated** the performance.

Gibson *et al* [Gibson00] described the types of errors that were present in the FLASH simulator when compared to the custom-built FLASH multi-processor system. To determine which errors were present in the FLASH simulator, they compared the simulated execution time from the FLASH simulator against the actual execution time of the FLASH processor. In addition, they tested several different versions of their simulator to evaluate the accuracy versus simulation speed tradeoff of using a faster, but less complex simulator instead of a slower, but more complex simulator. Their results showed that most simulators can accurately predict the architectural trends if all of the important components have been accurately modeled. They also showed that a faster, less complex simulator that uses a scaling factor for the results often did a better job of predicting a processor's performance than a slower, more complex simulator. Finally, their results showed that the margin of error (the percentage difference in the execution time) of some simulators was more than 30%, which is higher than the speedups that are often reported for specific architectural enhancements.

Collectively, [Desikan01, Gibson00] show that the results from unvalidated simulators cannot be fully trusted and that any conclusions drawn from those results are suspect.

Glamm and Lilja [Glamm00] verified the functional correctness of a simulated ISA by simultaneously executing the instructions from a program on a simulator and on the targeted machine. Then, after each instruction, the simulated processor's state was compared to the real machine's processor state. Any difference between the states identified an error in the simulated ISA, which can then be fixed.

Cain *et al* [Cain02] measured the effect of the operating system and the effects of input and output (I/O) on simulator accuracy. To accomplish this task, they integrated the SimOS-PPC, an operating system that targets the PowerPC architecture, with SimMP, a multiprocessor simulator. Their results showed that the lack of an operating system could introduce errors as high as 100%. Furthermore, their results showed the potential for error due to I/O if the additional memory traffic is not properly taken into account. Overall, their results showed the need to integrate an operating system into the simulator for increased simulator accuracy and precision.

7.4.2. Reducing the Simulation Time

As described in Section 1.9, simulators are the most important tool in computer architecture research. The most accurate and detailed simulators are execution-driven, cycle-accurate simulators, such as SimpleScalar. While this type of simulator fully models all major processor components, they trade-off increased accuracy and detail for slower simulation speed. The slow simulation speed can be further exacerbated by the length of the benchmark and input set. For example, executing the *ammp* benchmark from the SPEC 2000 benchmark suite with the reference input set requires simulating approximately two **trillion** instructions. Therefore, the simulation time of this benchmark with the reference input set on a MIPS R14000 processor (which simulates this benchmark at 75,000 instructions per second) requires more than 308 *days*! As a result, since it is virtually impossible to explore even a small fraction of the design space with these long simulation times, the following papers have proposed different solutions to reduce the simulation time.

The most obvious solution to reduce the simulation time is to modify the input set so that that benchmark executes fewer instructions. However, the stipulation is that a benchmark that uses the modified input set must have the same characteristics as when it is not using the modified input set. Failure to uphold that stipulation defeats the purpose of using benchmarks that are similar to "real-world" programs.

KleinOsowski and Lilja [KleinOsowski02] produced the MinneSPEC reduced input set for the SPEC CPU 2000 benchmarks. Benchmarks that use MinneSPEC reduced input sets ostensibly have reference-like characteristics (function-level execution patterns, instruction mixes, and memory behaviors), albeit with a much shorter simulator time. The input sets were reduced by modifying the command-line parameters, truncating the input set, or creating a completely new input set. For each benchmarks, they tried to create three reduced input sets: **small**, **medium**, and **large** that produced approximately 100 million, 500 million, and one billion, respectively, dynamic instructions.

To measure the fidelity of the MinneSPEC reduced input, as compared to the reference input set, they used two metrics. The first metric used the chi-squared test to measure the "goodness-of-fit" between the instruction mixes of MinneSPEC reduced and the reference input sets. The second metric did the same by comparing the function-level execution profiles for each input set. (A function-level profile is the set of times that is spent executing each function.) Their results showed that half of the benchmarks had statistically similar function-level execution profiles for both the MinneSPEC and reduced input sets. A slightly higher percentage of benchmarks had a statistically similar instruction mix profiles for both input sets.

While the MinneSPEC reduced and reference input sets are similar in some respects, KleinOsowski and Lilja showed that, for some benchmarks, the memory performance (as exemplified by the cache miss rate) can be quite different.

Another way of reducing the simulation time is to perform detailed (slow) simulations on some parts of the program while performing functional (fast) simulations on the other parts of the program. One problem with this approach is that the processor state coming out of the functional simulation reflects processor state that was present when going into the functional simulation.

To address this problem, Haskins and Skadron [Haskins01] proposed Minimal Subset Evaluation as a way to decrease the simulation time of the program's warm-up phase by probabilistically determining a minimal set of transactions that are necessary for a sufficiently accurate cache state. More specifically, they used a "crude heuristic" to determine the number of memory accesses that need to occur before the end of the fast-forwarding to achieve a cache state that is statistically similar to the cache state without fast-forwarding. They used separate formulas for the direct-mapped and set-associative caches to reduce the computation time.

Their results showed that this approach, for a 99.9% probability of achieving an accurate processor state, decreased the simulation time by an average of 47% with only a 0.3% error in the IPC. For a 95% probability, their approach decreased the simulation time by an average of 60% while incurring a 0.4% error.

Finally, the third way of reducing the simulation time is to determine a group of representative program intervals that could be substituted for the entire program. Using this way, the computer architect can simulate only those samples in lieu of executing the entire benchmark. Alternatively, the architect can perform detailed simulations on those samples while fast-forwarding between them. Using this approach can dramatically reduce the execution time while, if properly done, minimizing the error.

One implementation of this way is given in [Sherwood02]. In this paper, Sherwood *et al* used profiling to determine a group of representative program samples that could be substituted for the entire program. To characterize each program sample, they used the basic block execution frequency. Associated with each program sample is a basic block vector that contains the execution frequencies of each basic block in that program sample. Then, to determine the similarity of program samples, they calculated the Euclidean distances between vectors. Two program samples are similar if there is small Euclidean distance between them. After calculating the Euclidean distances, they chose the most dissimilar program samples as the representative set of program samples.

Their results showed that this method could decrease the simulation time of the reference input set by over a hundred or even a thousand times with only a 17% IPC error when using a single program sample and a 3% IPC error when using multiple program samples. By comparison, blindly fast-forwarding has an 80% IPC error without a comparable reduction in the simulation time.

7.4.3. Benchmark and Input Set Characterization

In most cases, simulating all of the benchmarks and input sets in a benchmark suite is not a tractable problem. To reduce the simulation time, computer architects usually simulate only a sub-set of the benchmarks in a benchmark suite. However, if the benchmarks in this sub-set are too similar with respect to each other, then the simulation results may be skewed. To address this problem, the following two papers propose solutions that classify benchmarks and determine a minimal set of benchmarks to run.

Eeckhout *et al* [Eeckhout02] used statistical data analysis techniques to determine the statistical similarity of benchmark and input set pairs. To quantify the similarity, they used metrics such as the instruction mix, the branch prediction accuracy, the data and instruction cache miss rates, the number of instructions in a basic block, and the maximum amount of parallelism inherent to the benchmark. After characterizing each benchmark with the aforementioned metrics, they used statistical approaches such as principal component analysis and cluster analysis to actually cluster the benchmarks and input set pairs together.

The key difference between their method of grouping benchmarks and the method presented in Chapter 4 is that their method is predicated on defining a set of metrics that encompass all of the key factors that affect the performance. The deficiency of their approach is that it assumes that all significant metrics have been incorporated into the statistical design without the benefit of simulations. However, since it is possible for two unrelated processor parameters to interact, picking metrics to identify the effect of either parameter does not necessarily cover the effect of their interaction. The approach given in this dissertation, on the other hand, does not make that assumption; instead, all parameters are weighted equally. Finally, their method requires a redefinition of the metrics if it were be used to classify benchmarks based on other metrics, such as the power consumption, while this method does not require any redefinition.

Taking a different approach to the same problem, Giladi and Ahituv [Giladi95] identified the “redundant” benchmarks in the SPEC 92 benchmark suite. They defined a redundant benchmark to be one that can be removed from the benchmark suite without significantly affecting the resulting SPEC number. In theory, the SPEC number measures the performance of a computer system across a wide range of programs. The SPEC number is generated by normalizing each benchmark’s execution time to a baseline system and then computing the geometric mean of the results.

Their results show that 13 of the 20 benchmarks in the SPEC 92 suite were redundant. In other words, the conclusion of their approach is that only seven benchmarks need to be simulated and that those seven adequately represent all 20.

This method of determining redundant benchmarks is significantly different from the one proposed in Chapter 4 for at least two reasons. First of all, this method is completely based on approximating the SPEC number. Secondly, since the SPEC number is calculated by using the benchmark's execution time and by normalizing the execution times to a baseline system, there is no direct connection to the effect that each benchmark has on the processor. However, the method in this dissertation focuses exclusively on the benchmark's effect on the processor.

7.4.4. Processor Parameter Analysis

One problem in computer architecture that is not very understood is the effect that different processor components have on the processor's performance. While it is relatively simple to understand the effect that the size or number of a component has on the performance, what complicates this problem is that the processor components interact in complex ways. Since these components and their interactions could significantly affect the processor's performance, it is important to understand their effect. The remainder of this section describes the related work that analyzes the effect of various processor components.

Skadron *et al* [Skadron99] performed an in-depth study of the trade-offs between the instruction-window size (i.e. number of ROB entries), branch prediction accuracy, and the sizes of the L1 caches. Their paper performed a set of detailed sensitivity analyses that examined the IPC for different instruction-window sizes, data and instruction cache sizes, and different branch prediction accuracies using the integer benchmarks of the SPEC 95 benchmark suite.

When evaluating the effect of a pair of parameters, they fixed two of the four parameters while varying the other two. For example, to determine the effect that L1 D-Cache and L1 I-Cache sizes has on the performance (IPC), they fixed the branch prediction accuracy to be 100% while using a 128-entry instruction window.

