MICROARCHITECTURAL TECHNIQUES TO EXPLOIT REPETITIVE COMPUTATIONS AND VALUES

Carlos Molina Clemente

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Barcelona (SPAIN)

Advisors:

Antonio Gonzalez Colás
Universitat Politècnica de Catalunya

Jordi Tubella Murgadas
Universitat Politècnica de Catalunya

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor en Informática

July 2005

"Once upon a time there was..."
Classical Fairy Tales Beginning.

PREFACE

Data dependences are some of the most important hurdles that limit the performance of current microprocessors. Some studies have shown that some applications cannot achieve more than a few tens of instructions per cycle in an ideal processor with the sole limitation of data dependences. This suggests that techniques for avoiding the serialization caused by them are important for boosting the instruction-level parallelism and will be crucial for future microprocessors.

Moreover, innovation and technological improvements in processor design have outpaced advances in memory design in the last ten years. Therefore, the increasing gap between processor and memory speeds has motivated that current high performance processors focus on cache memory organizations to tolerate growing memory latencies. Caches attempt to bridge this gap but do so at the expense of large amounts of die area, increment of the energy consumption and higher demand of memory bandwidth that can be progressively a greater limit to high performance.

We propose several microarchitectural techniques that can be applied to various parts of current microprocessor designs to improve the memory system and to boost the execution of instructions. Some techniques attempt to ease the gap between processor and memory speeds, while the others attempt to alleviate the serialization caused by data dependences. The underlying aim behind all the proposed microarchitectural techniques is to exploit the repetitive behaviour in conventional programs.

Instructions executed by real-world programs tend to be repetitious, in the sense that most of the data consumed and produced by several dynamic instructions are often the same. We refer to the repetition of any source or result value as *Value Repetition* and the repetition of source values and operation as *Computation Repetition*. In particular, the techniques proposed for improving the memory system are based on exploiting the value repetition produced by store instructions, while the techniques proposed for boosting the execution of instructions are based on exploiting the computation repetition produced by all the instructions.

ACKNOWLEDGEMENTS

Supongo que todos tenemos nuestros efectos mariposa. El que ha marcado profesionalmente mi vida, y que hace que esté escribiendo estas líneas, apareció una tarde de verano de hace bastantes años cuando cursaba mis últimos años de carrera. Mi falta de puntualidad por aquel entonces hizo que respondiera a una llamada de teléfono cuando ya salía por la puerta de casa. Buscaban a un becario de docencia para el departamento de arquitectura de computadores. Fue Jordi Tubella, uno de mis directores de tesis, quien hizo esa llamada. Mi primer agradecimiento va para él. Más tarde apareció Antonio González, mi otro director, quien me propuso hacer un Erasmus tras el cual me brindó la oportunidad de cursar el doctorado. Para él, también mi agradecimiento. De hecho, para ambos el agradecimiento es doble. No sólo por estar en momentos que han marcado que hoy esté vinculado al mundo universitario, sino por lo mucho que he aprendido de ellos en estos años de doctorado. Sin su confianza y paciencia hoy no estaría aquí.

En este largo camino, hay otros compañeros de viaje a los que debo agradecer los buenos ratos que hemos pasado y lo mucho que también me han enseñado. Me estoy refiriendo a los estudiantes de doctorado del departamento de arquitectura de computadores que como yo han intentado llegar a buen puerto. Todos ellos sin excepción, siempre han tenido tiempo para aclararme cualquier duda y para hacer más ameno este viaje. Especialmente quiero agradecérselo a Pedro, Llorenç, Suso, Pepe, Ramón y Carles que estuvieron desde el principio y con los que he pasado grandes momentos.

Finalmente, quisiera agradecer a mi familia el brindarme un entorno tan propicio para hacer tan llevadero todo este trabajo. Sobre todo a mis padres por darme la tranquilidad de estudiar sin ninguna otra preocupación. A ellos va dedicada esta tesis. También quiero agradecer a mi hermano que sea mi hermano. Y como no, a Esther. Tú eres el efecto mariposa que da sentido a mi vida.

A mis padres

Rafael y Olga

CONTENTS

“Civilization advances by extending the number of important operations which we can perform without thinking about them”
Alfred North Whitehead, English Mathematician, 1861-1947.

Chapter 1

INTRODUCTION

The serialization caused by data dependences and the gap between processor and memory speeds have become a key constraint in the design of current processors. Microarchitectural techniques aimed at breaking data dependences and improving the memory system are therefore becoming critical.

In this chapter we analyse the high percentage of repetition in real-world programs and introduce the work developed in this thesis.

1.1. Motivation

Current microprocessors designs primarily try to improve performance by increasing the clock frequency and the number of instructions that can fetch, decode, issue, execute and commit in a single cycle. Decode, execute and commit stages can manage several instructions per cycle without significant limitations, but fetch and issue stages are limited by two program characteristics: control flow and data flow [62],[125]. Control flow is defined as the sequence of execution of instructions in a program, which is determined at run time by the input data and control structures. Data flow is defined as the combination of operations and dependences which form the program and allows instructions to be executed as soon as their inputs are available, regardless of the original program order.

The program's control flow dependences, or *Control Dependences,* limit the number of instructions that can be fetched. Often, the average number of fetched instructions is below the maximum number of instructions that can be fetched in each cycle. Instruction flow techniques are still an important area of research because the throughput of the early pipeline stage determines the maximum throughput of all subsequent stages.

Issue is limited by the program's data flow dependences, or *Data Dependences,* because only data independent instructions can be issued in the same cycle. Moreover, data dependences impose a serialization on execution because processors have to stall the execution of one instruction until its source values have been produced by previous instructions. Data dependences are therefore some of the most important hurdles that limit the performance of current microprocessors.

The amount of *Instruction-Level Parallelism* (ILP) processors can exploit is significantly limited by the serialization caused by data dependences. This limitation is more severe for integer codes, where data dependences are more abundant. Some studies on the ILP limits of integer applications have shown that some of them cannot achieve more than a few tens of instructions per cycle (IPC) in an ideal processor with the sole limitation of data dependences [47],[89],[125]. This suggests that techniques for avoiding the serialization caused by data dependences are important for increasing ILP and will be crucial for future wide-issue microprocessors. In this thesis, we attempt to boost the execution of instructions to alleviate the serialization of data dependences.

In the last ten years, innovation and technological improvements in processor design have outpaced advances in memory design. For this reason, current high performance processors focus on cache memory organizations to ease the gap between processor and memory speed. The problem is that many of the techniques that are used to tolerate growing memory latencies lead to higher demand for memory bandwidth, large amounts of on-chip cache memory and increases in energy consumption. This has been shown to be a progressively greater limit to high performance [16],[17],[42],[103],[118]. In this thesis, we also attempt to improve the memory system by reducing the memory bandwidth, die area, access time and energy consumption of current data cache organizations.

To boost the execution of instructions and improve the memory system, we propose several microarchitectural techniques that can be applied to various parts of current microprocessor designs. The underlying aim behind all the proposed microarchitectural techniques is to exploit the repetitive behaviour in conventional programs.

1.2. Repetition in Conventional Programs

Instructions executed by real-world programs tend to be repetitious, in the sense that most of the data consumed and produced by several dynamic instructions are often the same. Some studies [75],[106],[111] identified several tendencies that lead to repetition in many real programs, even with aggressive compilers.

Primarily, repetition is due to the input sets of real-world programs that contain data with little variation (consider, for example, sparse matrices that contain a majority of zeros). Compiler also generates repetition when it has to address error checking, program constants, branch destination computation, virtual function calls, glue code, memory alias resolution, call-subgraph identities and register spill code. Finally, the nature of some algorithms can also introduce repetitive computation. For example, convergent algorithms result in redundant computation in the convergent areas.

1.2.1. Computation and Value Repetition

Computations performed by the instructions of a program can be summarized as follows: `F[x,y]=z`, where `F` represents the operation that is performed with `x` and `y` as source values and `z` represents the result of the computation. Repetition can be produced in many cases and involves operation, result and source values (individually or combined). In this thesis, we consider two types of repetition: (1) if the same combination of operation and source values have been produced exactly in the past, and (2) if source and/or result values have appeared earlier in the program. In the first case it is the left part of the equation (`F[x,y]`) that is repetitive and in the second case it is any of all values that are repetitive (`x,y,z`).

We call *Value Repetition* to the repetition of any source or result value and *Computation Repetition* to the repetition of source values and operation.

1.2.2. Analysis of Repetition

In this section we quantify the amount of repetition that is present in conventional programs.

The simulation environment is built on top of the Simplescalar [15] Alpha toolkit. The original simulator was modified to collect repetition statistics. To determine this repetition, source values, the opcode, and the destination value for all dynamic instructions are stored. Note that memory requirements are extremely high.

We randomly picked a subset of the SPEC2000 benchmark suite comprising both integer and FP codes: *bzip2, crafty, eon, gcc, gzip, parser, twolf, vortex* and *vpr* from the integer suite, and *ammp*, *applu, apsi, art, equake, mesa, mgrid, swim* and *wupwise* from the FP suite. These programs were compiled with the Compaq C compiler with -O5 -non_shared optimization flags (i.e, maximum optimization). Each program was run with the reference input set and statistics were collected for 500 million instructions after skipping the initializations (see Section 1.4 for further details of tools and benchmarks).

1.2.2.1. Amount of Computation Repetition

We first analyse the computations performed by the instructions. As pointed out earlier, computation repetition is produced when the combination of operation and source values has been produced exactly in the past. We consider all the dynamic instructions to analyse the computation repetition.

Figure 1.1 shows the percentage of repetitive computations. Note that the percentage of repetition is significant for all benchmarks and that, on average, more than 80% of the computations of a program have been done exactly in the past. Individual percentages range from 50% for *mgrid* to 96% for *equake*. Note also that this percentage is slightly lower for FP than for integer programs (77% and 87%, respectively).

Figure 1.2 shows the contribution of the most frequent computations to the total percentage of computations in a program. We only show the average numbers of all the simulated benchmarks. Note

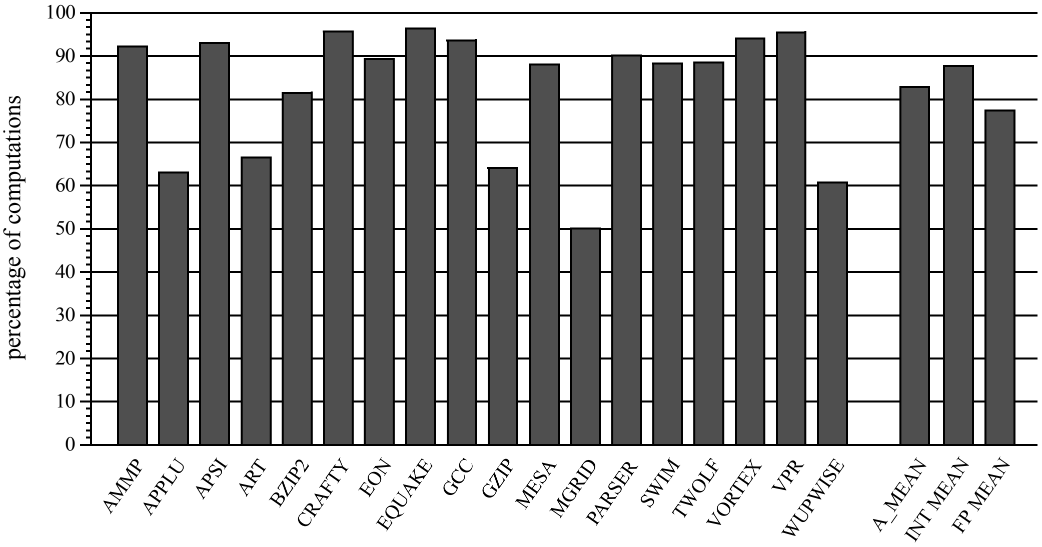


Figure 1.1. Percentage of computation repetition.

that the X-axis uses a logarithm range that moves from the most frequent computation on the left side to the 65,536 most frequent computations on the right side. The most important result of these simulations is that few computations contributes significantly to the total percentage of computations that programs perform. In other words, a program executes a small set of different computations a very large number of times. For instance, the 32 most frequent computations represent 20% of total computations, and we only need 1,024 different computations to obtain 50% of total computations. Intuitively, it seems that the combination of two source values and one operation will produce millions of possibilities (for

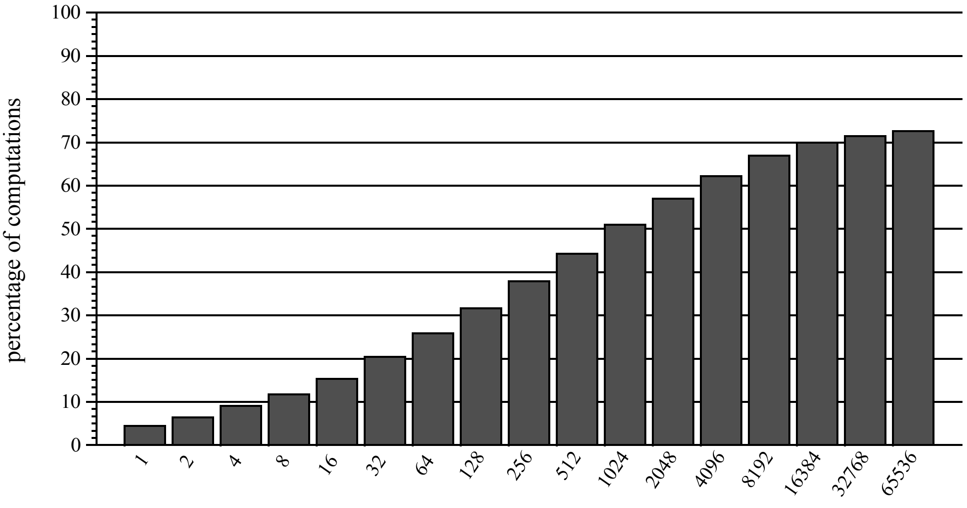


Figure 1.2. Contribution of the most frequent computations to the total computations.

instance, a 64-bit source value can contain billions of different values) but in reality with just with a few computations we can obtain most of the computations that a program performs.

1.2.2.2. Amount of Value Repetition

We now analyse the values produced by the instructions to give a flavour of the value repetition. As pointed out earlier, value repetition is produced when the values consumed/produced by one instruction have appeared earlier in the program. In this study, we only consider instructions that produce a result (in the case of store instructions, we consider the value stored in memory as a result value).

Note that computation repetition also produces value repetition of the result but value repetition does not necessarily mean computation repetition. In the first case, it is obvious that the same combination of operation and source values will always produce the same result, but in the second case the same result value can be produced by several computations. Therefore, the percentage of value repetition of the result will always be higher than the percentage of computation repetition.