While their results were very detailed and had several meaningful conclusions, these results and conclusions cannot be taken completely a face value for a couple of reasons. First of all, before they performed their sensitivity analyses, they did not determine the important parameters and interactions. As a result, some of the important parameters and interactions may have a disproportionate and unknown effect on the results. Second, the values of the fixed parameters also can have a significant impact on the results. The values for fixed parameters can also have a large, unknown effect by establishing a baseline result that is unrealistically high or low.

In summary, the related work described in this section has focused on simulation validation, reducing the simulation time, benchmark and input set characterization, and processor parameter analysis. The goal of the first three topics is to improve the accuracy of the simulation results while the goal of the fourth topic is to gain a deeper understanding of how the various processor components affect its performance. While these two goals are somewhat similar to the benefits of using statistically rigorous simulation methodology, the key difference between prior work and this dissertation is that the primary focus of this dissertation is improving simulation methodology. As a result, this dissertation covers its steps of the simulation methodology in more depth and also by basing its recommendations on a statistical foundation.

Chapter 8

Future Work

This chapter describes some future work related to Instruction Precomputation, to exploiting trivial computations, and, especially, to improving simulation methodology.

8.1. Instruction Precomputation

One potential problem with the current implementation of Instruction Precomputation is that the lifetime of the unique computations that are in the Precomputation Table could be very short. For example, the unique computation 0+0 may be heavily used for initialization purposes at the beginning of the program, and used relatively infrequently afterwards. Therefore, although this particular unique computation has a very high frequency of execution overall, its frequency of execution is much lower after the program's initialization section. More generally, since a unique computation's frequency of execution may be very high in some parts of the program and very low in others, the Precomputation Table could be redesigned to allow the Precomputation Table to replace unique computations that may have a low frequency of execution in the near future with high frequency ones. However, instead of dynamically replacing one unique computation at a time, to reduce the access time to the Precomputation Table (which is affected by the number of ports), the entire table could be updated at a single time. Of course, while the table is being updated, its entries cannot be accessed. Accordingly, since the compiler – through feedback-directed optimization – would determine what unique computations should be placed in the Precomputation Table at what time, the Precomputation Table would then be similar to a software managed cache.

In this case, the Precomputation Table would not need to be as large as when the table is not updated since the table needs only to hold the unique computations that could be used in the near future. Furthermore, instead of only at the beginning of the program, updating the Precomputation Table periodically may also yield better speedups when compared to using a single, monolithic Precomputation table.

Another possibility to improve the performance and the efficiency – as measured by table area – of Instruction Precomputation is to combine it with the hardware for Trivial Computation Simplification and Elimination. The advantage of combining these two approaches is that Instruction Precomputation can use the Trivial Computation Simplification and Elimination hardware to "filter" unique computations that are trivial out of the Precomputation Table. As a result, the Precomputation Table will be filled only with unique computations that are not trivial. Consequently, the combination of these two approaches will target a larger number of unique computations (for the same table size) or will target the same number of unique computations (for a smaller table).

8.2. Simplification and Elimination of Trivial Computations

Although the results in Chapter 6 showed that hardware can be used to reduce the execution time of the program by Simplifying and Eliminating Trivial Computations, a potentially more cost-effective solution is to use the compiler to statically simplify or eliminate any trivial computations. However, the efficacy of this alternative approach depends on the cause of trivial computations. In other words, why do typical programs have such a large percentage of trivial computations?

If trivial computations are strictly correlated to a specific input set, then it is probably more effective to use hardware to simplify and eliminate the trivial computations since the compiler may not be able to simplify or eliminate that trivial computation for all input sets. On the other hand, if the trivial computations are primarily a function of the program, then the compiler may be able to simplify or eliminate a large number of trivial computations since the trivial computations are independent of the input set. However, to determine which situation predominates, the causes of why a program executes so many trivial computations need to be identified.

8.3. Improving Computer Architecture Simulation and Design Methodology

Chapter 4 described the six major steps of the simulation process and the improvements to Steps 3, 4, and 6 (processor parameter value selection, benchmark selection, and enhancement analysis) of the simulation process proposed by this dissertation. Section 8.3.1 describes two potential improvements that could be applied to Steps 3 and 5 of the simulation process while Section 8.3.2 describes a potential improvement for processor design methodology.

8.3.1. Simulation Methodology

In step 3 of the simulation process, the computer architect chooses values for the different user-configurable processor parameters. Chapter 4 described a procedure for using a Plackett and Burman design to aid the architect in choosing an appropriate set of processor parameter values. However, when choosing processor parameters values, the architect needs to realize that using different input sets could result in very different program characteristics.

For example, one of the most important program characteristics is the miss rate of the L1 D-Cache. However, for several benchmarks in the SPEC 2000 benchmark suite, the L1 D-Cache miss rate when using the reference input set is statistically different that the L1 D-Cache miss rate when using the MinneSPEC large reduced, test, or train input sets [Yi02-2]. Therefore, if the architect incorrectly assumes that the L1 D-Cache miss rate of the test input set is fairly similar to the L1 D-Cache miss rate of the reference input set, then the memory performance of the processor will appear to be much better when using the test input set since the memory parameters were not adjusted appropriately. Therefore, although the architect may use the Plackett and Burman design to help choose processor parameter values, that careful effort could be negated by basing those values on an incorrect premise, namely, that the reduced and reference cache miss rates are the same.

One way of salvaging the utility of the reduced input sets – which are attractive since they have shorter-than-reference simulation times – is to reduce the cache size and associativity when using reduced input sets. The problem with this approach is that, to normalize the cache miss rates of the reduced and reference input sets, each benchmark may require a different cache size and associativity. Therefore, one benchmark may need to use an 8 KB, 1-way cache while another benchmark may need to use a 64 KB, 4-way cache.

While this non-uniformity of cache sizes and associativity across benchmarks minimizes the differences in the cache miss rates when using the reduced input set – which decreases the amount of error in the simulation results – this non-uniformity makes it very difficult for the architect to evaluate the performance of any memory-based enhancement since the efficacy of the enhancement may depend on the actual cache size or associativity.

An additional problem when scaling the cache size and associativity to minimize the difference in the cache miss rates is that there may be multiple cache configurations that appear to have approximately the same effect. However, since some of the cache configurations make seem a little “extreme” (i.e. using a 1 KB, 1-way cache for a reduced input set instead of a 128 KB, 8-way cache for the reference input set) the architect may opt to use less “extreme” cache configurations. The key question when deciding whether to use cache configuration A or cache configuration B is: What is the error for each configuration? The cache configuration that minimizes the differences in the cache miss rates of the reduced and reference input sets may have about the same amount of error as the cache configuration that is second best.

Therefore, as an item of future work that aims to improve the simulation methodology of Step 3, the first step is to quantify the amount of error in IPC that exists when reduced caches are not used with reduced input sets. Then, the second step is to determine a set of cache configurations that minimize the error.

In step 5 of the simulation process, the computer architect actually performs the simulations. There are at least four main ways to reduce the simulation time of the benchmarks, when it uses the reference input set. First, the architect could use reduced input sets like those from the MinneSPEC benchmark suite. Second, the architect could opt to fast-forward (functional simulation only) through the initialization section of the program and then resume normal simulation (full-timing and modeling). Third, the architect could modify the simulator to periodically fast-forward through parts of the program while performing full-simulation on the samples in-between. Fourth, the architect could simulate a fixed-number of instructions and then terminate the simulation at that point.

Ostensibly, these approaches reduce the total simulation time while preserving the characteristics of the reference input sets. However, the fidelity of these approaches has not been comprehensively and comparatively established. In addition to the possibility that some of these approaches may incur a lower amount of error than the others, some of these approaches also may be more appropriate for testing the effect of certain enhancements. For example, fast-forwarding through the initialization section of computationally-intensive benchmarks may under-represent the effect of trivial computation elimination since compilers often use 0+0 to clear the value of a register or a memory location. Also, since the initialization section may account for a significant percentage of the program’s total execution time, fast-forwarding through the initialization may inflate an enhancement’s speedup.

Therefore, to determine which approach is best for power reduction and performance evaluation, each of these approaches will be characterized to determine which approach is most similar to the reference input set, which approach has the least amount of error, and which approach is most suitable for evaluating the efficacy of different types of enhancements.

8.3.2. Design Methodology

One problem with processor design methodology is that new processors running future programs are designed using old processors running dated programs. So, how does a computer architect design and evaluate the performance of a future processor using the tools of the past (short of inventing a time machine)? Of course, the fundamental issue of this problem is summarized by the following question: What is the difference in performance (and power consumption) for the expected (i.e. simulated) and actual future processors? To address this issue, the difference in the expected performance of the future processor running past and future benchmarks with past and future compiler options will be quantified.

Chapter 9

Conclusion

The performance of superscalar processors is limited by the amount of instruction-level parallelism (ILP), which in turn is limited by control and data dependences between instructions. This dissertation describes two microarchitectural techniques, Instruction Precomputation and the Simplification and Elimination of Trivial Computations, which increases the amount of ILP.

Instruction Precomputation improves the performance of a processor by dynamically eliminating instructions that are redundant computations. A redundant computation is one that the processor previously executed. Instruction Precomputation uses the compiler to determine the highest frequency unique computations, which are loaded into the Precomputation Table before the program begins execution. For redundant computations, instead of re-computing its result, the output value is forwarded from the matching entry in the Precomputation Table to the instruction and then the instruction is removed from the pipeline.

The results in this dissertation show that a small number of unique computations account for a disproportionate number of dynamic instructions. More specifically, less than 0.2% of the total unique computations account for 14.68% to 44.49% of the total dynamic instructions. When using the highest frequency unique computations from Input Set B while running Input Set A (Profile B, Run A), a 2048-entry Precomputation Table improves the performance of a base 4-way issue superscalar processor by an average of 10.53%. This speedup is very close to the upper-limit speedup of 10.87%. This speedup is higher than the average speedup of 7.43% that value reuse yields for the same processor configuration and the same benchmarks, but value reuse requires slightly more hardware. More importantly, for smaller table sizes (16-entry), Instruction Precomputation outperforms value reuse, 4.47% to 1.82%. Finally, the results show that the speedup due to Instruction Precomputation is the approximately same regardless of which input set is used for profiling and regardless of how the unique computations are selected (frequency or frequency/latency product).

Overall, there are two key differences between Instruction Precomputation and value reuse. First of all, Instruction Precomputation uses the compiler to profile the program to determine the highest frequency unique computations while value reuse does its profiling at run-time. Since the compiler has more time to determine the highest frequency unique computations, the net result is that Instruction Precomputation yields a much higher speedup than value reuse does for a comparable amount of chip area. Second of all, although using the compiler to manage the PT eliminates the need for additional hardware to dynamically update the PT, it can dramatically increase the compile time since the compiler must profile the program.

Trivial computations are computations where the output value is zero, one, equal to one of the input values, or a shifted version of one of the input values. Examples of trivial computations include: 0+X, X*0, and X/X. The results in this dissertation show that, for 12 selected SPEC 2000 benchmarks, 12.24% of all dynamic instructions are trivial computations. For the five selected benchmarks from the MediaBench benchmark suite, trivial computations account for 5.73% of all dynamic instructions.

This dissertation has demonstrates that since a significant percentage of a program's instructions are trivial computations, simplifying or eliminating these trivial computations can improve the processor's performance. A processor simplifies a trivial computation by converting it to a less complex (i.e. lower latency) instruction type with different operands, but that will still produce the correct result. On the other hand, a processor eliminates a trivial computation simply by removing the instruction from the pipeline and selecting the correct output value (0, 1, 0xffffffff, or the value of the other input operand).

The results in this dissertation show that dynamically simplifying and eliminating trivial computations improves the performance of a base 4-way issue superscalar processor by an average of 8.86% for 12 SPEC 2000 benchmarks and by 4.00% for five MediaBench benchmarks. Additionally, simplifying and eliminating trivial computations also improves the performance of a processor that does not have any functional unit constraints (i.e. where the number of each type of functional unit is equal to the issue width of the processor) by an average of 6.60% (SPEC 2000) and 2.92% (MediaBench).

Overall, Simplifying and Eliminating Trivial Computations yields fairly impressive speedups at a relatively low hardware cost. This proposed enhancement is particularly novel because it improves the processor's performance with early non-speculative instruction execution. This allows the processor to exceed the dataflow limit without requiring verification of a prediction and misprediction recovery.

While these new techniques can improve the performance, the primary focus of this dissertation is on improving the quality of simulation methodology. More specifically, this dissertation describes how a statistical Plackett and Burman design can be used to improve the way user-configurable processor parameter values are chosen, benchmarks are chosen, and finally processor enhancements are analyzed. When choosing processor parameter values, a computer architect can use a Plackett and Burman design to identify the key processor parameters that have a disproportionate effect on the processor's performance. Identifying the key processor parameters is an extremely difficult task since unknown interactions can severely skew the results. Since a Plackett and Burman design is able to quantify the effect of the most significant interactions, the architect can confidently use that design to help determine the most significant parameters. After identifying the key parameters, the architect can carefully choose values for those parameters and then choose values for the remaining parameters. The values for the remaining parameters do not need to be chosen with as much care since they have less effect on the results than do the key parameters.

The results in Section 6.3.1 demonstrated that a Plackett and Burman design could efficiently identify the most significant processor parameters. In this case, the most significant processor parameters in *sim-outorder* of the SimpleScalar tool set for the 12 C benchmarks of the SPEC 2000 benchmark suite were the:

1) Number of Reorder Buffer Entries
2) L2 Cache Latency
3) Branch Predictor (i.e. the Branch Prediction Accuracy)
4) Number of Integer ALUs
5) L1 D-Cache Latency
6) L1 I-Cache Size
7) L2 Cache Size
8) L1 I-Cache Block Size
9) Memory Latency of the First Block
10) Number of LSQ Entries

After realizing that these parameters have the most effect on the processor's performance, choosing their values and the values of the other processor parameters is fairly simply.

Since the results of a Plackett and Burman design is a vector of ranks, those results can also be used to aid the architect in choosing a set of benchmarks that are either distinct or similar – depending on what is appropriate to evaluate the performance of the architect's enhancement. If all of the parameters' ranks for two benchmarks are similar, then those two benchmarks have a similar effect on the processor. After calculating the Euclidean distance between vectors, which represents the amount of dissimilarity between those two benchmarks, the results can be displayed using a dendrogram.

If the architect wishes to select N benchmarks, the architect moves a horizontal line up from a dissimilarity of zero. When the horizontal line intersects N vertical lines, the benchmarks have been categorized into N groups. Then, to select the final set of benchmarks, the architect needs only to select one benchmark from each group. In the example in Section 6.3.2, *gzip*, *vpr-Place*, *vpr-Route*, *gcc*, *art*, *mcf*, *equake*, and *ammp* form the final set of benchmarks.

The final application of a Plackett and Burman design that is proposed in this dissertation is to use it to analyze the effect of a processor enhancement. For each parameter, by comparing its average sum-of-ranks in the base processor against its average sum-of-ranks in the processor with the enhancement, the architect can see how the enhancement affects the processor. For example, adding Instruction Precomputation to the processor will primarily improve that processor's performance by decreasing the amount of functional unit contention. (Instruction Precomputation also improves the processor's performance by decreasing the execution latency of redundant computations.) In other words, Instruction Precomputation addresses the performance bottleneck of busy functional units.

Adding hardware to Simplify and Eliminate Trivial Computations does not significantly relieve or exacerbate any performance bottlenecks. Rather, the order of and the sums-of-ranks for the top ten most significant parameters are essentially the same. Therefore, it is concluded that this processor enhancement more or less uniformly affects all processor parameters (i.e. it does not markedly mitigate or create any bottlenecks).

Overall, one of the key contributions of this dissertation is that it advocates the use of statistically-based simulation methodology. Since architects usually approach the simulation process in an ad-hoc manner and since no prior work has explicitly focused on improving simulation methodology, using Plackett and Burman designs represents a fundamental improvement in simulation methodology. In particular, Plackett and Burman designs are effective in helping the computer architect choose processor parameter values and benchmarks, and analyze the effect of processor enhancements. The cost of this approach is that it requires a few extra simulations.

In conclusion, this dissertation shows the following four key results. First, the results show that significant amounts of redundant and trivial computations exist in typical programs. Second, the Instruction Precomputation approach of statically determining the highest frequency unique computations yields higher speedups than the value reuse approach of dynamically determining the redundant computations. Third, Simplifying and Eliminating Trivial Computations can also significantly improve a processor’s performance. Finally, a Plackett and Burman design can be used to improve simulation methodology by helping the architect choose processor parameters and benchmarks, and analyze the effect of a processor enhancement.

Bibliography

[Azam97]	M. Azam, P. Franzon, and W. Liu, “Low Power Data Processing by Elimination of Redundant Computations”, International Symposium on Low Power Electronics and Design, 1997.
[Bannon97]	P. Bannon and Y. Saito, “The Alpha 21164PC Microprocessor”, International Computer Conference, 1997.
[Black98]	B. Black and J. Shen, “Calibration of Microprocessor Performance Models”, IEEE Computer, Vol. 31, No. 5, May 1998, Pages 59-65.
[Burger97]	D. Burger and T. Austin, “The SimpleScalar Tool Set, Version 2.0”, University of Wisconsin-Madison Computer Sciences Department Technical Report #1342, 1997.
[Cain02]	H. Cain, K. Lepak, B. Schwartz, and M. Lipasti, “Precise and Accurate Processor Simulation”, Workshop on Computer Architecture Evaluation using Commercial Workloads, 2002.
[Citron98]	D. Citron and D. Feitelson, “Accelerating Multi-Media processing by Implementing Memoing in Multiplication and Division Units”, International Conference on Architectural Support for Programming Languages and Operating Systems, 1998.
[Citron00-1]	D. Citron and D. Feitelson, “The Organization of Lookup Tables for Instruction Memoization”, Hebrew University of Jerusalem Technical Report: 2000-4.
[Citron00-2]	D. Citron and D. Feitelson, “Hebrew University of Jerusalem Technical Report: 2000-5”, Hebrew University of Jerusalem Technical Report: 2000-5.
[Desikan01]	R. Desikan, D. Burger, and S. Keckler, “Measuring Experimental Error in Microprocessor Simulation”, International Symposium on Computer Architecture, 2001.

[Edmondson95] J. Edmondson, P. Rubinfeld, and R. Preston, “Superscalar Instruction Execution in the 21164 Alpha Microprocessor”, IEEE Micro, Vol. 15, No. 2, March-April 1995, Pages 33-43.

[Gabbay98] F. Gabbay and A. Mendelson, “Using Value Prediction to Increase the Power of Speculative Execution Hardware”, ACM Transactions on Computer Systems, Vol. 16, No. 4, August 1998, Pages 234-270.

[Gibson00] J. Gibson, R. Kunz, D. Ofelt, M. Horowitz, J. Hennessy, and M. Heinrich, “FLASH vs. (Simulated) FLASH: Closing the Simulation Loop”, International Conference on Architectural Support for Programming Languages and Operating Systems, 2000.

[Giladi95] R. Giladi and N. Ahituv, “SPEC as a Performance Evaluation Measure”, IEEE Computer, Vol. 28, No. 8, August 1995, Pages 33-42.

[Glamm00] R. Glamm and D. Lilja, “Automatic Verification of Instruction Set Simulation Using Synchronized State Comparison”, Annual Simulation Symposium, 2001.

[Gonzalez98] A. Gonzalez, J. Tubella, and C. Molina, “The Performance Potential of Data Value Reuse”, University of Politecenica of Catalunya Technical Report: UPC-DAC-1998-23, 1998.

[Hennessy96] J. Hennessy and D. Patterson, “Computer Architecture: A Quantitative Approach”, (Second Edition), Morgan-Kaufman 1996.

[Henning00] J. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium", IEEE Computer, Vol. 33, No. 7, July 2000; Pages 28-35.

[Horel99] T. Horel and G. Lauterbach, “UltraSPARC-III: Designing Third-Generation 64-Bit Performance”, IEEE Micro, Vol. 19, No. 3, May-June 1999, Pages 73-85.

[Huang98] J. Huang and D. Lilja, “Improving Instruction-Level Parallelism by Exploiting Global Value Locality”, University of Minnesota Technical Report: HPPC-98-12, 1998.

[Huang99] J. Huang and D. Lilja, “Exploiting Basic Block Locality with Block Reuse”, International Symposium on High Performance Computer Architecture, 1999.