Figure 1.3 shows the percentage of repetitive values. As expected, the percentage of value repetition is better than the percentage of computation repetition and is also extremely high. Over 90% of the result values have been produced by an earlier instruction. Individual percentages range from 56% for *mgrid* to 99% for *ammp, equake, gcc* and *vortex*. Note also that this percentage is slightly lower for FP than for integer programs (87% and 96%, respectively).

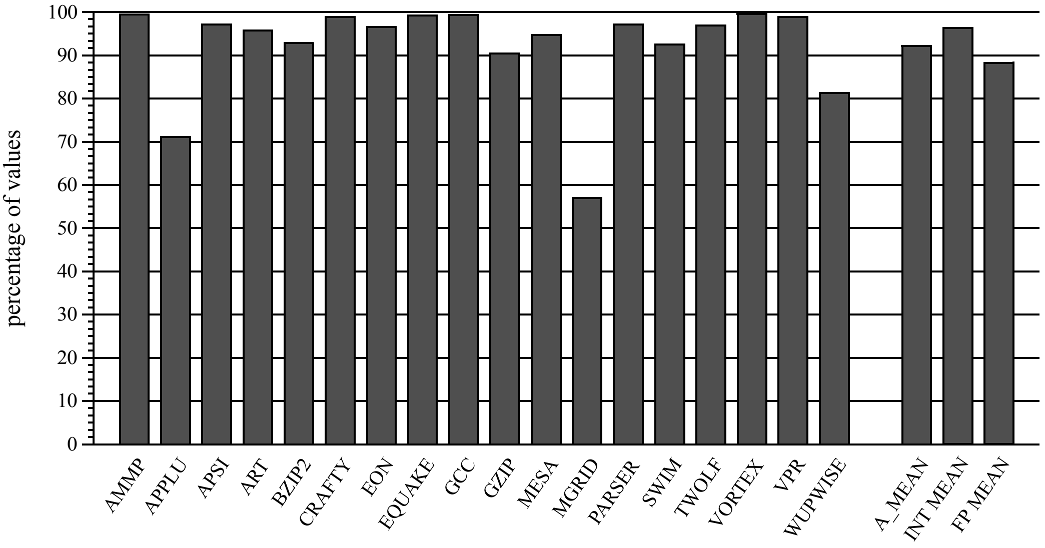


Figure 1.3. Percentage of value repetition.

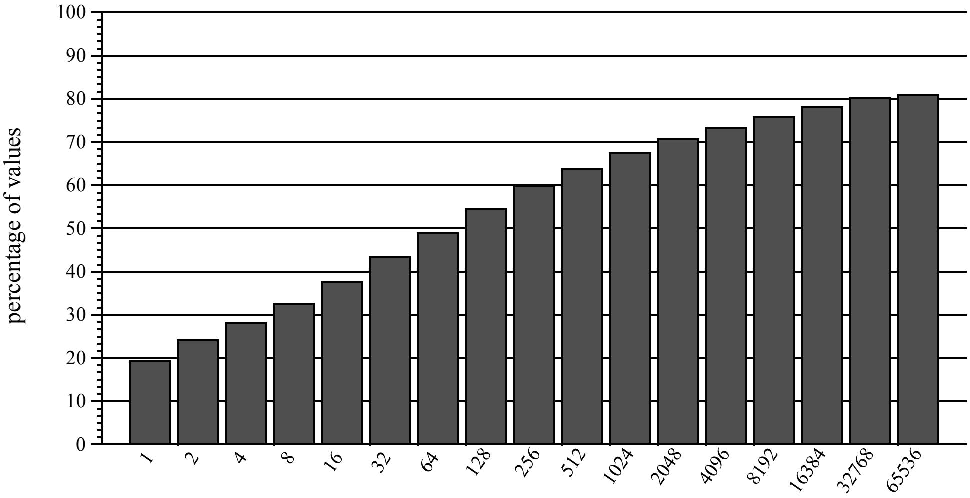


Figure 1.4. Contribution of the most frequent values to the total values.

Figure 1.4 shows the contribution of the most frequent values to the total percentage of values produced in a program. We only show the average numbers of all the simulated benchmarks. Note again that the X-axis uses a logarithm range that moves from the most frequent value on the left side to the 65,536 most frequent values on the right side. The values analysed have a 64-bit width, so billions of different values can be produced. Simulations results show that just a few values can greatly contribute to the total percentage of values produced in a program. For instance, just one value is produced by close to 20% of a program's instructions. It is easy to guess that this value is zero. Besides zero, note that we only need 1,024 different values to obtain close to 70% of the total values managed by programs.

1.3. Main Contributions

We propose several microarchitectural techniques that can be applied to various parts of current microprocessor designs. These techniques attempt (1) to improve the memory system and (2) to boost the execution of instructions by exploiting the repetitive behaviour in conventional programs. In particular, the techniques proposed for improving the memory system are based on exploiting the value repetition produced by store instructions, while the techniques proposed for boosting the execution of instructions are based on exploiting the computation repetition produced by all the instructions.

1.3.1. Contributions to Improve the Memory System

Modern processor designs manage large amounts of on-chip cache memories to deal with the gap between processor and memory speeds. This has special considerations in terms of memory traffic, die area, power dissipation and latency. To tackle these issues, we propose novel cache designs that benefit from the repetition in the memory hierarchy.

We therefore introduce the concept of *Redundant Store Instructions.* This type of stores do not modify memory since the value they write is equal to the existing value. In other words, they do not modify the state of the memory. In Chapter 2, we study the behaviour of these particular stores and show how they can be applied to a processor or multiprocessor system memory hierarchy in order to increase performance and decrease power consumption. This technique has no extra cost, in the sense that the added hardware and complexity are negligible. In particular, we show that we can achieve a significant saving on memory traffic between the first and second level cache by exploiting this feature. In this case, we exploit the value repetition of the effective address and the value that is stored in memory.

In Chapter 2, we also analyse the repetition of values into several storage locations of data caches. We conclude that data caches exhibit a high percentage of value replication at any given time. From this observation, we present a new data cache design called *Non-Redundant Cache* to reduce its die area, power dissipation and latency. Basically, the *Non-Redundant Cache* reduces the storage requirements of data cache by exploiting the significant amount of replication present in conventional cache designs. It also includes a simple compression scheme based on inlining narrow values. In this case, we simply exploit the value repetition of the value that is stored in memory.

Briefly, the main contributions are:

- A detailed study of value repetition in data caches.
- The concept and analysis of *Redundant Store Instructions*.
- A technique that exploits *Redundant Store Instructions* to reduce memory traffic between levels of memory hierarchy.
- A data cache design called *Non-Redundant Cache* that avoids the replication of values in data caches and reduces die area, power dissipation and access time.

Our work on this topic has been published in three papers. The concept of redundant store instructions and its application to reduce memory traffic was presented at the *International Conference on High Performance Computing and Networking* (HPCN 1999) [80]. The study of data cache repetition, the *Non Redundant Cache* and its analysis in terms of die area, latency and power was presented at the *International Symposium on Low Power Electronics and Design* (ISLPED 2003) [78]. Finally, a modification of the *Non-Redundant Cache* that applies extensive value compression was presented at the *International Conference on Parallel and Distributed Computing* (EUROPAR 2003) [4].

1.3.2. Contributions for Boosting the Execution of Instructions

Three techniques are applied for boosting the execution of instructions: *Instruction-Level Reuse*, *Trace-Level Reuse* and *Trace-Level Speculation*. All of these exploit the repetitive behaviour of computations present in conventional programs.

1.3.2.1. Instruction-Level Reuse

Instruction-level reuse alleviates the serialization caused by data dependences. The idea is that the work done by some instructions can be non-speculatively reused when they perform the same work again. This reduces functional units utilization and, more importantly, reduces the time needed to compute the results, thus shortening the lengths of critical paths of the execution.

In Chapter 3, we analyse the phenomenon of instruction-level reuse by evaluating its performance potential for an infinite resource machine and for a machine with a limited instruction window. The aim is to study the performance limits of instruction-level reuse while ignoring implementation aspects. We will show that instruction-level reuse can exploit a high degree of reuse and may provide very large speed-ups for an ideal machine. However, performance is degraded when the reuse latency is considered.

In this chapter, we also present a hardware implementation for dynamic instruction-level reuse in superscalar microprocessors. The underlying concept exploited by this mechanism is the run-time removal of redundant computations and particularly the elimination of *quasi-invariants* and *quasi-common subexpressions*. In the first case, a quasi-invariant is defined as a computation that is repeated many times and often produces the same result. In the second case, a quasi-common subexpression is defined as a computation that often produces the same result as another piece of code. Removing redundant computation is a target of optimizing compilers. Because of their limited knowledge of the data, however, they do not always succeed. The proposed mechanism can also remove quasi-redundant computations, such as subexpressions that often produce the same result but sometimes differ (depending on the data values) and cannot therefore be eliminated by the compiler.

Briefly, the main contributions are:

- A detailed study that evaluates the performance potential of instruction-level reuse.
- A mechanism called *Redundant Computation Buffer* that can exploit reuse due to quasi-invariants and quasi-common subexpressions while exhibiting a low reuse latency. Its novel features, such as memory management, can be applied to previous hardware implementations of instruction-level reuse.

Our proposals on this topic were published at the *International Conference on Supercomputing* (ICS 1999) [77] and in a technical report [37] evaluating the performance potential of instruction-level reuse.

1.3.2.2. Trace-Level Reuse

Here we introduce and study the concept of trace-level reuse. Trace-level reuse exploits the fact that many sequences of instructions are repeatedly executed (most of these repetitions have the same inputs and thus generate the same results) by buffering previous inputs and their corresponding outputs. When a trace is encountered again and its current inputs are found in that buffer, its execution can be avoided by obtaining the outputs from the buffer.

Exploiting reuse at trace level implies that a single reuse operation can skip the execution of a potentially large number of instructions. More importantly, as these instructions do not need to be fetched, they do not consume fetch bandwidth. Moreover, since these instructions are not placed in the reorder buffer, they do not occupy any slot of the instruction window, so the effective instruction window size is increased as a side effect. Particularly interesting is the fact that this technique may compute all at once the results of a chain of dependent instructions (e.g. in a single cycle), which allows the processor to exceed the dataflow limit that is inherent in the program.

In Chapter 4 we analyse the performance potential of trace-level reuse under several scenarios. We also compare the relative advantages of trace-level reuse and instruction-level reuse and show that trace-level reuse is more effective than instruction-level because it reduces fetch bandwidth and instruction window requirements. Trace-level reuse also has a lower overhead because a single reuse operation can avoid the execution of a long sequence of instructions. Finally, we address essential issues for integrating a trace-level reuse scheme into a superscalar processor.

Briefly, the main contributions are:

- The concept of trace-level reuse.
- A detailed analysis of the performance potential of trace-level reuse under several scenarios.
- A discussion of the design issues for integrating a trace-level reuse scheme on a superscalar processor.

Our proposals on this topic were published at the *International Conference on Parallel Processing* (ICPP 1999) [38] and in a technical report [37] evaluating the performance potential of trace-level reuse.

1.3.2.3. Trace-Level Speculation

Trace-level speculation avoids the execution of a dynamic sequence of instructions by predicting the set of live-output values based on previously seen results. This prediction exploits the observation that a limited set of unique values constitutes the majority of values produced and consumed by conventional programs. Trace-level speculation solves the reuse test of trace-level reuse, that it is not easy to handle, but it introduces penalties due to a misspeculation.

There are two important issues in trace-level speculation. The first of these involves the microarchitecture support for trace speculation and how the microarchitecture manages trace speculation. The second involves trace selection and data value speculation techniques.

In Chapter 5 we present a novel microarchitecture for exploiting trace-level speculation using two threads working cooperatively. One thread, called the speculative thread, executes instructions ahead of the other by speculating on the result of several traces. The other thread executes speculated traces and verifies the speculation made by the first thread. This architecture has two main advantages: (a) no significant penalties are introduced in the presence of a misspeculation and (b) any type of trace predictor can work with this proposal.

We also propose a static program analysis for identifying candidate traces to be speculated. This approach identifies large regions of code whose produced values may be successfully predicted. We present several heuristics to determine the best opportunities for dynamic speculation based on compiler analysis and program profiling information.

Briefly, the main contributions are:

- A microarchitecture to exploit trace-level speculation called *Trace-Level Speculative Multithreaded Architecture*. This architecture does not introduce significant trace misprediction penalties and does not impose any constraint on the approach to building or predicting traces.
- A trace selection method to identify large regions of code that may be successfully predicted based on a static analysis that uses profiling data.

Three papers have been published on this topic: the definition of the architecture was presented at the *International Conference on Computer Design* (ICCD 2002) [81]; a hardware improvement that reduces misspeculation penalties was presented at the *6th International Symposium on High Performance Computing* (ISHPC 2005) [79]; and compiler analysis to support *Trace-Level Speculative Multithreaded Architectures* was presented at the *9th Annual Workshop on Interaction between Compilers and Computer Architectures* (INTERACT 2005) [76].

1.4. Methodology and Experimental Framework

The microarchitectural techniques introduced in this thesis have been analysed, proposed and evaluated using a similar methodology. For each technique, we first analyse a set of benchmarks to quantify a special repetitive behaviour of the instructions. Basically, this analysis is performed with the help of tools that easily provide information about the instructions executed in a program. Then, we analyse the results to define proposals that are targeted to improve the memory system or boost the execution of instructions. Once the proposals are defined, we evaluate them. To do so, we modify a set of simulators widely used in the computer architecture research community. These simulators can provide information about computer performance in terms of speed-up or cache hit/miss ratio. Finally, we also use a well-known analytical model, that greatly facilitates the work of circuit simulation, for the evaluation of area, power and cycle time of data caches.

In the following subsections we provide an overview of tools, benchmarks and baseline microarchitecture that are used to propose and evaluate the microarchitectural techniques introduced in this thesis. We detail the specific experimental framework of the techniques in their corresponding chapters.

1.4.1. Tools

ATOM [117]: *Analysis Tools with Object Modification* (ATOM) is a single framework for building a wide range of high performance program analysis tools. It provides instrumentation routines that allow the user to have access to each procedure in an application, each basic block in that procedure, and each instruction in that basic block. In addition to the instrumentation routines, the user can also write analysis routines. The main advantage of this tool is that provides a very flexible and efficient code instrumentation interface that helps to build customized simulators with very little effort. The main disadvantage is that ATOM is a trace-driven simulator. This means that the analysis reads existing executed traces of instructions to simulate models. Thus, ATOM can not model accurately speculation or misprediction. In this thesis, we use the ATOM tool to analyse and quantify a special repetitive behaviour of programs.