[Kessler98] R. Kessler, E. McLellan, and D. Webb, “The Alpha 21264 Microprocessor Architecture”, International Conference on Computer Design, 1998.

[Kessler99] R. Kessler, “The Alpha 21264 Microprocessor”, IEEE Micro, Vol. 19, No. 2, March-April 1999, Pages 24-36.

[KleinOsowski02] A. KleinOsowski and D.J. Lilja, “MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research”, Vol. 1, June 2002.

[Kumar97] A. Kumar, “The HP PA-8000 RISC CPU”, IEEE Micro, Vol. 17, No. 2, March-April 1997, Pages 27-32.

[Lee97] C. Lee, M. Potkonjak, and W. Mangione-Smith, “MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems”, International Symposium on Microarchitecture, 1997.

[Leiholz97] D. Leiholz and R. Razdan, "The Alpha 21264: A 500 MHz Out-of-Order Execution Microprocessor", International Computer Conference, 1997.

[Lilja00] D. Lilja, "Measuring Computer Performance", Cambridge University Press, 2000.

[Lipasti96-1] M. Lipasti, C. Wilkerson and J. Shen, "Value Locality and Load Value Prediction", International Conference on Architectural Support for Programming Languages and Operating Systems, 1996.

[Lipasti96-2] M. Lipasti and J. Shen, "Exceeding the Dataflow Limit via Value Locality", International Symposium on Microarchitecture, 1996.

[Matson98] M. Matson, D. Bailey, S. Bell, L. Biro, S. Butler, J. Clouser, J. Farrell, M. Gowan, D. Priore, and K. Wilcox, "Circuit Implementation of a 600 MHz Superscalar RISC Microprocessor", International Conference on Computer Design, 1998.

[Molina99] C. Molina, A. Gonzalez, and J. Tubella, "Dynamic Removal of Redundant Computations", International Conference on Supercomputing, 1999.

[Montgomery91] D. Montgomery, "Design and Analysis of Experiments" (Third Edition), Wiley 1991.

[Normoyle98] K. Normoyle, M. Csoppenszky, A. Tzeng, T. Johnson, C. Furman, and J. Mostoufi, "UltraSPARC-IIi: Expanding the Boundaries of a System on a Chip", IEEE Micro, Vol. 18, No. 2, March-April 1998, Pages 14-24.

[Plackett46] R. Plackett and J. Burman, "The Design of Optimum Multifactorial Experiments", Biometrika, Vol. 33, Issue 4, June 1946, Pages 305-325.

[Richardson92] S. Richardson, "Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation", Sun Microsystems Laboratories Technical Report SMLI TR-92-1, 1992.

[Sazeides97] Y. Sazeides and J. Smith, "The Predictability of Data Values", International Symposium on Microarchitecture, 1997.

[Silc99] J. Silc, B. Robic, and T. Ungerer, "Processor Architecture: From Dataflow to Superscalar and Beyond", Springer-Verlag, 1999.

[Sima97] D. Sima, T. Fountain, and P. Kacsuk, "Advanced Computer Architectures, A Design Space Approach", Addison Wesley Longman, 1997.

[Skadron99] K. Skadron, P. Ahuja, M. Martonosi, and D. Clark, "Branch Prediction, Instruction-Window Size, and Cache Size: Performance Trade-Offs and Simulation Techniques", IEEE Transactions on Computers, Vol. 48, No. 11, November 1999, Pages 1260-1281.

[Sodani97] A. Sodani and G. Sohi, "Dynamic Instruction Reuse", International Symposium on Computer Architecture, 1997.

[Sodani98] A. Sodani and G. Sohi, "An Empirical Analysis of Instruction Repetition", International Symposium on Architectural Support for Programming Languages and Operating Systems, 1998.

[Song94]	S. Song, M. Denman, and J. Chang, “The PowerPC 604 RISC Microprocessor”, IEEE Micro, Vol. 14, No. 5, October 1994, Pages 8-17.
[Tremblay96]	M. Tremblay and J.M. O'Connor, “UltraSparc I: A Four-Issue Processor Supporting Multimedia”, IEEE Micro, Vol. 16, No. 2, March-April 1996, Pages 42-50.
[Wang97]	K. Wang and M. Franklin, "Highly Accurate Data Value Prediction using Hybrid Predictors", International Symposium on Microarchitecture, 1997.
[Weinberg98]	N. Weinberg and D. Nagle, “Dynamic Elimination of Pointer-Expressions”, International Conference on Parallel Architectures and Compilation Techniques, 1998.
[Yeager96]	K. Yeager, “The MIPS R10000 Superscalar Microprocessor”, IEEE Micro, Vol. 16, No. 2, March-April 1996, Pages 28-40.
[Yi02-1]	J. Yi and D. Lilja, “Effects of Processor Parameter Selection on Simulation Results”, MSI Report 2002/146, 2002.
[Yi02-2]	J. Yi and D. Lilja, “Cache Scaling for Realistic Memory Behavior in Processor Simulations When Using Reduced Input Sets”, Unpublished Technical Report, 2002.

Appendix A – Supplemental Results

This Appendix shows the figures and/or tables for three sets of results: 1) The profiling results for the amount of redundant and trivial computations when using a second input set; 2) The speedup results due to Instruction Precomputation for all possible combinations of input sets, frequency, and the frequency and latency product; and 3) The speedup results due to Simplifying and Eliminating Trivial Computations when using a second input set. With the exception of some of the Instruction Precomputation speedup results, each of these figures and tables in this Appendix has a corresponding figure or table in Chapters 2, 3, or 6. Specifically, with the exceptions of Figures A6.1.4.2 to A6.1.4.5, the name of each figure or table is based on the name of its corresponding figure or table, respectively. For example, Figure 3.2.1 shows the amount of trivial computation present in select SPEC 2000 and select MediaBench benchmarks with one input set; the corresponding figure, when using another input set, is Figure A3.2.1.

A1 Amount of Global-Level Redundant Computation

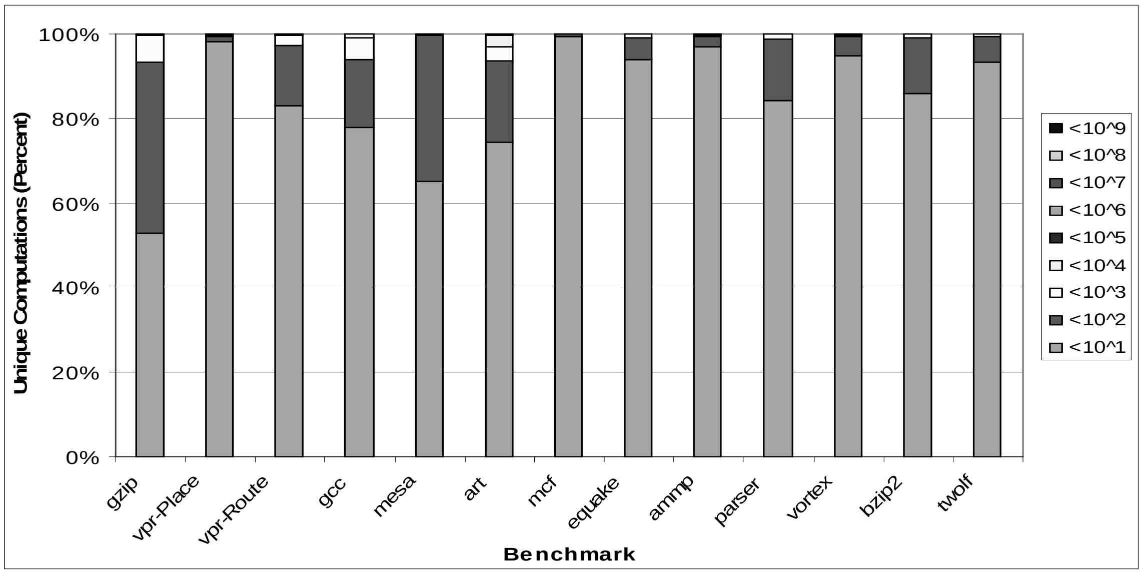

Figure A2.2.1.1: Frequency Distribution of Unique Computations per Benchmark, Global-Level, Normalized, Input Set B

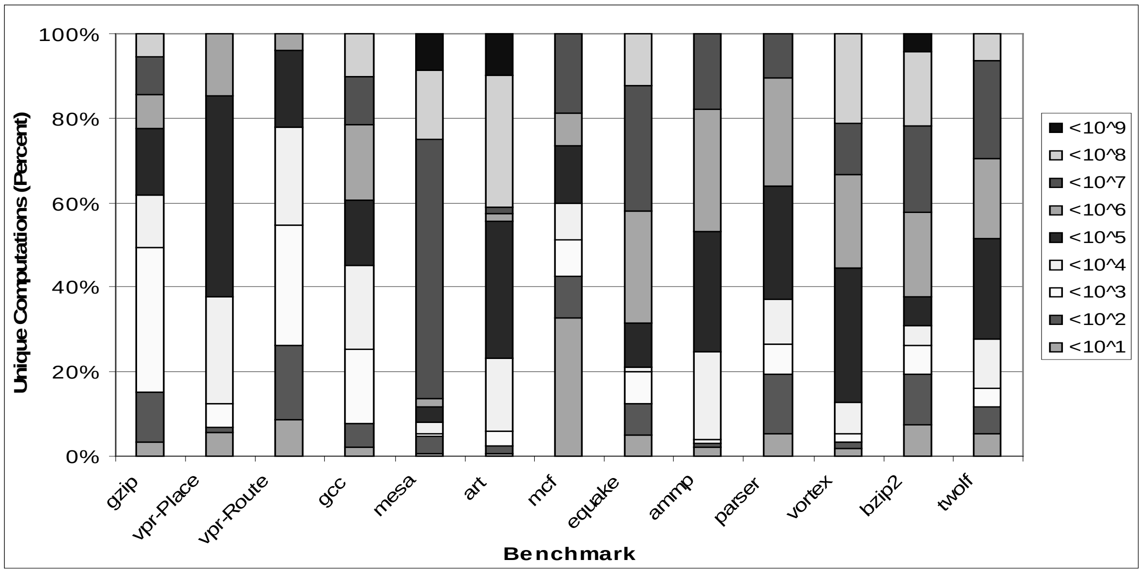

Figure A2.2.1.2: Percentage of Dynamic Instructions Due to the Unique Computations in each Frequency Range, Global-Level, Normalized, Input Set B

Table A2.2.1.1: Characteristics of the Unique Computations for the Top 2048 Global-Level Unique Computations, Input Set B

Benchmark	% of Unique Computations	% of Total Instructions
gzip	0.020	13.94
vpr-Place	0.258	41.85
vpr-Route	0.804	28.61
gcc	0.011	25.45
mesa	0.009	38.35
art	0.012	16.44
mcf	0.005	17.39
equake	0.004	28.71
ammp	0.168	29.78
parser	0.028	26.51
vortex	0.014	24.64
bzip2	0.002	30.64
twolf	0.007	22.04

A2 Amount of Local-Level Redundant Computation

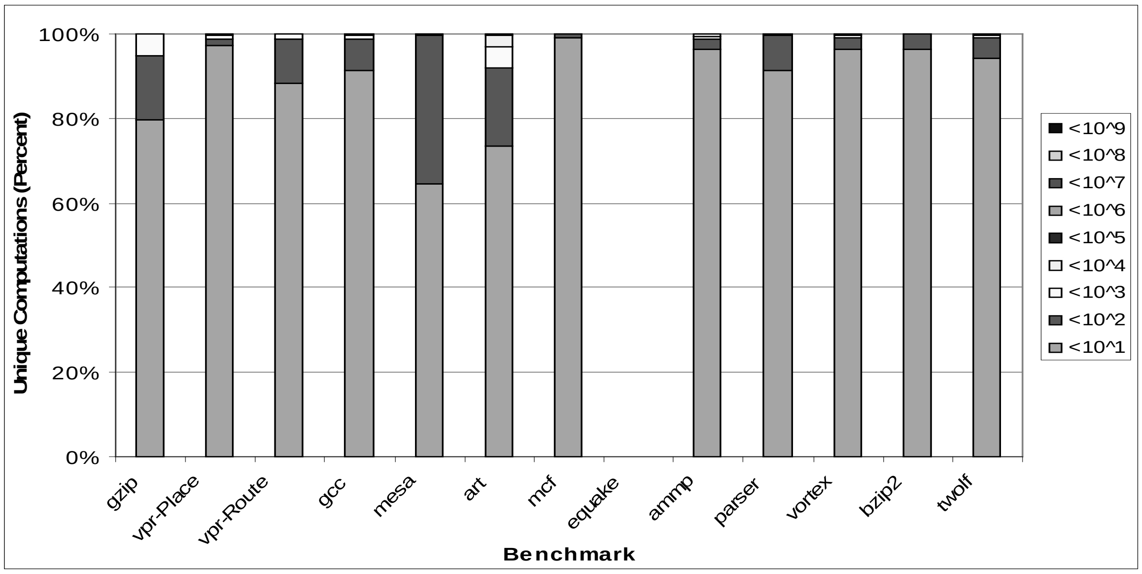

Figure A2.2.2.1: Frequency Distribution of Unique Computations per Benchmark, Local-Level, Normalized, Input Set

(*) Results for *equake* not presented due to a problem with the Origin 3800 system

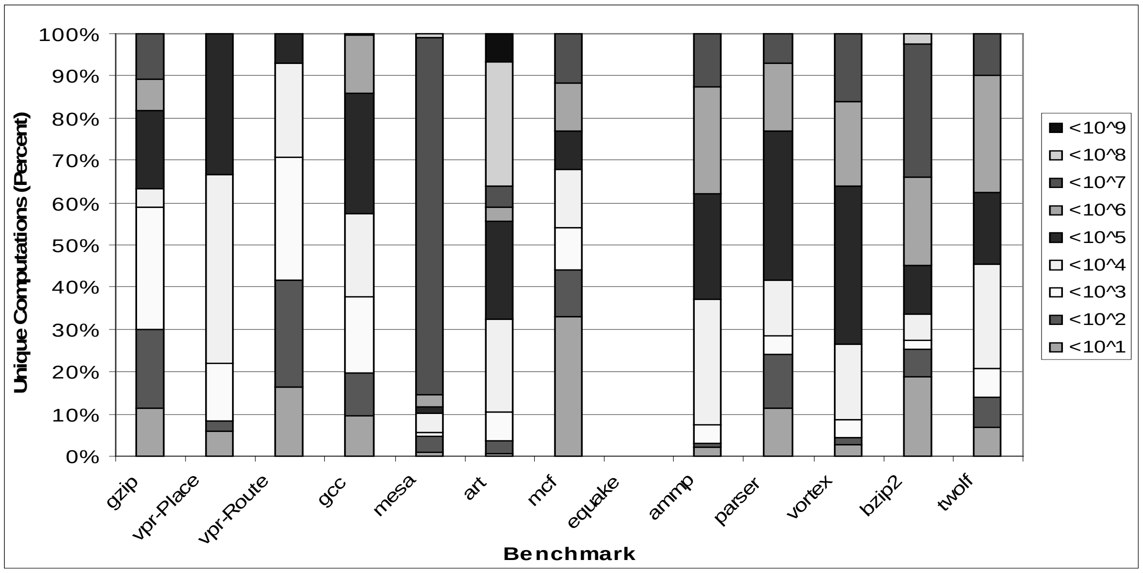

Figure A2.2.2.2: Percentage of Dynamic Instructions Due to the Unique Computations in each Frequency Range, Local-Level, Normalized, Input Set B

(*) Results for *equake* not presented due to a problem with the Origin 3800 system

A3 Comparison of the Amount of Global and Local Level Redundant Computation

Table A2.2.2.1: Percentage of Instructions Due to the 2048 Highest Frequency Unique Computations at the Global and Local Levels, Input Set B

Benchmark	Global	Local	Global - Local
gzip	13.94	11.13	2.81
vpr-Place	41.85	35.24	6.61
vpr-Route	28.61	19.90	8.72
gcc	25.45	12.96	12.49
mesa	38.35	37.40	0.95
art	16.44	15.93	0.51
mcf	17.39	15.05	2.34
equake	28.71	0.00	28.71
ammp	29.78	24.21	5.57
parser	26.51	24.94	1.57
vortex	24.64	20.01	4.63
bzip2	30.64	25.60	5.04
twolf	22.04	16.51	5.53

A4 Amount of Trivial Computation

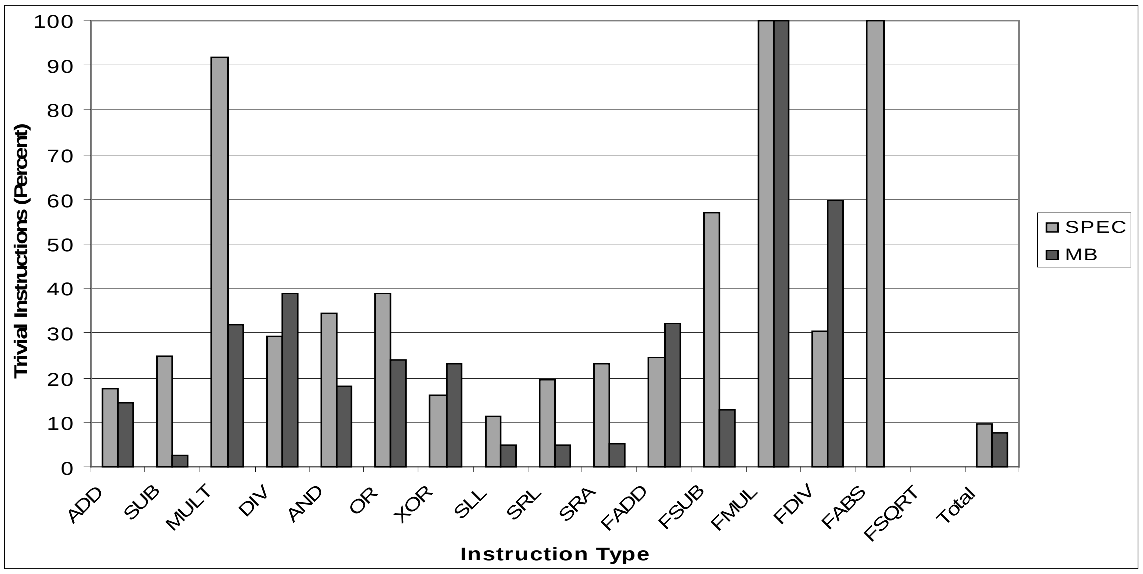

Figure A3.2.1: Percentage of Trivial Computations per Instruction Type and per Total Number of Dynamic Instructions for the SPEC and MediaBench Benchmarks

Table A4.1: Selected MediaBench Benchmarks and Input Sets (Dynamic Instruction Count in Millions of Instructions), Input Set B

Benchmark	Input Set Name	Instr. (M)
adpcm-Decode	S_16_44.adpcm	8.7
adpcm-Encode	S_16_44.pcm	10.5
epic-Compress	test_image.pgm	55.4
epic-Uncompress	test.image.pgm.E	10.3
g721-Decode	S_16_44.g721	408.4
g721-Encode	S_16_44.pcm	434.1
mpeg2-Decode	options.par	1180.8
mpeg2-Encode	tek6.m2v	1171.1
pegwit-Decrypt	pegwit.dec	15.9
pegwit-Encrypt	plaintext.doc	28.7
pegwit-Pub-Key	my.sec	12.7