SimpleScalar Tool Set [15]: Besides trace-driven simulation, execution-driven simulation is a useful technique for modelling high-performance of modern microprocessors. This technique, which is the most accurate and most costly of the simulation techniques, requires instruction and I/O emulators to reproduce program computation. This means that it decodes and executes machine instructions on-the-fly. The *SimpleScalar Tool Set* provide simulators ranging from a fast functional simulator to a detailed out-of-order issue processor that supports non-blocking caches, speculative execution, and state-of-the-art branch prediction. In this thesis, we use the simulators of the Simplescalar tool set in several ways. The *sim-fast* simulator is modified to provide results of the repetitive behaviour of programs; the *sim-outorder* simulator is modified to evaluate some techniques in an out-of-order superscalar processor with a two level memory system and speculative execution support; and finally, the *sim-cache* simulator is modified to evaluate the memory hierarchy techniques when the cache performance on execution time is not considered.

CACTI [107]: This tool greatly facilitates the work of circuit simulation. It was originally developed to quantify the access time trade-offs of on-chip cache organizations. Later versions of the CACTI tool provide features to estimate area, access and cycle time and power dissipation of different direct-mapped and set-associative cache configurations. Moreover, this tool is also used to determine the cache configuration that best satisfies a desired optimization criterion. In this thesis, we use the CACTI tool as an analytical model for the evaluation of area, power and cycle time of on-chip direct-mapped and set-associative caches.

1.4.2. SPEC Benchmarks

The Standard Performance Evaluation Corporation (SPEC) [54] is an organization founded in 1988 dedicated to producing benchmarks that are reasonably scientific, unbiased, meaningful and relevant. This organization provides several families of benchmarks to measure the performance of different computer systems (CPU family), parallel and distributed computer architectures (HPC family), java application servers (JAPPSERVER family), mail servers (MAIL family), network file servers (SFS family) and web servers (WEB family). .

In this thesis we consider the CPU family, designed to provide performance measurements that can be used to compare compute-intensive workloads on different computer systems. In particular, we consider the following suites of the CPU family: SPEC CPU95 (released in August 1995) and Spec CPU2000 (released in January 2000). Each suite defines a standard reference machine and a set of programs to run, and formulas for computing the scores. Traditionally, the SPEC benchmarks define three standard runs called reference, test and train. Moreover, each suite contains two benchmark components: CINT for measuring and comparing compute-intensive integer performance, and CFP for

SPEC CPU 95		SPEC CPU 2000	
CINT95	**CFP95**	**CINT2000**	**CFP2000**
099.go	101.tomcatv	164.gzip	168.wupwise
124.m88ksim	102.swim	175.vpr	171.swim
126.gcc	103.su2cor	176.gcc	172.mgrid
129.compress	104.hydro2d	181.mcf	173.applu
130.li	107.mgrid	186.crafty	177.mesa
132.ijpeg	110.applu	197.parser	178.galgel
134.perl	125.turb3d	252.eon	179.art
147.vortex	141.apsi	253.perlbmk	183.equake
	145.fpppp	254.gap	187.facerec
	146.wave5	255.vortex	188.ammp
		256.bzip2	189.lucas
		300.twolf	191.fma3d
			200.sixtrack
			301.apsi

Table 1.1. SPEC CPU95 and CPU2000 benchmarks.

measuring and comparing compute-intensive floating point performance. Table 1.1 shows the SPEC CPU95 and SPEC CPU2000 benchmarks Note that several SPEC CPU95 programs have been included with minor changes in SPEC CPU2000 (i.e. compress, gcc, perl and vortex form the integer component and applu, apsi, mgrid and swim from the floating point component).

We began to use the SPEC CPU95 test platform but then shifted to the SPEC CPU2000. Note that the organization of this thesis does not preserve the temporal order in which we proposed the techniques. Therefore, SPEC CPU95 and SPEC CPU2000 simulations are merged throughout the chapters.

Although SPEC CPU95 still provide a meaningful point of comparison (half of SPEC CPU95 are included in SPEC CPU2000), we believe that it is important to consider the changes in technology. SPEC CPU2000 are designed to measure compute intensive performance that reflects the current advances in microprocessor technologies. However, Yi and Lilja [135] analysed repetitive behaviour in both SPEC CPU95 and SPEC CPU2000 benchmarks and did not find significant differences.

We have compiled the programs with the DEC C and Fortran compilers with full optimizations ("-*non_shared -O5 -tune ev5 -migrate -ifo*" for C codes and "*-non_shared -O5 -tune ev5*" for Fortran codes). In our simulations we consider the reference and test inputs of SPEC benchmarks. Furthermore, we forward a significant number of instructions to avoid the initialization behaviour of programs. We have empirically checked that skipping 500 million instructions with the reference input and 125 million instructions with the test input represent a good design point.

1.4.3. Baseline Microarchitecture

We use the *sim-outorder* simulator of the *SimpleScalar* tool set to determine the speed-up of a particular technique in an out-of-order superscalar processor with a two level memory system and speculative execution support. The baseline microarchitecture that we assume in this simulator is a 4-way dynamically scheduled superscalar processor based on the Register Update Unit [114]. The main characteristics of this architecture are shown in Table 1.2.

Instruction fetch	4 instructions per cycle.
Branch predictor	2048-entry bimodal predictor
Instruction issue/ commit	Out-of-order issue, 4 instructions committed per cycle, 64-entry reorder buffer, loads execute only after all the preceding store addresses are known, store-load forwarding
Architectural registers	32 integer and 32 FP
Functional units	4 integer ALUs, 4 load/store units, 4 FP adders, 2 integer mult/div, 2 FP mult/div
FU latency/repeat rate	int ALU 1/1, load/store 1/1, int mult 3/1, int div 20/19, FP adder 2/1, FP mult 4/1, FP div 12/12
Instruction cache	16 KB, direct-mapped, 32-byte block, 6-cycle miss latency
Data cache	16 KB, 2-way set-associative, 32-byte block, 6-cycle miss latency
Second Level Cache	Shared instruction & data cache, 256 KB, 4-way set-associative, 32-byte block, 100-cycle miss latency

Table 1.2. Parameters of the baseline microarchitecture.

“Do not keep in your head what you can fit in your pocket”
Albert Einstein, US (German-born) physicist, 1879-1955.

Chapter 2

VALUE REPETITION IN DATA CACHES

On-chip cache memories are getting bigger and bigger in order to ease the ever-increasing gap between processor speed and memory access. Die area, latency and power dissipation have therefore become a key constraint in the design of current cache organizations. Memory bandwidth is also a scarce resource in high-performance systems because modern processor techniques designed to deal with memory latencies lead to greater bandwidth demands.

In this chapter we propose novel data cache designs that reduce memory traffic and produce significant die area savings, power reduction and latency decrease. These novel schemes are based on the observation that most values managed by the memory hierarchy are frequently repetitive.

2.1. Introduction

In the last decade, innovation and technological improvements in processor design have outpaced those in memory design. That is why current high performance processors focus on cache memory organizations [128] to ease the gap between processor and memory speed. Several studies [42],[109] have shown the importance of a good design to maximise the hit ratio, minimise the access time to cache, minimise the delay due to a miss and minimise the overheads of updating the next memory level.

One problem is that many of these techniques for tolerating growing memory latencies do so at the expense of increased memory bandwidth, and this has been shown to be a progressively greater limit to high performance [16],[17],[42]. Techniques such as lookup free caches [115], software and hardware prefetching [21], stream buffers [58], speculative load execution [39], and multithreading [122] reduce latency related stalls but also increase the total traffic between the main memory and the processor. Memory bandwidth has therefore also become a key constraint in the design of current cache organizations.

Processor and memory integration can build cheaper and less complex competitive systems [103]. On-chip cache memories of microprocessor are getting larger because the scale of integration continues to grow and because applications are using larger working sets. Chip multiprocessor designs are also a promising way of increasing throughput. However on-chip memories and processing cores both compete for the die area, and the area occupied by one affects the amount left for the other [52]. Table 2.1 summarizes cache occupancy for several commercial processors [53]. For instance, in a Power4 processor, the die contains 2 processors and a unified 1.5MB second level cache that occupies close to 50% of the total die area. Even small processors such as Mips-R20k devote a significant area to implementing caches. On average, caches use close to 50% of the total die area. Moreover, some

	L1 Dcache	L1 Icache	L2 Cache	Total Area
Pentium 4	2%	3%	20%	25%
MipsR20k	23%	26%	none	54%
Crusoe:5400	10%	9%	27%	46%
Power4	2%	1%	50%	53%
Alpha 21364	4%	3%	36%	43%

Table 2.1. Die are occupied by caches in some commercial processors.

authors [14],[43] have reported that caches may be responsible for 10% to 20% of the total power dissipated by a processor.

In processor design several trade-offs are needed to obtain a good balance between cost and performance. For high productions volumes, cost can be associated with area chip, so a way to reduce the cost is to reduce area requirements. On the other hand, power dissipation is becoming a critical issue for microprocessors. Power dissipation determines the cost of the cooling system and ultimately may limit the performance of the microprocessor. Dynamic power dissipation of on-chip memories is strongly related to its area, whereas static power dissipation depends on the number of transistors.

In the previous chapter we demonstrated the high percentage of repetition produced by instructions in conventional programs. Therefore, store instructions manage values that are frequently repetitive. Those repetitive values may be stored in the same storage location or in several storage locations of the memory hierarchy. The first case means that the effective address and the value stored in memory are repetitive. The second case means that just the value stored in memory is repetititive. We will show that both repetitive behaviours can be exploited to improve several aspects of conventional data caches.

In this chapter, value repetition in the same storage location is exploited by a novel technique for reducing memory traffic with no special hardware requirement. In particular, we show that a significant saving on memory traffic between the first and the second level cache can be avoided with no extra cost, since the added hardware and complexity are negligible. This technique is based on the observation that some memory writes do not modify memory because the value they write is equal to the existing value. In other words, they do not modify the state of the memory. We refer to these stores which exploit this value repetition as *Redundant Stores*. We study how they behave and show how they can be applied to a processor or multiprocessor system memory hierarchy in order to increase performance and decrease power consumption.

We also present in this chapter a novel cache design, the *Non-Redundant Cache*, which reduces the storage requirements of data cache by exploiting the significant amount of replication in conventional data cache designs. These storage savings lead to significant die area savings, power reduction and latency decrease. The underlying concept behind this approach is the exploitation of value repetition in several storage locations of data caches.

The rest of this chapter is organized as follows. Section 2.2 analyses the value repetition in the same storage location of data caches and introduces the concept of *Redundant Store Instructions.*

Section 2.3 proposes a technique based on *Redundant Store Instructions* that reduces memory traffic between levels of the memory hierarchy. Section 2.4 evaluates the value repetition in several storage locations of conventional data caches. Section 2.5 introduces a novel cache design called *Non-Redundant Cache* which avoids the replication of values and reduces die area, power dissipation and access time. Finally, Section 2.6 reviews related work and Section 2.7 summarizes our main conclusions.

2.2. Value Repetition in the Same Storage Location

2.2.1. Overview

Lipasti *et al* [72] coined the term *value locality* to describe the likelihood to references of a previously seen value within a storage location. These authors reported that the most recent value produced by an instruction is frequently also the next. Based on this observation, they initially proposed last value prediction as a mechanism for predicting load values [72] and, in a subsequent study, for predicting all values produced by instructions that write into a register [71]. Unfortunately, store instructions were not analysed, since the target was to boost the execution of instructions.

We observed that several memory writes do not modify memory since the value they write is equal to the existing value. In other words, they do not modify the state of memory. These stores are what we call *Redundant Stores.* Figure 2.1 summarizes the underlying concept behind *Redundant Stores* and the next section quantifies them by analysing a set of benchmarks. Note again that a *Redundant Store* is produced when the stored value is the same as the previous one.

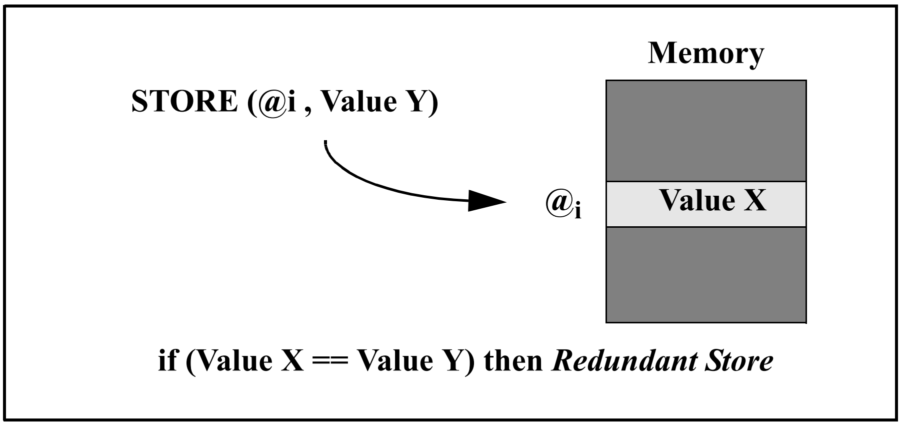


Figure 2.1. Redundant store concept.

Since our initial definition [80], several authors [10], [60], [65], [66], [67] have evaluated this particular behaviour for several stores. All these authors called them *Silent Stores* but the underlying concept is the same as in our definition of R*edundant Stores*.

Redundant Stores can be applied to any part of the processor which has a buffer that is modified such as cache memories, write buffers, load/store queues for memory disambiguation, buffers of instruction reuse or prediction schemes, etc. We will show the behaviour of redundant stores in the memory hierarchy. Rather than boosting the execution of instructions, the proposal reduces the memory bandwidth demand. By exploiting the redundancy of some stores we can achieve a significant saving on memory traffic between the different levels of the hierarchy, thus increasing the performance. We also achieve a reduction in power consumption because the memory bus is used less.

In the following subsections, we first revise different cache write policies and then quantify the amount of *Redundant Stores* in several data cache configurations.

2.2.2. Cache Write Policies

Cache memories are largely known as small amount storage of high speed access designed to supply the processor with the most frequently requested instructions and data. When data is read from, or written to, main memory, a copy is also saved in the cache, along with the associated main memory address. The cache monitors addresses of subsequent reads to see if the required data is already in the cache. If the data is cached (i.e. a cache hit), it is returned immediately and the main memory read is not started. If the data is not cached (i.e. a cache miss), it is fetched from the main memory and saved in the cache.

Basically, there are two options when writing to cache [47],[57]:

- *Write Through,* where the information is written both to the block in the cache and to the block in the lower level memory.
- *Copy Back,* where the information is written only to the block in the cache and the modified cache block is written to the next memory level only when it is replaced. This technique uses a dirty bit that indicates whether the block was written, thus reducing the frequency of writing back blocks on replacement.

Similarly, there are two options on a write miss[47],[57]:

- *Write Allocate,* where the new block is loaded into the cache.
- *No Write Allocate,* where the new block is *not* loaded in the cache and is modified in the next level of the memory hierarchy.