A5 Speedup Due to Instruction Precomputation

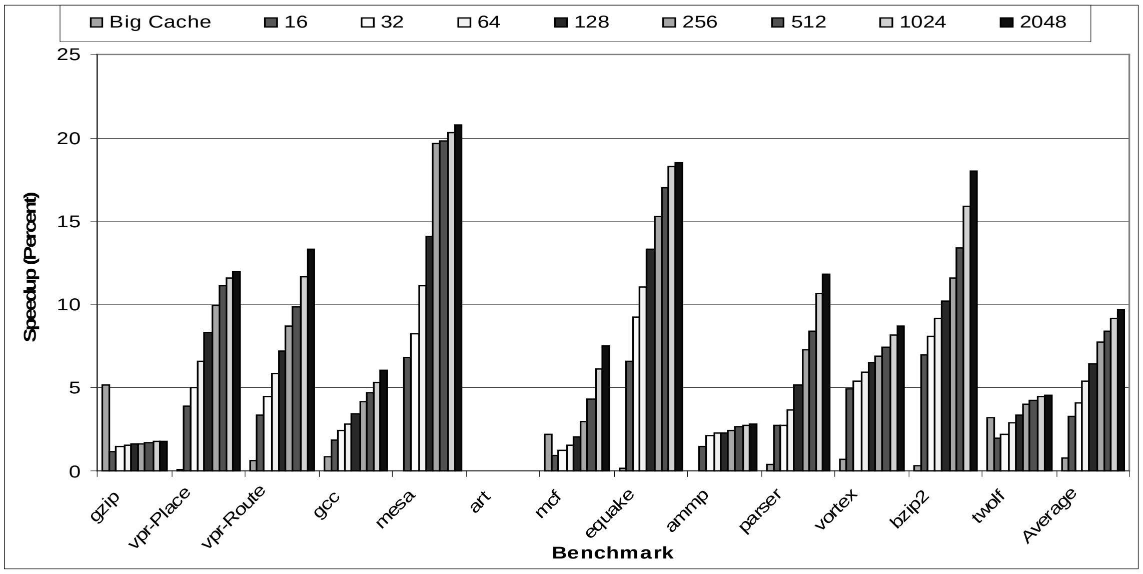

Figure A6.1.1.1: Speedup Due to Instruction Precomputation; Profile Input Set B, Run Input Set B, Frequency

(*) Results for *art* not presented due to a problem with the Netfinity system

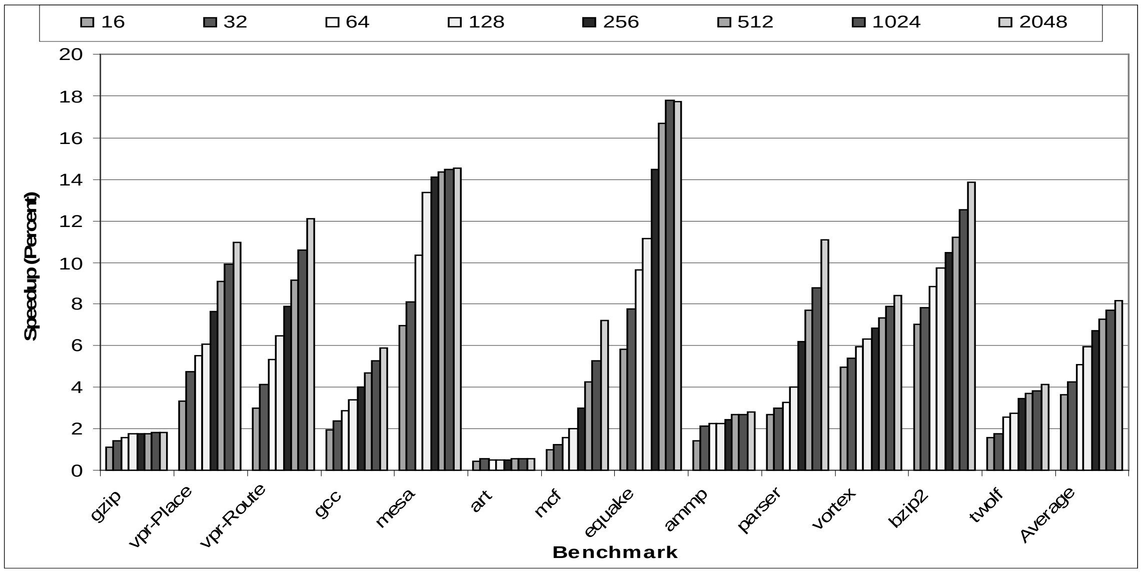

Figure A6.1.2.1: Speedup Due to Instruction Precomputation; Profile Input Set A, Run Input Set B, Frequency

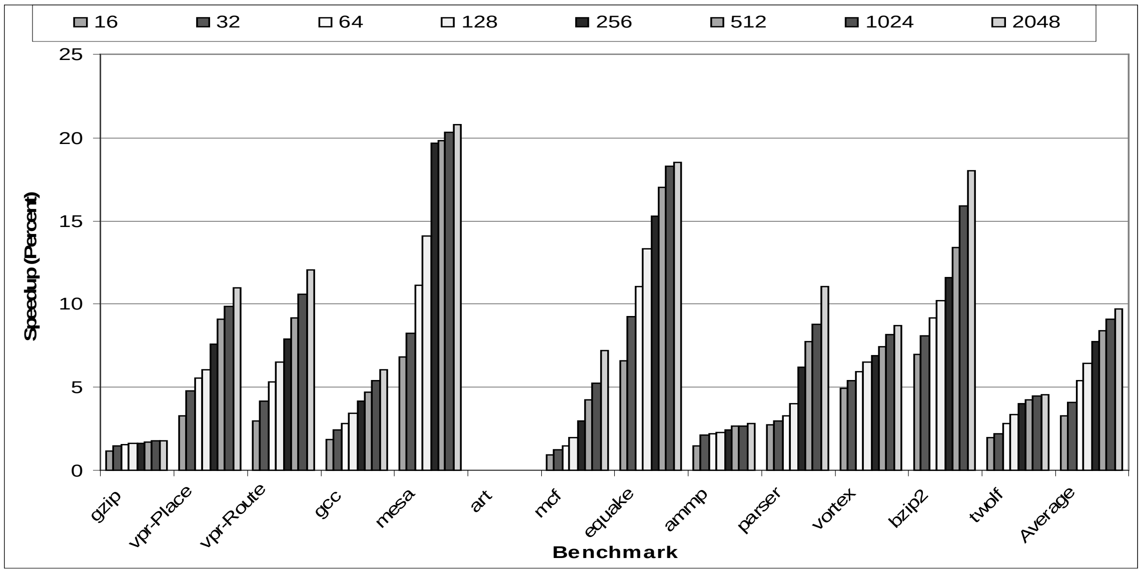

Figure A6.1.3.1: Speedup Due to Instruction Precomputation; Profile Input Set AB, Run Input Set B, Frequency

(*) Results for *art* not presented due to a problem with the Netfinity system

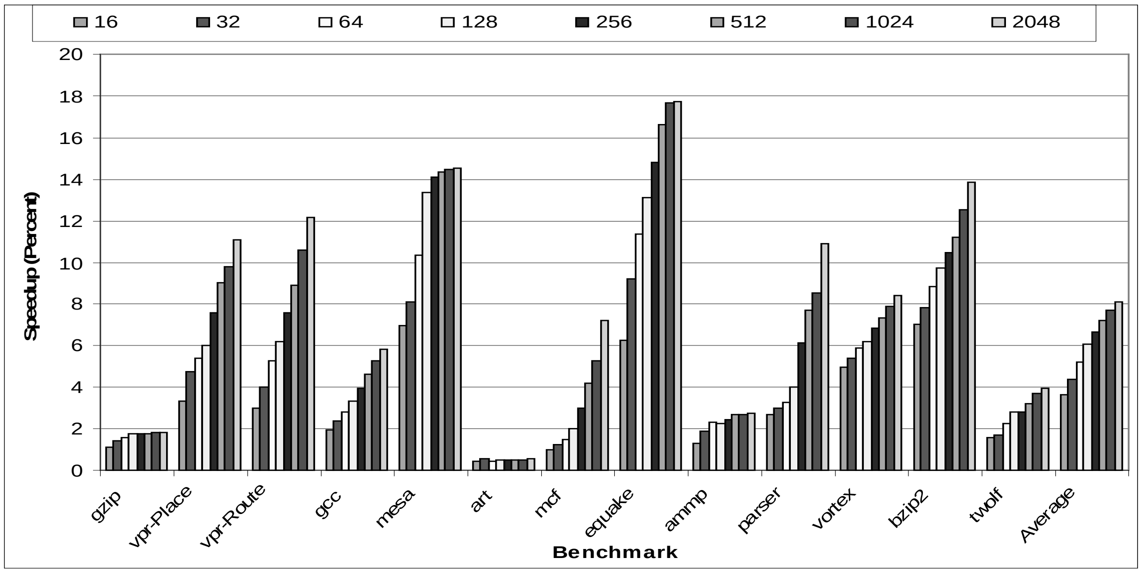

Figure A6.1.4.1: Speedup Due to Instruction Precomputation for the Highest Frequency and Latency Product Unique Computations; Profile A, Run B

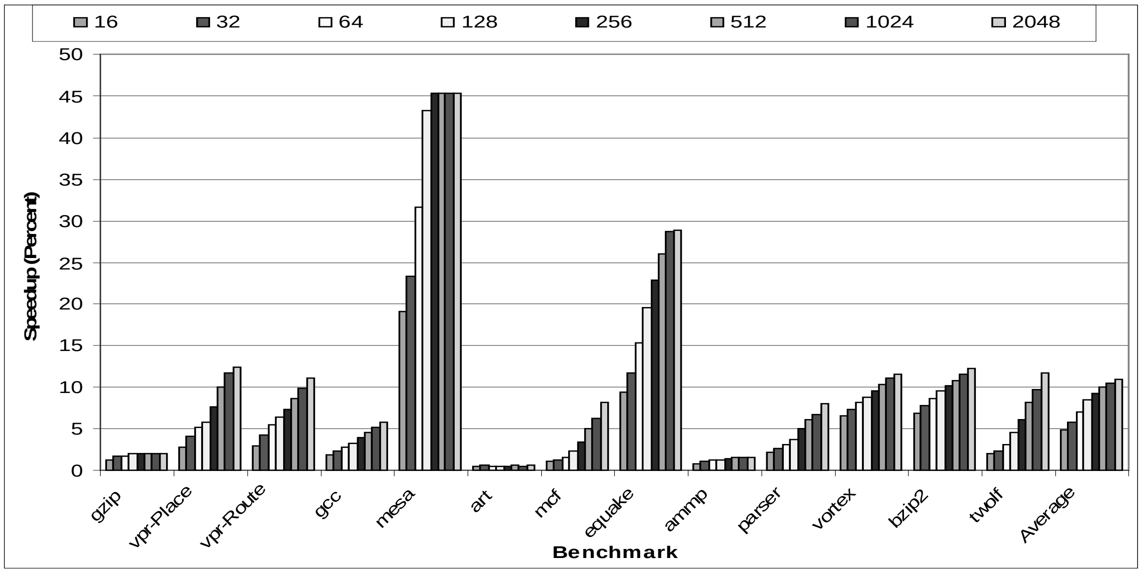

Figure A6.1.4.2: Speedup Due to Instruction Precomputation for the Highest Frequency and Latency Product Unique Computations; Profile A, Run A