The combination of the above options leads to four main configurations in cache: *Write Through with Write Allocate (WT-WA), Write Through with No Write Allocate (WT-NWA), Copy Back with Write Allocate (CB-WA),* and finally *Copy Back with No Write Allocate (CB-NWA)*. Normally Copy Back with Write Allocate, and Write Through with No Write Allocate are used because, in the first case, subsequent writes to the same block can be captured by the cache and, in the second case, subsequent writes to the same block still have to go to memory.

2.2.3. Performance Evaluation

This section evaluates the amount of redundant store instructions in several data cache configurations.

2.2.3.1. Experimental Framework

We have developed a functional parameterized simulator for cache memories. Briefly, this simulator provides information about hit/miss ratios and quantifies the number of writes that does not modify the state of memory.

For the evaluation we have considered a subset of the Spec95 benchmark suite. The programs have been compiled with the DEC Fortran and C compilers for a DEC AlphaStation 600 5/266 with full optimizations, and instrumented by means of the Atom tool [117]. Each program was run with the reference input sets, and statistics were collected for 1 billion instructions after skipping the initial part, which corresponds to initializations (see Section 1.4 for further details of tools and benchmarks).

We simulated several cache configurations and cache write policies. We considered four cache sizes: 32 KB, 16 KB, 8 KB and 4 KB, and 32-byte line size. We simulated the following cache policies: *Copy Back with Write Allocate (*CB-WA for short*)* and *Write Through with No Write Allocate* (WT-NWA for short). Direct mapped caches were assumed for all simulations. Table 2.2 shows the miss ratio of every combination of the above described sizes and policies.

	COPY BACK WRITE ALLOCATE				WRITE THROUGH NO WRITE ALLOCATE			
	32 KB	*16 KB*	*8 KB*	*4 KB*	*32 KB*	*16 KB*	*8 KB*	*4 KB*
Applu	10.81 8.31 16.5	11.29 8.88 16.8	12.15 9.86 17.4	14.01 12.0 18.5	16.40 8.09 35.4	16.84 8.51 35.9	17.65 9.27 36.8	19.48 11.0 38.7
Apsi	12.61 14.0 9.95	13.12 14.5 10.4	15.03 16.6 11.9	29.59 28.3 31.9	14.99 13.9 16.9	15.71 14.3 18.2	18.95 16.9 22.8	35.26 26.4 51.9
Gcc	2.40 2.85 1.01	5.13 5.89 3.64	7.81 8.96 5.53	11.61 13.5 7.88	2.85 2.98 2.41	6.57 7.01 5.08	11.58 12.3 8.83	18.90 20.4 13.5
Go	3.52 3.96 2.65	5.73 6.74 2.28	10.26 11.9 4.54	17.13 19.8 7.72	7.71 4.23 14.6	9.67 6.26 16.4	12.90 9.59 19.4	17.87 14.5 24.4
Ijpeg	1.58 2.10 0.06	2.69 3.46 0.43	7.32 9.64 0.54	15.34 20.3 0.67	1.63 2.11 0.22	3.16 3.57 1.96	7.87 9.77 2.32	15.98 20.5 2.72
M88ksim	0.67 1.05 0.01	1.30 1.95 0.18	5.52 6.85 3.21	7.68 9.30 4.86	1.22 1.05 1.51	1.82 1.05 1.60	7.11 7.32 6.74	11.49 10.7 12.8
Mgrid	4.97 4.51 13.8	5.21 4.76 13.8	5.35 4.90 13.9	8.46 8.17 14.0	6.89 4.55 51.4	7.12 4.80 51.4	7.25 4.93 51.5	10.35 8.19 51.5
Perl	0.27 0.22 0.37	0.55 0.43 0.75	1.64 1.93 1.13	7.35 8.25 5.78	1.80 0.43 4.19	3.37 0.86 7.75	4.91 2.58 8.99	12.86 10.7 16.6
Tomcatv	12.06 14.5 5.64	15.98 18.8 8.44	29.13 31.9 21.8	36.26 39.8 27.0	16.07 14.2 20.9	18.76 17.7 21.4	30.87 31.9 28.1	38.87 40.3 35.0
Turb3d	5.82 5.55 6.18	7.59 7.40 7.84	8.92 8.73 9.17	10.60 10.6 10.5	5.79 5.73 5.87	7.93 7.59 8.37	10.62 9.17 12.5	13.52 11.3 16.4
Vortex	1.88 2.57 0.87	3.45 4.51 1.87	5.40 7.09 2.88	8.04 10.3 4.58	2.96 2.79 3.22	5.83 5.31 6.59	9.09 8.04 10.6	14.22 11.6 18.0
Wave	15.33 13.3 18.8	25.85 23.2 30.5	36.02 32.0 43.3	41.97 40.0 45.5	21.36 11.1 40.0	31.93 20.5 52.7	40.90 31.5 57.8	47.93 40.0 62.3
A_MEAN	5.99 6.09 6.33	8.16 8.39 8.09	12.05 12.5 11.2	17.34 18.3 14.9	8.31 5.93 16.4	10.73 8.20 18.9	14.98 12.7 22.2	21.39 18.8 28.6

Table 2.2: Miss ratios of CB-WA and WT-NWA for different cache sizes

Cell Description

Total Miss Ratio

Load_Miss_Ratio **Store_Miss_Ratio**

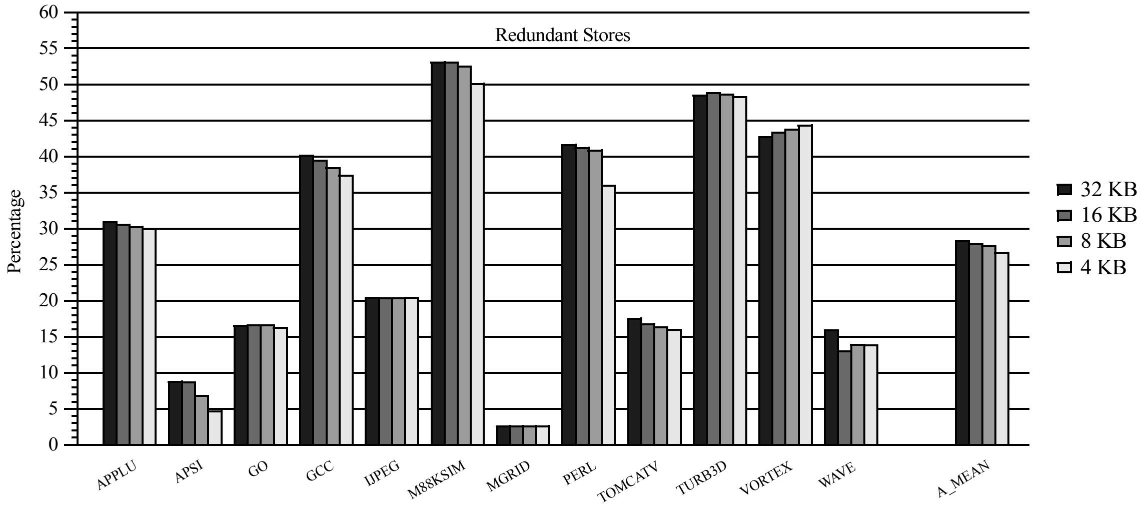


Figure 2.2. Percentage of redundant stores for CB-WA with different cache sizes.

2.2.3.2. Analysis of Results

Figure 2.2 and Figure 2.3 show the percentage of *Redundant Stores* over all the stores that access the simulated caches. Specifically, Figure 2.2 shows the percentage of *Redundant Stores* for CB-WA caches and Figure 2.3 shows the percentage for WT-NWA caches.

Note that the amount of *Redundant Stores* in cache was significant in all benchmarks. Only *mgrid* did not present a significant number of *Redundant Stores*. On average, around 30% of the stores were redundant for a CB-WA cache and 15% were redundant for a WT-NWA cache. This difference is due to

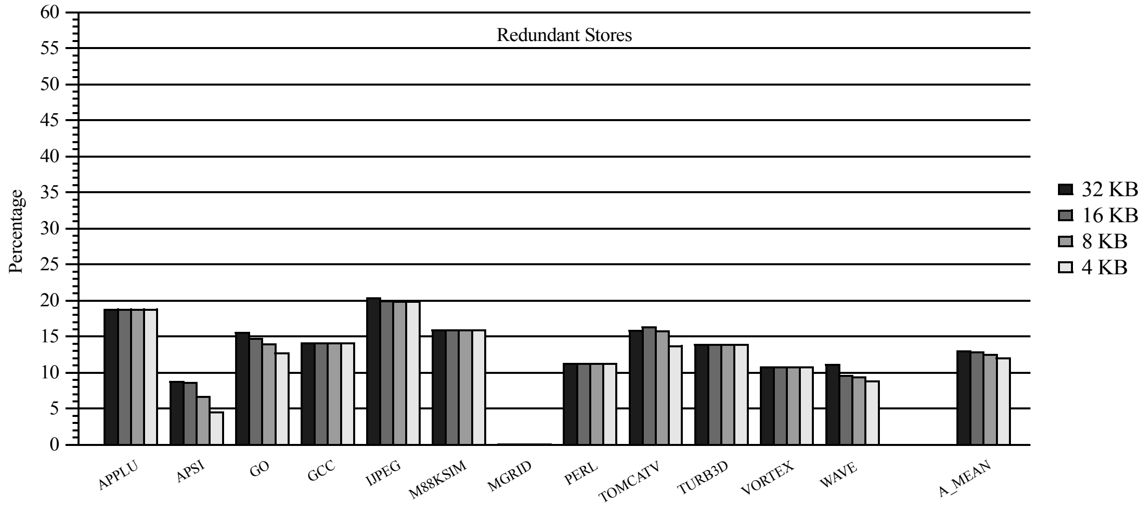


Figure 2.3. Percentage of redundant stores for WT-NWA with different cache sizes.

the fact that CB-WA has a lower store miss ratio than WT-NWA (on average, about half), so storage locations refer longer to the same memory address. The results also show that the size of the caches did not modify the percentage of *Redundant Stores*.

Using this observation, in the next section we propose a novel technique that can significantly reduce the memory bandwidth required between levels of the memory hierarchy.

2.3. Redundant Store Mechanism for Reducing Memory Traffic

In this section, we propose to benefit from redundant store instructions by slightly modifying conventional data cache designs in order to achieve significant traffic reduction between levels of the memory hierarchy.

2.3.1. General Description

We first describe the changes required in different cache configurations in order to take advantage of the *Redundant Stores* for memory traffic reduction. Note that the additional hardware required is minimal and is highlighted in each scheme.

- **Cache memory with copy-back policy:** In the regular version, a store accesses cache to write its value and sets the dirty bit. We propose first checking whether the current value in cache matches the value that this store is going to write. If so, it is a *Redundant Store* and the dirty bit is not set.

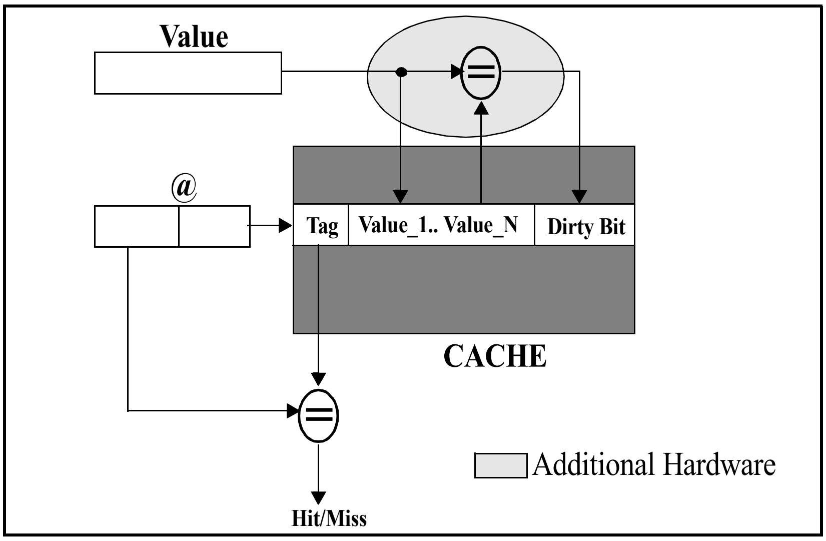


Figure 2.4. Redundant store mechanism with copy back

A store in a conventional cache first reads the tag and then writes the new value. Our mechanism simultaneously reads the old value and the tag and then writes the new value if necessary, so the cache latency is not increased. For the sake of simplicity, direct mapped caches are assumed. Note that associative caches will only need to replicate the additional hardware for each way. Figure 2.4 shows how the mechanism works for copy back caches. Here we have a reduction in the frequency at which the dirty bit is set. This reduces memory traffic because fewer blocks of the cache will have to go to the next level of the memory hierarchy when there is a cache miss. Note that, to exploit the benefit of *Redundant Stores*, we have to read the old value from the cache and compare it with the value that is going to be written. We just need a simple comparator that decides whether the dirty bit is set.

- **Cache memory with write-through policy:** In this case, *Redundant Stores* do not need to update either the cache or the next memory level. As explained earlier, it is identified by reading the old value and comparing it with the current one (see Figure 2.5). Note that the output of the comparator decides whether the buffer can send the value to the next level of the hierarchy.

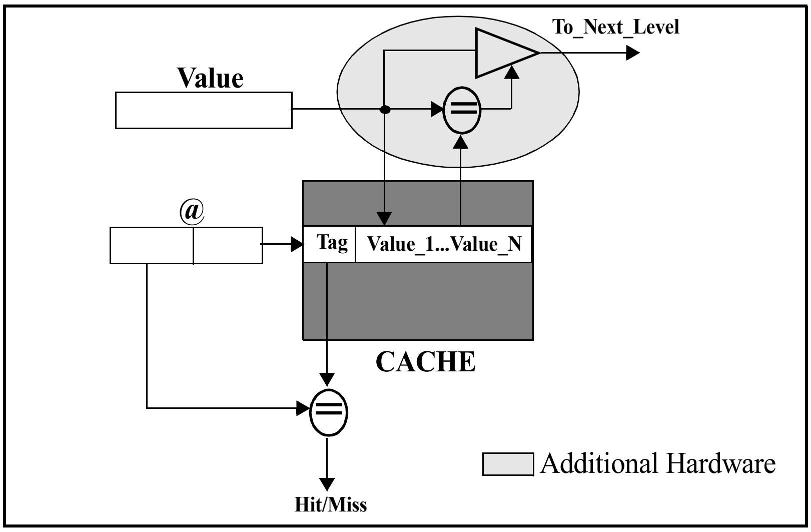


Figure 2.5. Redundant store mechanism with write through

2.3.2. Performance Evaluation

In this section we evaluate the reduction of memory traffic in different cache configurations.

2.3.2.1. Experimental Framework

We have developed a functional parameterized simulator for cache memories, as described in Section 2.2.3.1. This simulator is now modified to provide information about required memory bandwidth. We consider the same subset of the Spec95 benchmark suite and evaluated the same cache sizes and cache write policies (see Section 2.2.3.1 for further details of the specific experimental framework and Section 1.4 for further details of tools and benchmarks).

2.3.2.2. Analysis of Results

Memory traffic between the memory cache and its next level of the memory hierarchy is now analysed. The traffic is computed as the number of bytes transmitted between levels.

Figure 2.6 and Figure 2.7 show the memory traffic for each policy and for several cache sizes. As expected, the memory traffic decreased when we used bigger cache sizes because the miss ratio decreased. Note that memory traffic has two main sources: the hit/miss ratio and the size of the cache block. Intuitively, CB-WA caches should have less memory traffic than WT-NWA caches because they have a better hit ratio (see Table 2.2). Our results show that, on average, CB-WA and WT-NWA caches

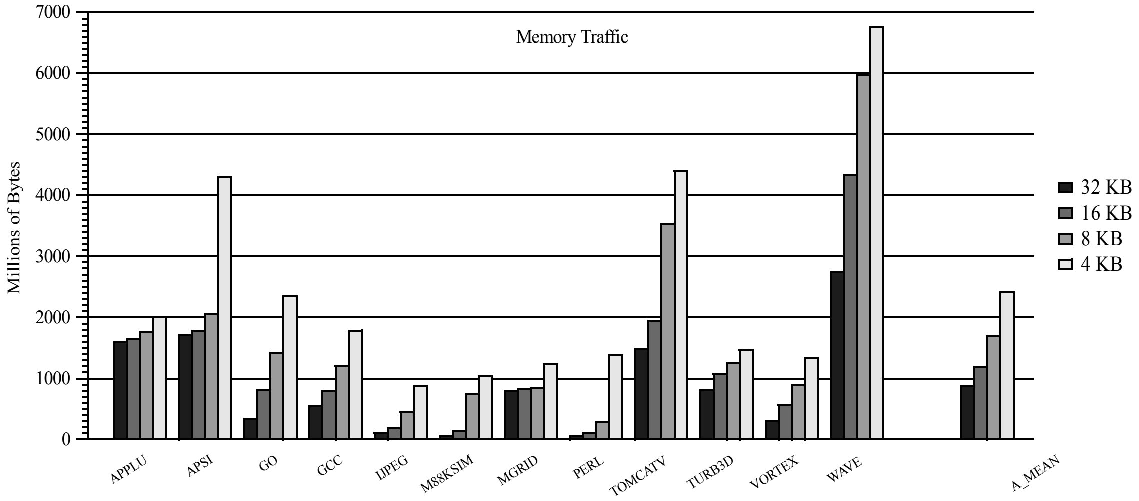


Figure 2.6. Memory traffic (millions of bytes) for CB-WA with different cache sizes

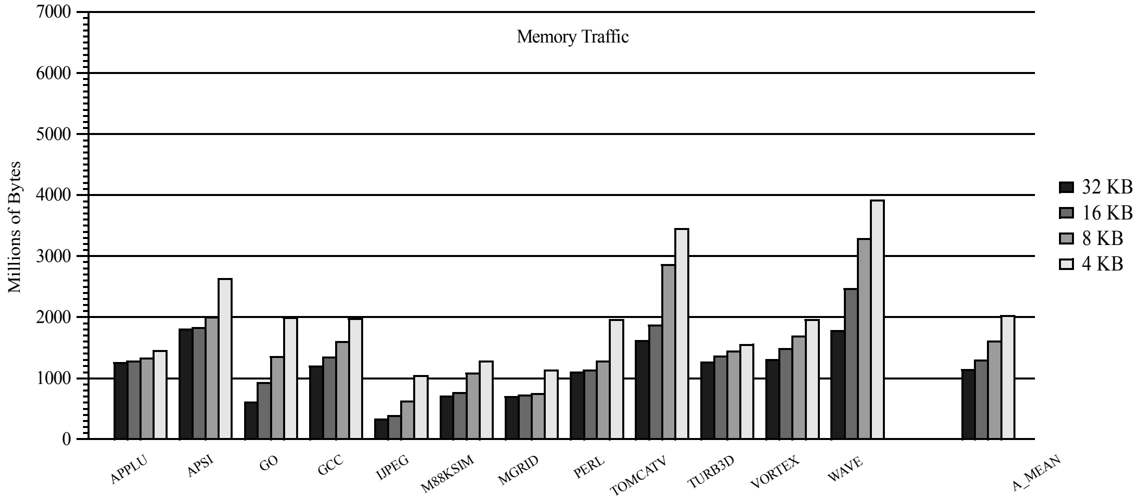


Figure 2.7. Memory traffic (millions of bytes) for WT-NWA with different cache sizes

have similar amount of memory traffic. This is because CB-WA caches have to transmit the whole cache block on a miss if the replaced block is dirty, while WT-NWA only has to transmit a single value for each write.

Figure 2.8 and Figure 2.9 show the percentage reduction in memory traffic after the redundant store mechanism is applied. Figure 2.8 shows the results for CB-WA caches and Figure 2.9 shows the results for WT-NWA caches. Note that both are significant, though WT-NWA caches provide higher percentage reductions than CB-WA caches.

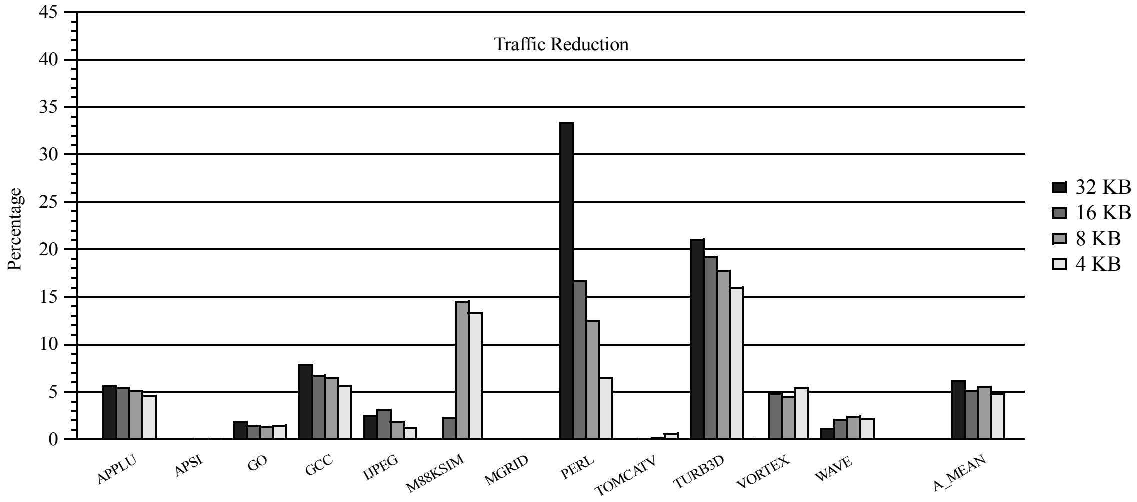


Figure 2.8. Reduction of memory traffic of CB-WA with different cache sizes

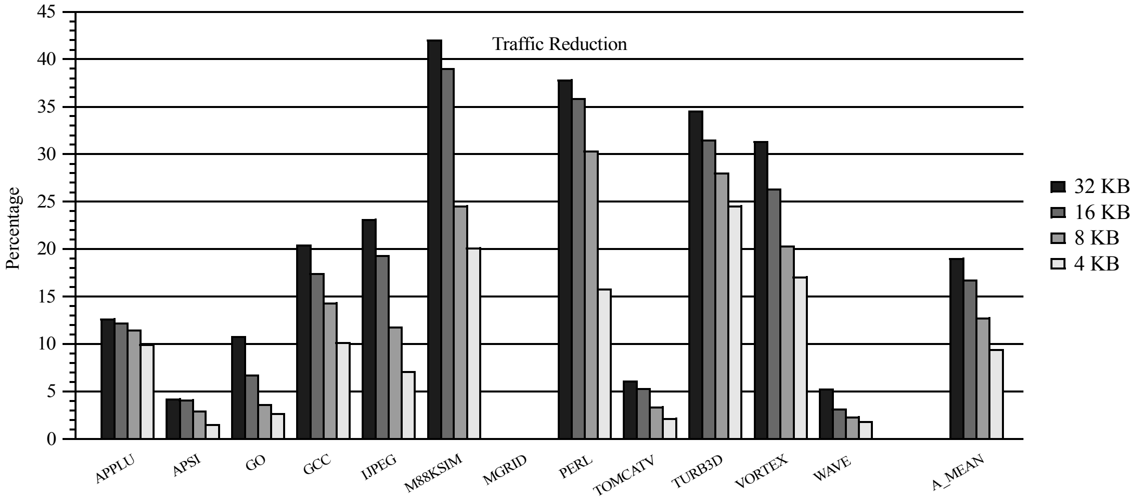


Figure 2.9. Reduction of memory traffic of WT-NWA with different cache sizes

For example, there is a 7% memory traffic reduction for a 32KB CB-WA cache and 19% traffic reduction for a 32KB WT-NWA. This difference is mainly due to the fact that all the stores have to go to the next level of the memory hierarchy for *Write Through*, so every *Redundant Store* leads to a saving on memory traffic. On the other hand, when *Copy Back* is used, every *Redundant Store* reduces the frequency of the dirty bit setting, which does not always lead to a traffic reduction. For instance, two consecutive *Redundant Stores* mapped to the same cache line just reduce the traffic in one block. The same argument explains why the size of the caches affects the percentage reduction of the WT-NWA caches (this increases when the caches get bigger) and does not affect the percentage reduction of the CB-WA caches (which is similar for all sizes).

2.4. Value Repetition in Several Storage Locations

2.4.1. Overview

Several studies [34],[59],[72],[112],[139] have pointed out that value replication is common in different parts of a processor. In this section, we focus on value replication into several storage locations of data caches during the execution of programs. As explained before, Lipasti *et al* [72] coined the term of *value locality* to describe the likelihood of references of a previously seen value within a storage location. We are interested in analysing references of a previously seen value into several storage locations.

We will consider a repetitive value if is frequently stored into several storage locations of the data cache. In this way, we will analyse at any given time the percentage of values that are repeated in a conventional data cache.

2.4.2. Performance Evaluation

In this section we analyse in greater detail the percentage of value replication into several storage locations that can be achieved for individual benchmarks under several scenarios.

2.4.2.1. Experimental Framework

Several configurations of data cache memories have been simulated with sizes ranging from 1KB to 256KB, and different degrees of associativity. The evaluation was done with the sim-cache simulator from the Alpha version of the Simplescalar toolset [15]. The original simulator was modified to collect value repetition statistics.

We now consider the next generation of Spec95 benchmarks: Spec2000. However, Yi and Lilja [135] analysed computation repetition in both Spec95 and Spec2000 benchmarks and did not find significant differences. Thus, the following Spec2000 benchmarks were randomly picked: *crafty, eon, gcc, gzip, mcf, parser, twolf, vortex and vpr* from the integer suite; and *ammp, apsi, art, equake, mesa, mgrid, sixtrack, swim* and *wupwise* from the floating point suite. The programs were compiled with the Compaq C compiler with -O5 -non_shared optimization flags (i.e, maximum optimization). Each program was run with the reference input set and statistics were collected for 1 billion instructions after skipping the initializations (see Section 1.4 for further details of tools and benchmarks).

2.4.2.2. Analysis of Results

In this section we analyse the value replication of several data caches ranging from 1KB to 256 KB. Our conclusion from all the simulations performed is that there is significant value replication in data caches at any given time.

For example, Figure 2.10 shows the value replication of a 256 KB direct-mapped data cache. To obtain this data, the content of the data cache is analysed every cycle and the percentage of replicated 64-bit values is obtained. The X-axis represents the percentage of different values (0% means that all the values in the cache are the same and 100% means that all the values in the cache are different). The Y-axis represents the percentage of total execution time that the corresponding percentage of different values has been observed. Figure 2.10.a is a histogram of the degree of variability and Figure 2.10.b shows the accumulated distribution. We can see that there is a huge degree of value replication. For instance, Figure 2.10.b shows that during 80% of the execution time, less than 25% of the values of the cache are different. This means that on average, a value is stored four times. Note also that this cache never has more than 80% of different values.

Another important observation from these simulations is that the percentage of value replication increases when the caches get bigger. Figure 2.11 shows the accumulated histograms of caches ranging from 1KB to 256KB. As we indicated earlier, there is a huge percentage of value replication at any given time in all caches, but the percentage gets better when the size of the cache increases. For example, the percentage of the values in the cache that are different during 80% of the time changes from 80% in a 1KB cache to 25% in a 256KB cache. This percentage drops dramatically when caches manage the storage locations of megabytes. Finally, we also observed that the value replication degree was not affected by associativity.