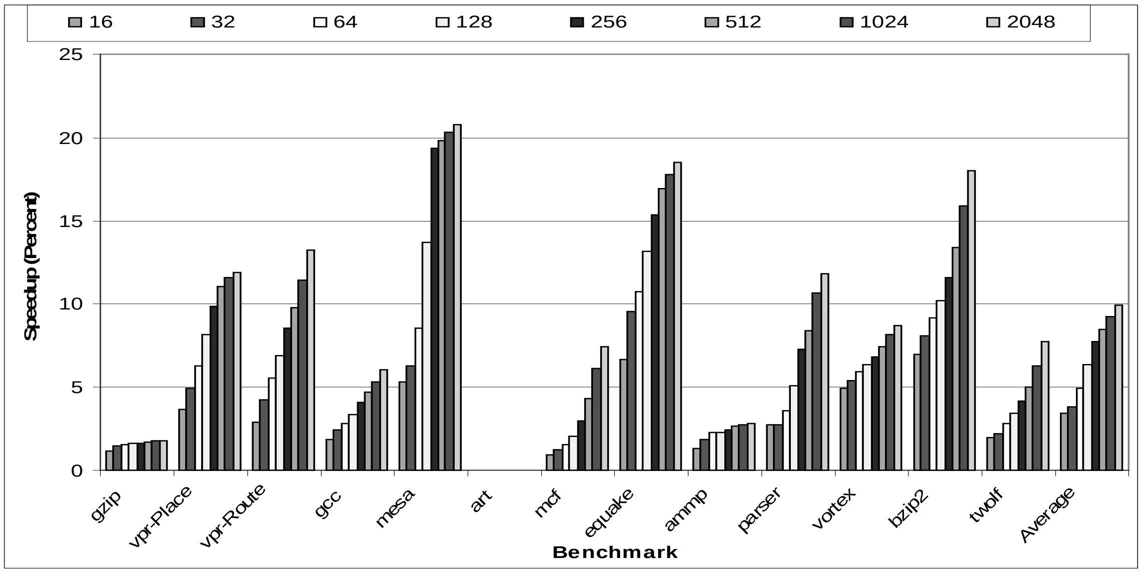

Figure A6.1.4.3: Speedup Due to Instruction Precomputation for the Highest Frequency and Latency Product Unique Computations; Profile B, Run B

(*) Results for *art* not presented due to a problem with the Netfinity system

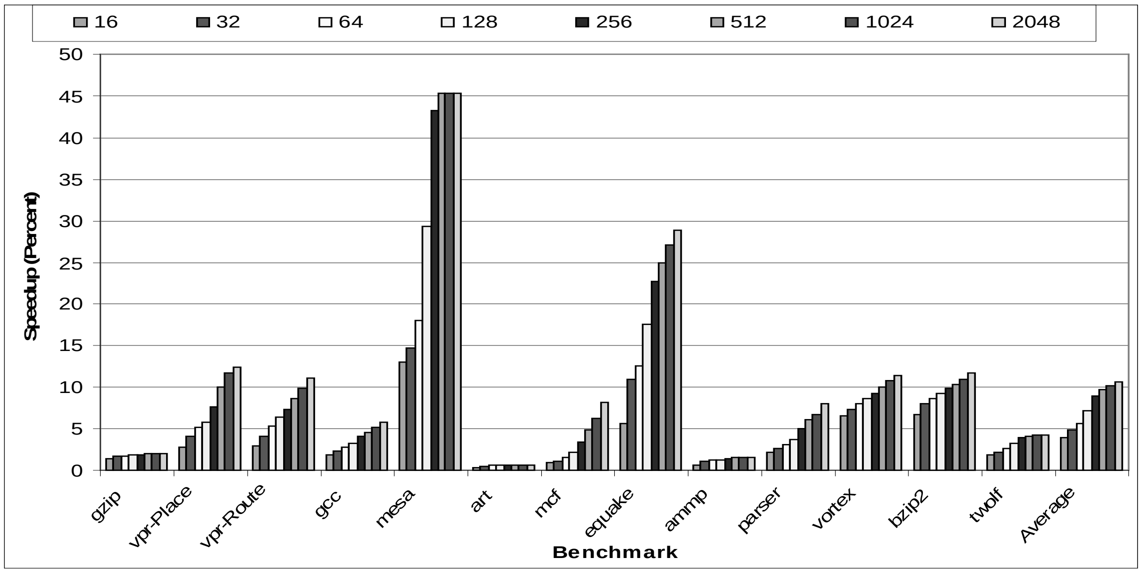

Figure A6.1.4.4: Speedup Due to Instruction Precomputation for the Highest Frequency and Latency Product Unique Computations; Profile AB, Run A

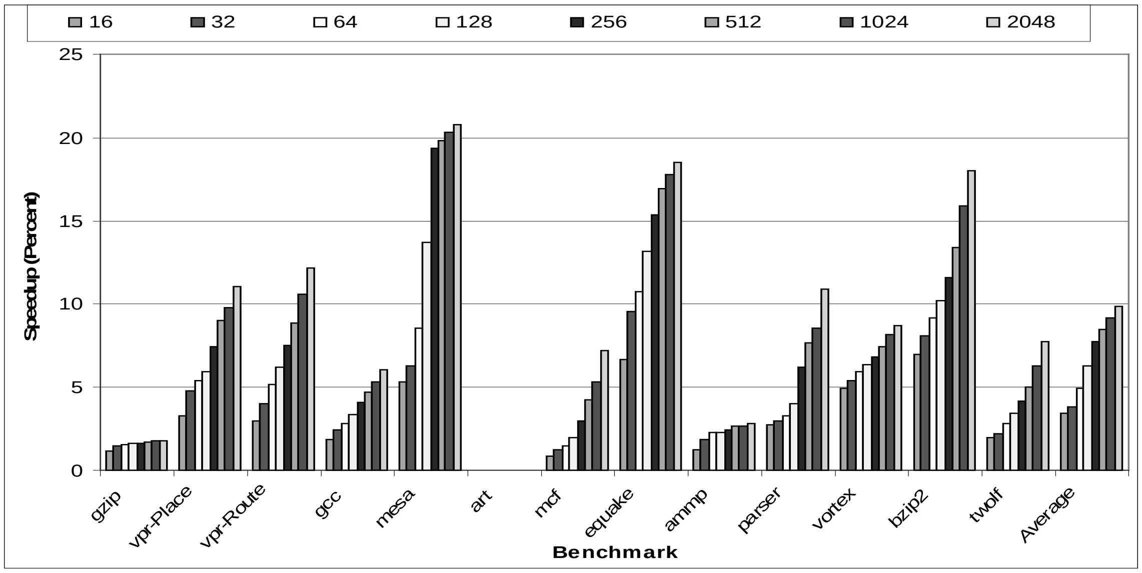

Figure A6.1.4.5: Speedup Due to Instruction Precomputation for the Highest Frequency and Latency Product Unique Computations; Profile AB, Run B