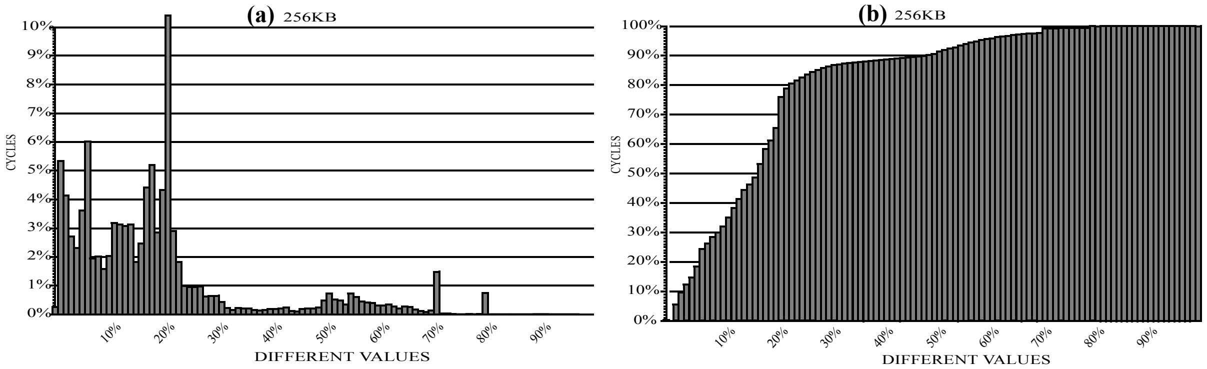


Figure 2.10. Average histograms of a 256KB data cache

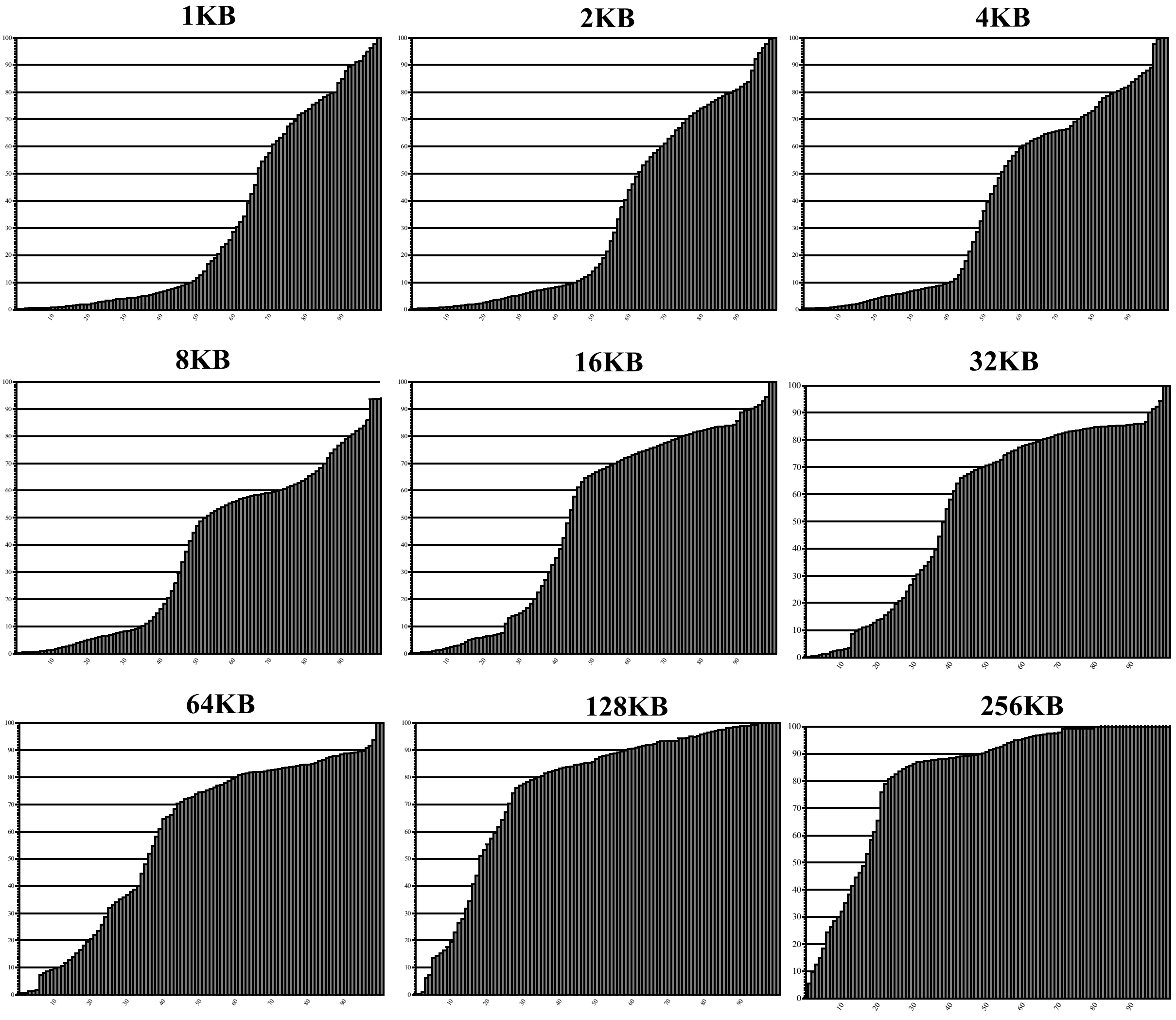


Figure 2.11. Average histograms of data caches ranging from 1KB to 256KB

Figure 2.12 shows the histograms of the degree of variability for benchmarks considered to obtain average numbers of a 32KB data cache. In this figure the Y-axis goes from 0% to 20% because is not an accumulative histogram. Note that only a few benchmarks do not present a significant percentage of replication (i.e. *gzip*, *mesa*, *mgrid* and *wupwise*). The rest are significantly repetitive (i.e. *ammp*, *art*, *crafty*, *eon*, *gcc*, *mcf*, *parser*, *sixtrack* and *vortex* exhibit a high degree of replication) which encourages further work to develop novel techniques to exploit this feature. Moreover, value replication increases when the caches get bigger.

From the observation that many values stored in data caches are repeated, novel cache designs can be proposed and analysed to reduce die area, power dissipation and latency.

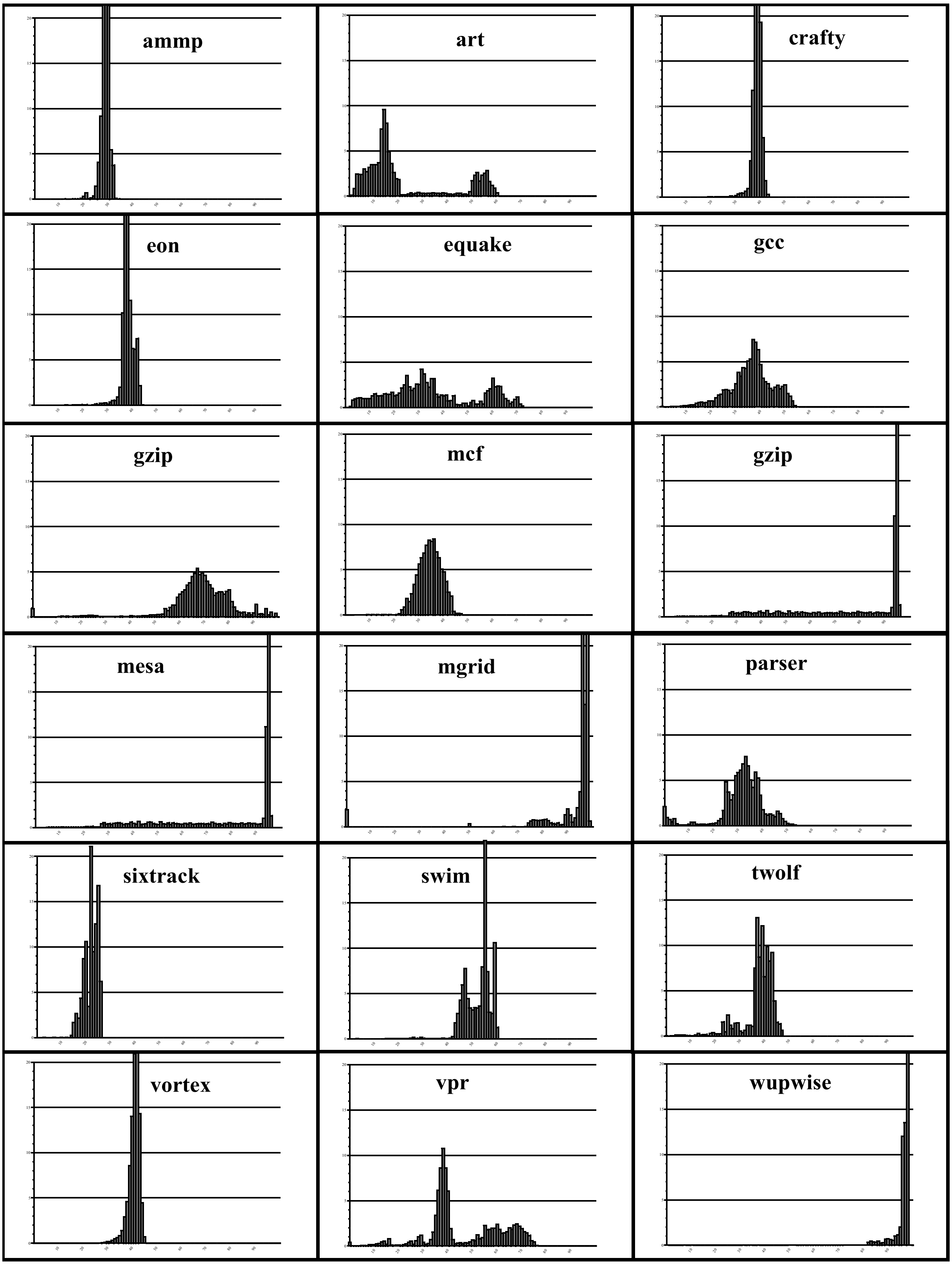


Figure 2.12. Individual histograms of a 32KB level 1 data cache

2.5. Non-Redundant Data Cache

We propose a new cache architecture, which we call *Non-Redundant Cache* to reduce storage area, power dissipation and latency. The underlying concept behind this proposal is based on the observation that data caches exhibit a high percentage of value replication at any given time. We first present a general description and then discuss a simple encoding/decoding scheme that can be applied to the cache. Finally, we evaluate this scheme using a dynamic and static analysis.

As it is shown in Figure 2.11, the percentage of replication gets better when the size of the cache increases. Thus, we concentrate on second-level data caches because they occupy a very important part of the die area in most processors.

2.5.1. General Description

The *Non-Redundant Cache* reduces the storage requirements of data cache by removing the significant amount of replication present in conventional cache designs. Figure 2.13 depicts the proposed design.

The scheme is divided into two areas: the tag area and the data area. The tag area or Pointer Table (PT) stores pointers to the data area or Value Table (VT).

PT is indexed as a conventional data cache. Each PT entry has two fields: a *tag* that identifies the memory address stored in each line, as in a conventional cache, and the *pointer* field that determines, for each 64-bit word in the line, the entry of VT where the value may be found. Note that additional storage is required to maintain pointers but that values are not replicated in the VT. Therefore, any word of any cache line of PT that has the same value points to the same location in VT.

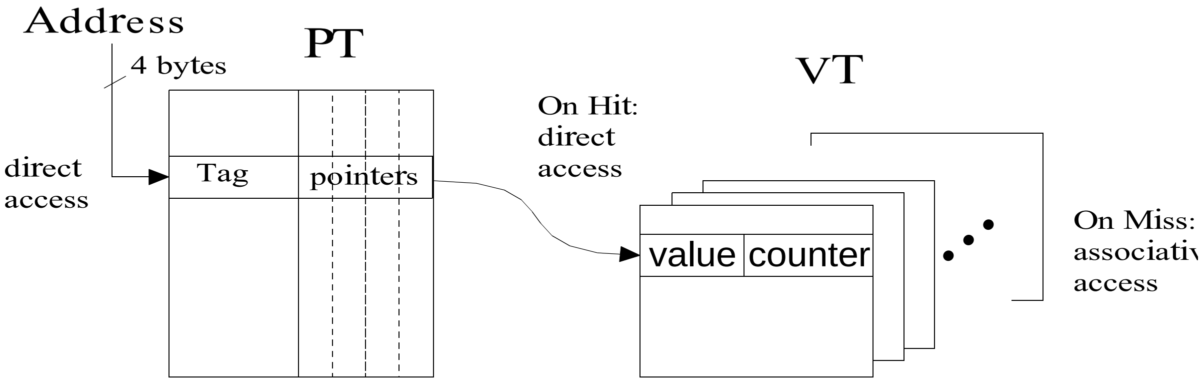


Figure 2.13. Block diagram of the Non-Redundant Cache

The location of a value in the VT is determined by the value itself, i.e. the VT is indexed by values. Each VT entry also has two fields: a full 64-bit data value and a counter. This counter maintains the number of pointers from PT that point to that value. When the counter reaches its maximum value, no further lines are allowed to point to this VT entry. If a new line needs to point to this value, the value is replicated and stored in a new VT entry with its counter initialized to 1. We have experimentally confirmed that when 4-bit saturating counters are used, less than 1% of the values of the VT have some replica (note that narrow values, which are some of the most repeated ones, are not stored in the VT but in the PT, as we will describe in the next section).

The behaviour of the *Non-Redundant Cache* can be summarized as follows:

- Miss on Read or Write: the new line is brought from the upper level of the memory hierarchy. At this time, all the values try to be stored in the VT. Finally, PT has to be updated with the new tag and pointers. The search for room in the VT is simple and may be summarized as follows. First, a value search is performed in the VT. If the value is found, the associated counter is increased and that entry will determine the content of the pointer in the PT. If the value does not exist, a free entry is required. A VT entry is considered free if its associated counter is zero. If no free entry is available, the value cannot be stored in the VT. Replacement in the PT is implemented in the same way as in a conventional cache. When a line from the PT is replaced, its associated pointers are used to decrement the counters of the corresponding values.

- Hit on read: if there is a hit in the PT, the associated pointer provides the index to the VT where the value will be found.

- Hit on write: the new value is searched in VT as explained above and a new VT entry is allocated if it is not found. The counter of the old value is decreased.

Note that a line in the PT may not have all its pointers valid. An invalid pointer is produced when its associated value cannot be stored in the VT.

The indexing function for the VT is an important parameter for the performance of the *Non-Redundant Cache*. We assume a direct-mapped VT i.e. the least significant bits of the values (ignoring the least significant two) determine the location in the VT. To reduce aliasing in VT, an associativity of 8 is considered. This configuration works fairly well but is still far from the behaviour of the full associative approach.

Note finally that the VT does not need to store all the bits of a value. The least significant bits are implicitly given by the set that the value occupies (in the same way as the tags of a cache do not contain the least significant bits).

2.5.2. Data Value Inlining

A further enhancement can be applied since the tag area of the *Non-Redundant Cache* provides some storage for pointers. In particular, narrow values (i.e. values that can be represented by a small number of bits) can be inlined into the tag area. The idea is that if a value can be represented in the number of bits allocated for a pointer in the PT, instead of storing the value in the VT and setting up a pointer from the PT to the VT, the value can be directly stored in the pointer location itself. Narrow values are very common [48], [124] so this simple extension can provide significant benefits. We will refer to a value stored in the PT as a *narrow value*. This improvement not only enlarges the logical capacity of VT, it also reduces latency and power dissipation because the access to the VT is avoided for narrow values.

The extensions to support narrow values are quite simple. Each value of a new line brought on a cache missed is tested. This basically consists of an OR and an AND gate applied to the most significant bits to check whether they are all ones or all zeroes. If a value is narrow, there is no need to search for store in the VT: it is stored in the corresponding pointer field. A bit in each pointer field is intended to indicate whether its content is a pointer or a narrow value. The search for room in the VT is always done after a Miss. For each value in the line, a narrow test is performed. If the value is in the selected range, there is no need for a search in the VT. Value is managed as narrow and compressed in the pointer field of the PT, setting an additional narrow bit.

On the other hand, if there is a hit on the PT and the value is narrow, the value is provided by the PT and the access to the VT is avoided. The value obtained from the PT is sign-extended before it is sent to the processor data path.

Finally, note that in case of a miss, if due to the lack of space none of the values of the new line can be inlined or stored in the VT, the line is not stored in the *Non-Redundant Cache*.

2.5.3. Static Analysis

In this section we evaluate the cache area, latency and power dissipation of the *Non-Redundant Cache* using the CACTI tool version 3.0 [107]. CACTI 3.0 is a cache area, access and cycle time and power consumption estimation tool that is widely used to estimate power dissipation in caches and to determine the cache configuration that best satisfies the desired optimization criterion.

The CACTI tool has been adapted to model the structure of the *Non-Redundant Cache* and the processor technology assumed is 0.09µm.

2.5.3.1. Die Area

This evaluation focuses on the tag and data blocks since they represent the vast majority of the cache area. For instance, the total area of a 512KB direct cache with 32 bytes per line is: 0.1386 cm^2, whereas the data and tags occupy 0.1376 cm^2 (99.3%).

The PT has the same number of lines and associativity as the conventional cache used as baseline. Note that the PT and the baseline cache have the same tag area, but the PT also stores the pointer fields. The number of bits in a pointer field depends on the size of the VT. On the other hand, the VT is managed as a table when accessed through a pointer of the PT (i.e. the pointer indicates the precise location of the value) and is indexed as a set-associative cache when new values have to be stored.

Figure 2.14 shows the total die area for different cache configurations for a 32-byte cache line size. Figure 2.14.a and Figure 2.14.b correspond to a 512KB and 2MB baseline L2 data cache respectively. Several Pointer Cache configurations are depicted. VTxx means a Pointer Cache with a VT capacity reduced to xx% of the baseline size (e.g., VT30 means a VT capacity reduced to 30%). The different

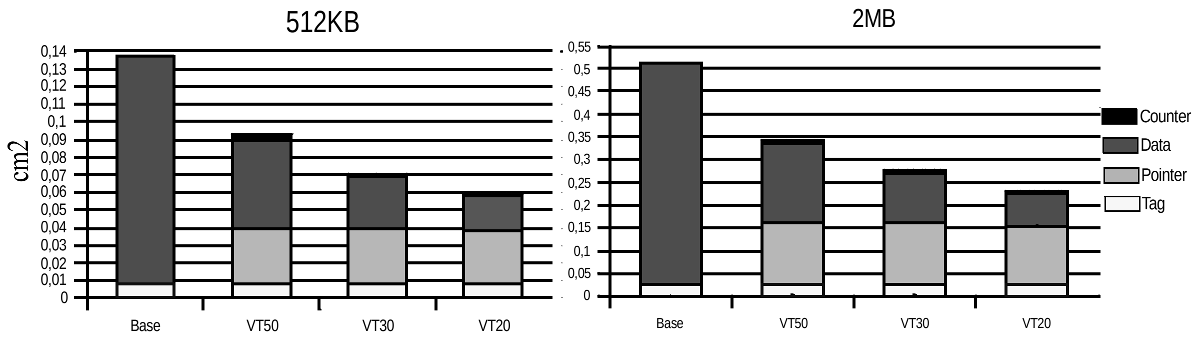


Figure 2.14. Static analysis: cache area

colours in each bar represent the contribution of tags, pointers and data. As expected, the area reduction was significant in all cases. For instance, VT30 achieved a reduction of 49% with respect to the 512 KB baseline cache. Below, we discuss how these reductions interact with the miss ratio. Statistics for other cache capacities ranging from 256KB to 4MB are given in Table 2.3.

2.5.3.2. Latency

We also evaluated the access time of the various cache architectures with the CACTI tool. Figure 2.15 shows the critical path of an access to the *Non-Redundant Cache*.

When an address is sent to the *Non-Redundant Cache*, a direct-mapped access (other indexing functions are also possible) is performed to the PT, which provides the tag and the pointers at the same time. The corresponding pointer is selected and used to access the VT. The tag comparison is done in parallel with the access to the VT.

If there is a miss, an access to the next level of memory is started. As the values of the line arrive from the next memory level, their corresponding ranges are checked. Depending on this range, each value is stored in the VT or just inlined in the PT. If the cache line is stored in the *Non-Redundant Cache*, a line from the *Non-Redundant Cache* is replaced. The replaced entry in PT is read and all their valid pointers determine the counters to be decreased in the VT.

The behaviour of the *Non-Redundant Cache* for a hit can be summarized as follows: (a) the PT is accessed and the corresponding tag and pointers are checked; (b) if the value is not inlined, an access to

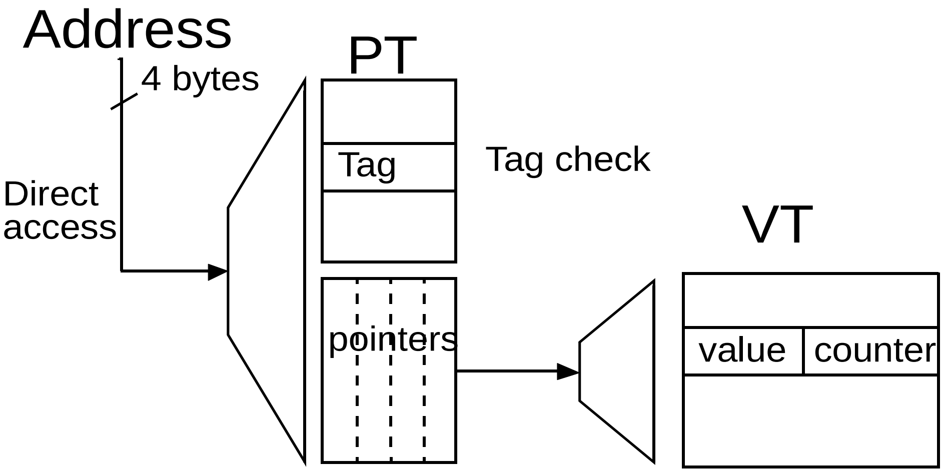


Figure 2.15. Critical path of the Non-Redundant Cache

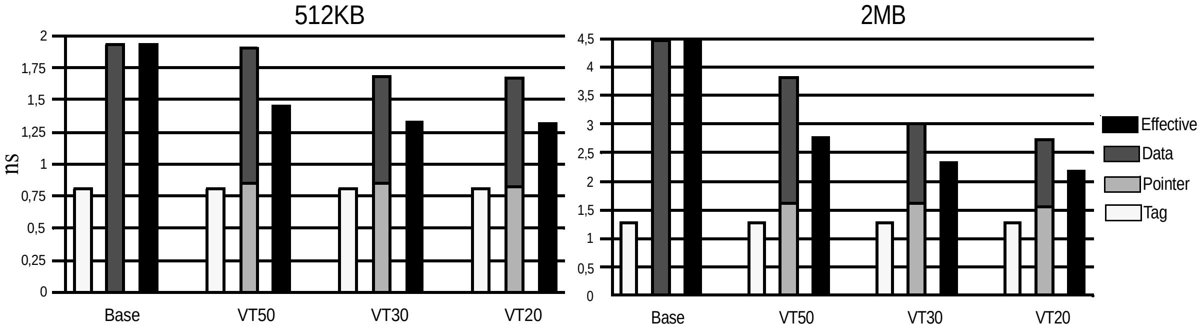


Figure 2.16. Static analysis: latency

VT is performed. Total access time is the maximum of tag access and comparison or the pointer access and the VT access.

Figure 2.16 shows the different time components of a cache access. We considered the same configurations as in Figure 2.14. Three bars are shown for each configuration. The first bar depicts the time required for the tag check. The second bar determines the time required for accessing the value. Note that this bar is split into two components for the *Non-Redundant Cache*: pointer access and VT access. The third bar represents the effective data access time, considering the percentage of accesses (see Figure 2.19) that are satisfied by the PT and the percentage that requires an access to the VT.

The latency of the *Non-Redundant Cache* drops significantly as the size of the VT decreases. Overall, the *Non-Redundant Cache* offers a much shorter latency than a conventional organization. For example, VT30 achieves a reduction in access time of 49% with respect to the 512 KB baseline cache. As reported below, the latency advantage of the *Non-Redundant Cache* is even greater for larger cache sizes.

2.5.3.3. Energy Consumption

We also evaluated energy consumption by extending the CACTI tool for the *Non-Redundant Cache*. The *Non-Redundant Cache* provides significant benefits in terms of energy consumption because the energy consumption of memory structures depend, among other things, s on their area and the total number of bits read/written.

Figure 2.17 shows the energy consumed per each access for different types of accesses. *Hit inlined* corresponds to an access that finds the value inlined in the pointer field. *Hit VT* corresponds to an access for which the value is obtained from the VT. This involves an access to the PT plus an access to

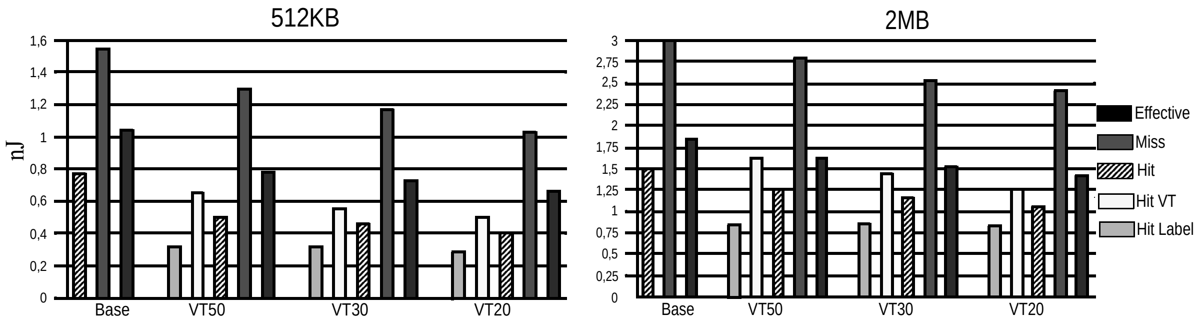


Figure 2.17. Static analysis: energy consumption per access

the VT. *Hit* is the average energy per access for hits, considering the percentage of accesses that hit in the pointer field and the percentage that hit in the VT. *Miss* corresponds to an access that misses in the *Non-Redundant Cache* and has to be served by the next level of memory. This involves two accesses to the PT (the first access determines a miss and the second access updates the PT pointers) and a number of accesses to the VT that depends on the number of values that can be inlined. These numbers have been obtained through simulation (see Figure 2.19 below). Finally, *Effective* shows the average energy consumed by access.

Table 2.3 shows savings for caches ranging from 256KB to 4MB. Note that the reduction in power is very significant across the whole range of capacities.

2.5.4. Dynamic Analysis

The simulation environment is built on top of the Simplescalar [15] Alpha toolkit, which has been modified to model the *Non-Redundant Cache*.

The following Spec2000 benchmarks were considered: *crafty, eon, gcc, gzip, mcf, parser, twolf, vortex and vpr* from the integer suite; and *ammp, apsi, art, equake, mesa, mgrid, sixtrack, swim* and *wupwise* from the FP suite. The programs were compiled with the Compaq C compiler with -non_shared -O5 optimization flags (i.e, maximum optimization). Each program was run with the reference input set and statistics were collected for 1 billion of instructions after skipping the initial part of initializations.

2.5.4.1. Replication in Conventional L2 Caches

In this section we study the degree of value replication in conventional L2 caches considering a 256 KB second level data cache.

Figure 2.18 shows the percentage of values that are different in a conventional organization. Two bars are depicted for each program. The first bar corresponds to the percentage of different values observed during 99% of the execution time. The remaining 1% was not considered in order to disregard the effect of rare cases that would have little influence on the total execution time. The second bar reports the percentage of values that are different and cannot be inlined. A value is considered to be inlinable if it can be represented with no more than 10 bits.

There was a significant degree of value replication. For most benchmarks, less than 25% of the values of the L2 cache were different. *Gzip*, *mgrid* and *vpr* had the lowest degree of replication, though this was not negligible. On average, only 28% of the values were different, which can be translated into significant gains for the *Non-Redundant Cache*. Benefits are even greater for larger cache capacities (see below). For example, for a 4 MB data cache only 15% of the values were different.

2.5.4.2. Inlining Performance

Another result of Figure 2.18 is that inlining slightly reduces the storage required by values that are different (i.e., the storage required in the VT of the *Non-Redundant Cache*) for integer benchmarks, but

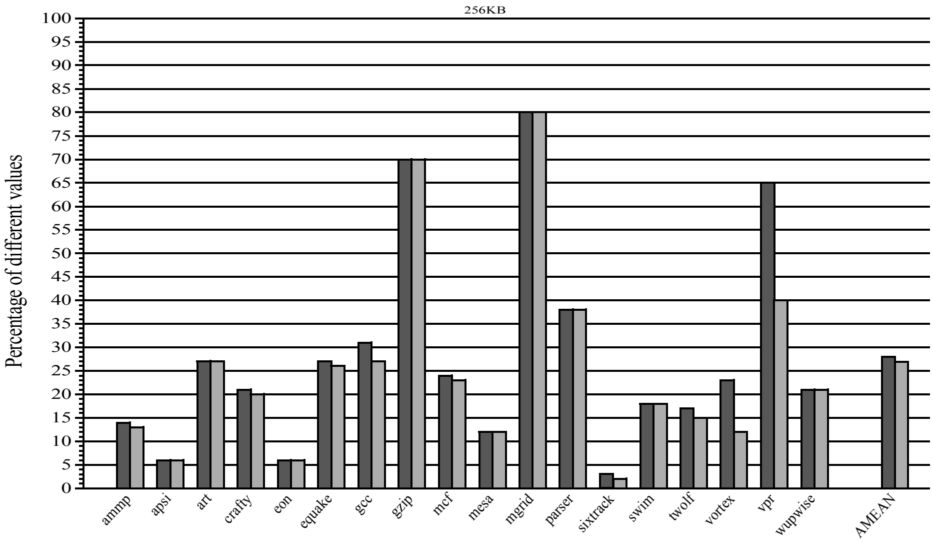


Figure 2.18. Percentage of different values for a 256KB L2 data cache

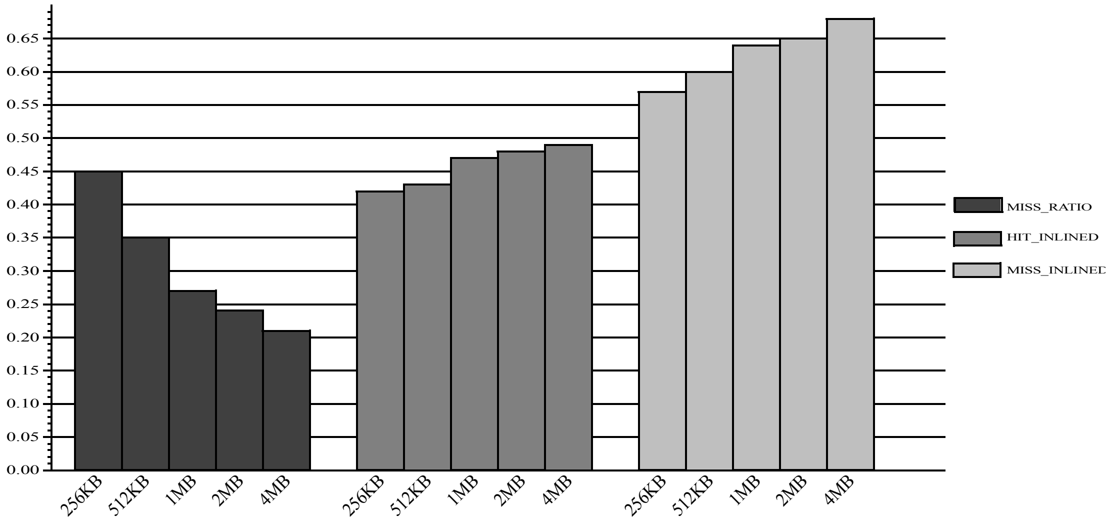


Figure 2.19. Miss ratio, hit inlined and miss inlined

its effect is almost null for FP programs. However, inlining has another important benefit. Since in general the inlined values are those that have the highest degree of repeatability (e.g., 0, 1 and -1 are usually the most frequent values) [139], inlining these values allows the use of a very small counter in the VT table (see Section 2.5.1).