(*) Results for *art* not presented due to a problem with the Netfinity system

A6 Speedup Due to Value Reuse

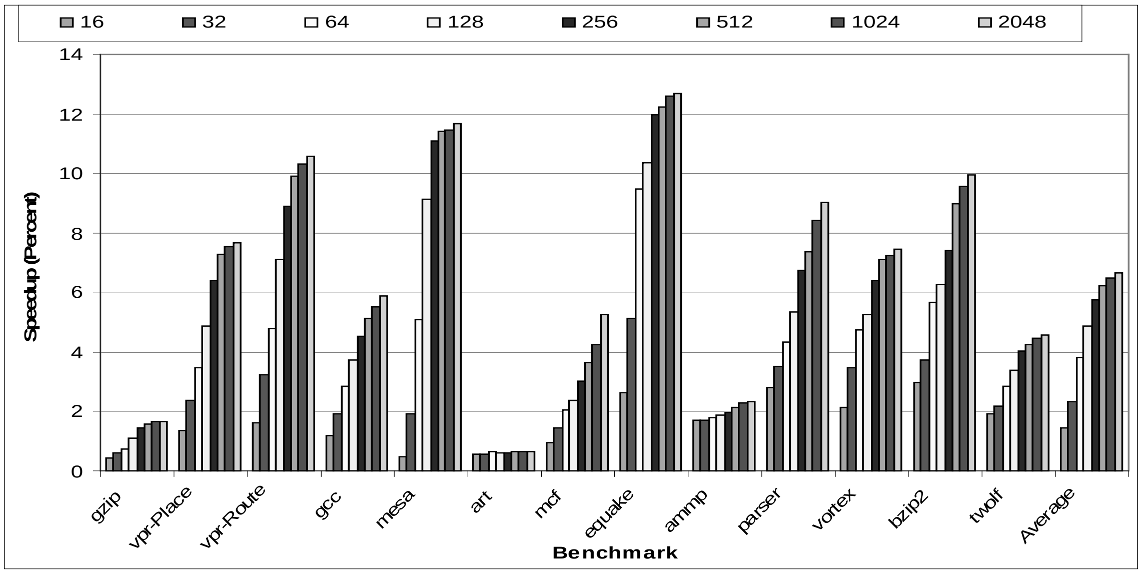

Figure A6.1.5.1: Speedup Due to Value Reuse; Run B

A7 Speedup Due to Simplifying and Eliminating Trivial Computations

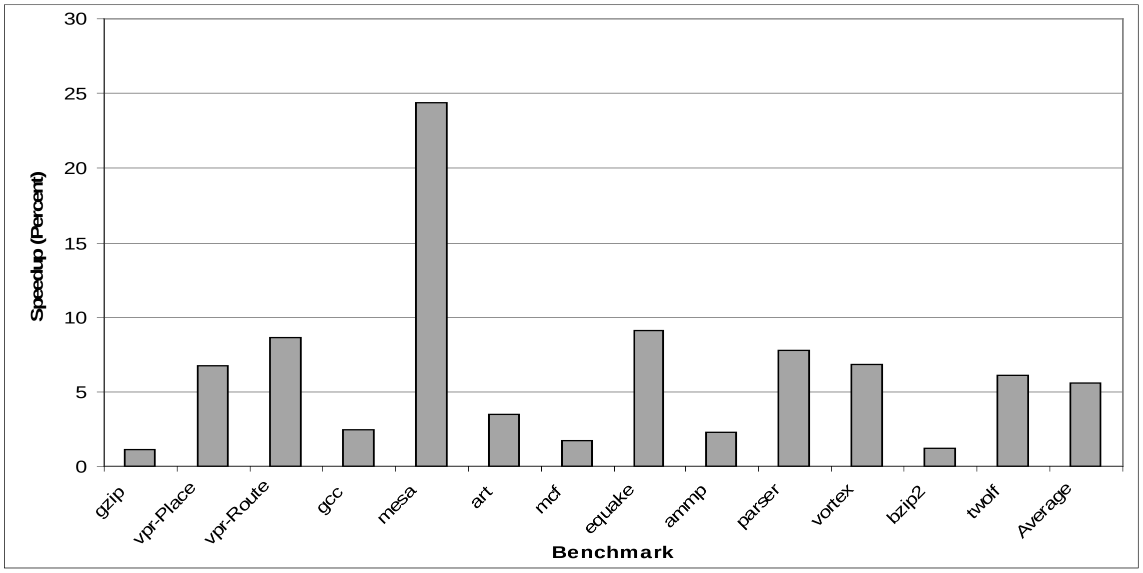

Figure A6.2.1.1: Speedup Due to the Simplification and Elimination of Trivial Computations for Selected SPEC 2000 Benchmarks, Realistic Processor Configuration, Input Set B

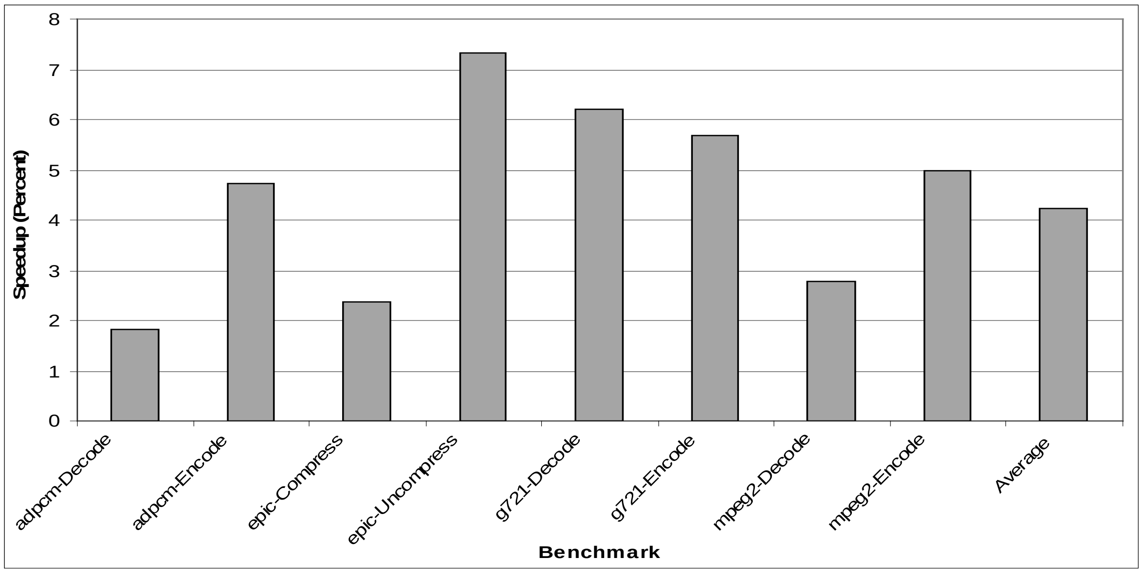

Figure A6.2.1.2: Speedup Due to the Simplification and Elimination of Trivial Computations for Selected MediaBench Benchmarks, Realistic Processor Configuration, Input Set B

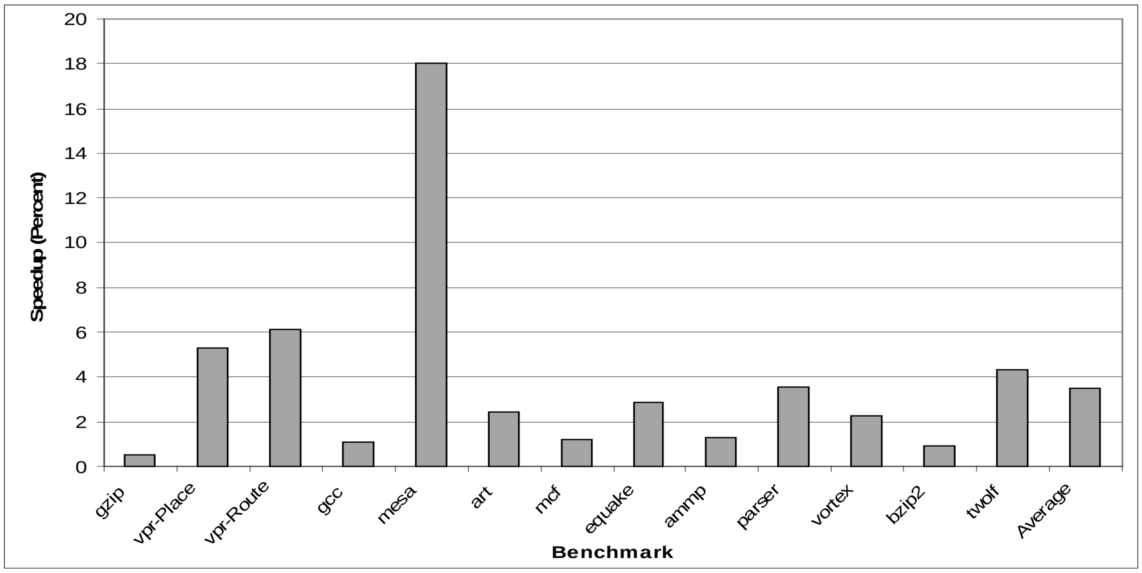

Figure A6.2.2.1: Speedup Due to the Simplification and Elimination of Trivial Computations for Selected SPEC 2000 Benchmarks, Enhanced Processor Configuration, Input Set B

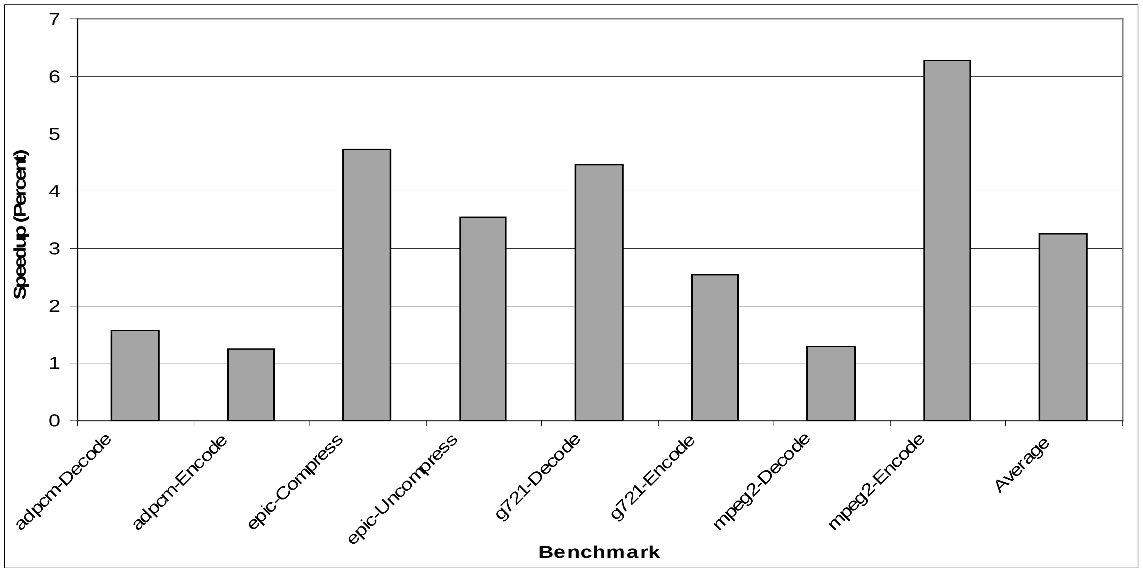

Figure A6.2.2.2: Speedup Due to the Simplification and Elimination of Trivial Computations for Selected MediaBench Benchmarks, Enhanced Processor Configuration, Input Set B

List of Publications

Book Chapters:

- Joshua J. Yi and David J. Lilja, "Instruction Precomputation", *Speculative Execution in Modern Computer Architectures*, edited by Pen-Chung Yew and David Kaeli, To be Published.
- Joshua J. Yi and David J. Lilja, "Computer Architecture", *Handbook of Innovative Computing*, edited by Albert Zomaya, Springer-Verlag, To be Published.

Conference Papers:

- Joshua J. Yi, David J. Lilja, and Douglas M. Hawkins, "A Statistically Rigorous Approach for Improving Simulation Methodology", International Conference on High-Performance Computer Architecture, February 2003.
- Joshua J. Yi and David J. Lilja, "Improving Processor Performance by Simplifying and Bypassing Trivial Computations", International Conference on Computer Design, September 2002.
- Joshua J. Yi, Resit Sendag, and David J. Lilja, "Increasing Instruction-Level Parallelism with Instruction Precomputation", Euro-Par, August 2002.

Workshop Papers:

- Joshua J. Yi and David J. Lilja, "An Analysis of the Amount of Global Level Redundant Computation in the SPEC 95 and SPEC 2000 Benchmarks", Workshop on Workload Characterization, December 2001.