Figure 2.19 presents further statistics for caches ranging from 256KB to 4MB. The dark grey bars show the miss ratio. The medium grey bars depict the hit ratio to inlined values. Note that the percentage increases as the caches get bigger and the range of inlined values increases, since the pointers in the PT require more bits to point to a bigger VT. On average, between 42% and 49% of the memory requests do not need to access the VT.

Finally, the light grey bars show the percentage of values that are brought from the next memory level on a miss and can be inlined. On average about two thirds of the values can be inlined and this percentage increases as the caches get bigger. This clearly shows out that inlining is very effective.

2.5.4.3. Miss Rate vs. Die Area

To analyse the advantages of the *Non-Redundant Cache*, we consider the following scenario: a second level data cache ranging from 256KB to 4MB. All cache configurations were considered to be direct-mapped with a cache line of 32 bytes. The PT is dimensioned to store the same number of lines as the base configurations but reducing the VT to 50%, 30% and 20% of the original size. The VT is dimensioned to store values of 8 bytes, so there are 4 pointers in each entry of the PT in addition to the tag. Each line of the VT stores an 8-byte value and a 3-bit as counter that determines the number of pointers from the PT that refer to that value.

	Die Area Reduction			Energy Consumption Reduction			Access Time Reduction			Number of Misses Increment		
Cache	VT50	VT30	VT20	VT50	VT30	VT20	VT50	VT30	VT20	VT50	VT30	VT20
256KB	32%	47%	54%	13%	18%	23%	8%	17%	22%	2.8%	7.7%	11.5%
512KB	33%	49%	57%	25%	30%	37%	25%	31%	32%	4.3%	11.6%	16.5%
1MB	31%	46%	55%	19%	26%	33%	25%	30%	30%	6.8%	14.4%	20.5%
2MB	33%	46%	55%	12%	18%	23%	38%	47%	51%	6.5%	14.8%	21.8%
4MB	31%	48%	55%	0%	14%	18%	32%	42%	48%	5.1%	11.8%	19.5%
AMEAN	32%	47.2%	55.2%	13.8%	21.2%	26.8%	25.6%	33.4%	36.6%	5.1%	12%	17.9%

Table 2.3. Comparison results.

Table 2.3 shows the main statistics for each cache configuration considered. These results show that the *Non-Redundant Cache* is very effective for the second-level cache. For instance, a configuration with a VT with 50% of the capacity of the baseline leads to an average die area reduction of 32%, an energy consumption reduction of 13.8%, an access time reduction of 25.6%, and a very minor increase (5.1%) in the number of misses. The increase in misses reported in Table 2.3 for VT30 and VT20 is only due to three of the eighteen benchmarks simulated. These are the three benchmarks with the lowest degree of replication (see Figure 2.18): *mgrid*, *gzip* and *vpr*. For the remaining fifteen benchmarks, the *Non-Redundant Cache* does not lead to more misses than a conventional cache.

Figure 2.20 plots the miss ratio against the area for several configurations. The X-axis shows the total area required in cm^2. The line labelled as 100% corresponds to a conventional cache. The other lines correspond to a *Non-Redundant Cache* with sizes of the VT reduced. These reductions range from 20% to 50% of the reference cache. The dots in a line correspond to configurations with different numbers of entries in the PT, corresponding to the number of entries of a conventional cache with 256KB, 512KB, 1MB, 2MB and 4MB.

The results show that the *Non-Redundant Cache* outperforms the base configuration. For a fixed die area, the *Non-Redundant Cache* provides a significant reduction in miss ratio. For instance, a 256KB cache base configuration has a miss ratio of 45%, whereas the *Non-Redundant Cache* has a miss ratio of 39%, 38% and 37% for a VT reduced to 50%, 30% and 20%, respectively. Alternatively,

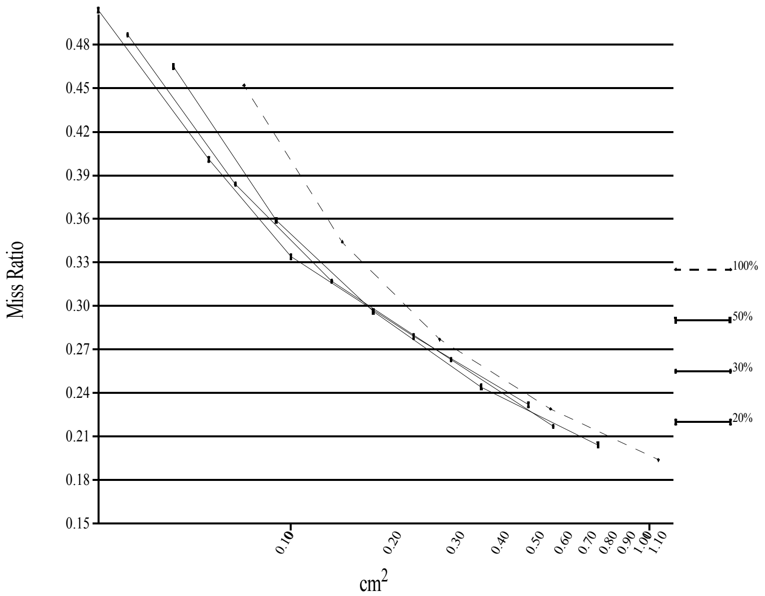


Figure 2.20. Miss ratio vs. die area for second level data caches

the *Non-Redundant Cache* provides the same miss ratio as a conventional cache but with a smaller area. For example, a 256KB base configuration with a miss ratio of 45% occupies 0.072 cm^2, whereas the *Non-Redundant Cache* achieves the same miss ratio with a die area of 0.054, 0.048 and 0.044 cm^2 for a VT reduced to 50%, 30% and 20%, respectively.

2.6. Related Work

Cache memories were proposed by Maurice Wilkes [128] from an idea by Gordon Scarott, as a way to ease the gap between memory access and processor speed. Goodman [42] concluded that using cache memories has often aggravated the bandwidth problem rather than reducing it. Later, Goodman and Hsu [41] showed that registers can be more effective in reducing the bus traffic than cache memory of the same size. Surendra et *al* [121] combined load instruction reuse and prefetching to reduce data cache traffic in network processors units. A novel prefetching scheme that improves performance without increasing memory traffic was also proposed by Zhang and Gupta [138]. These authors observed that a significant percentage of dynamically appearing values have characteristics that enable a simple compression scheme. Lee *et al* [64] explored a selective compressed memory system that can increase the effective memory space and effective bandwidth of each level of memory hierarchy. Finally, Hallnor and Reinhardt [45] proposed a similar memory hierarchy that increases the effective bandwidth of interconnects by storing and transmitting data in compressed form.

Redundant Stores can be considered trivial operations. The first study to explore the degree of triviality in computation was done by Richardson [91],[92] and was based on the observation that simple operands trivialize potentially complex operations. This author also proposed a mechanism for exploiting this triviality by restricting the definition of trivial computations to certain multiplications, divisions and square roots of 0 and 1. Yi and Lilja [136] extended the method of detecting and eliminating trivial computations proposed by Richardson. The key differences were the types of benchmarks used and the scope of the definition of trivial computations. Unfortunately, they did not explored memory operations as trivial operations.

Since our initial and novel definition of *Redundant Stores* [80], several studies have been done in this area. Lepak and Lipasti [65] proposed the same concept of *Redundant Stores* and called them *Silent Stores*. Like us, they used the redundant store concept to reduce the number of dirty cache lines and so reduce the number of writebacks in a uniprocessor system memory hierarchy. They also used this concept to reduce address and data bus traffic on shared memory multiprocessors. Bell *et al* [10] analysed redundant store instructions in several benchmarks in the context of their high-level source code and explained why they occur. They showed that *Redundant Stores* occur in all levels of program execution and compiler optimization. They also introduced the notion of critical silent stores, described how to find them, and showed that removing the small subset of critical *Redundant Stores* is sufficient for removing all the avoidable cache writebacks. Later, Lepak and Lipasti [66] proposed the concept of free silent store squashing, which uses existing resources with slight modifications to squash a significant portion of all *Redundant Stores*, and explained three ways of implementation. Silent store squashing was also explored by Kim and Lipasti [60]. Lepak and Lipasti [67] extended the definition of silent stores (equivalent to *Redundant Stores*) to encompass sets of stores that change the value stored at a memory location (but only temporarily) and subsequently return a previous value of interest to the memory location. Stores that cause the value to revert are called temporally silent stores. They described a practical mechanism that detects temporally silent stores and removes the coherence traffic they cause in conventional multiprocessors. Finally, Purser *et al* [90], [119] used the concept of *Redundant Stores* for slipstream processors and Roth *et al* [95] used it for dynamic techniques for load scheduling.

Several recent studies focus on the value replication phenomenon in data caches. Zhang *et al.* [139] coined the term *frequent value locality*. These authors observed that a few values appear frequently in the memory locations involved in a large fraction of memory accesses. From this observation, they proposed the design of the FCV (Frequent Value Cache). The FVC only contains frequently accessed values stored in a compact encoded form and is used in conjunction with a traditional direct-mapped cache. Both caches are accessed in parallel to provide memory values. Yang *et al* [132] presented a similar cache design called CC (Compression Cache) in which each line can hold one uncompressed line or two cache lines that have been compressed to at least half of their lengths. A modification of the FVC was proposed by Yang and Gupta [130] to improve energy efficiency. In their study, the data array is portioned into two arrays such that if a frequent value is accessed, the first data array is the only one accessed. Otherwise, an additional cycle is needed to access the second data array, which stores values in an unencoded form.

Value replication into the register file was also studied. Balakasharian and Sohi [8] considered the use of value locality to optimize the operation of physical register files and proposed three schemes. Later, González *et al* [40] presented a new integer register file organisation that reduces the energy consumption, die area and access time of the register file with a minimal effect on overall IPC by exploiting partial value locality, which is defined as the occurrence of multiple identical live value instances in a subset of their bits.

Significance compression was used by Brooks and Martonosi [13] and Canal *et al* [19] to reduce power dissipation, not only in data cache but also in the full pipeline. Other data compression schemes were also presented by Larin [63] and Villa *et al* [124]. Larin explored different encoding schemes for a large number of values that dynamically adapt to the different distributions of data values. Villa *et al* proposed a dynamic zero compression technique to reduce cache energy by taking advantage of the high-frequency occurrence of zero-valued bytes in the cache. Register file was also analysed by Ergin *et al* [33], who observed that a large percentage of computed results have fewer significant bits than the full width of a register. These authors exploited this fact by packing multiple results into a single register file to reduce the pressure. Narrow width values were also exploited by Sato and Arita [102] to reduce the huge hardware budget of conventional data value predictors.

Finally, Black *et al* [12] introduced a mechanism called block-based trace cache, that also uses pointers to avoid replication in the instruction trace cache, and Collins et *al* [27] proposed the *Pointer Cache*, which tracks pointer transitions in order to aid prefetching. The *Pointer Cache* provides a prediction of the object address indicated by a particular pointer

2.7. Conclusions

In this chapter we have analysed value repetition in the same location of the memory hierarchy and introduced the concept of *Redundant Store Instructions*. This concept is based on the particular behaviour of some writes to memory that do not change their contents because the value they write is equal to the existing value. In particular, we have shown how *Redundant Stores* can be applied to current cache organizations. We have presented a simple mechanism for reducing the memory traffic between cache and the next level of the memory hierarchy. This novel idea requires minimal hardware support. We have shown that we can achieve a significant memory traffic reduction in the memory hierarchy. On average, this is close to 7% for a cache with *Copy Back-Write Allocate* and 19% for a cache with *Write Through-No Write Allocate.*

Note that this particular behaviour of *Redundant Stores* can be exploited in other parts of a superscalar processor such as load/store queues for memory disambiguation, buffers of instruction reuse schemes, etc. In fact, it can be applied to any part of the processor that has a buffer and is modified. For instance, cache memories are commonly designed with a write buffer to reduce the overhead associated with write operations [58]. A write buffer can cause processor stalls when memory operations find this buffer full [108]. The redundant stores mechanism decreases processor stalls because it reduces the number of stores to be sent to the write buffer and consequently the probability of filling it up. Moreover, multiprocessors will also benefit from the reduction in memory traffic generated by cache coherence protocols [47]. *Redundant Stores* do not need to send invalidations/updates to other processor caches. In summary, we conclude that the redundant store mechanism can increase system performance and reduce power consumption in processor or multiprocessor systems.

We have also shown the high degree of value repetition into several storage locations of conventional data caches at any given time. This value replication increases as caches get bigger. From this observation, we have presented a novel data cache design called *Non-Redundant Cache* that avoids the replication of values. The underlying concept of our proposal is based on leveraging this phenomenon in order to reduce the area, power dissipation and access time. The *Non-Redundant Cache* also includes a simple compression scheme based on inlining narrow values for values that require fewer bits than their corresponding pointers. Simulation results show that the N*on-Redundant Cache* outperforms conventional caches in terms of power dissipation, access time and die area at the expense of a very minor increase in miss ratio.

“If you want the present to be different from the past, study the past”
Baruch Benedict Spinoza, Dutch Philosopher, 1632-1677.

Chapter 3

INSTRUCTION LEVEL REUSE

Data value reuse techniques were proposed based on the idea that previous work by instructions can be non-speculatively reused when they perform the same work again. Instruction-level reuse is a special implementation of data value reuse that avoids the execution of single instructions

In this chapter we analyse in detail the performance potential of instruction-level reuse and propose a novel microarchitectural technique that can benefit from computation repetition to boost the execution of instructions.

3.1. Introduction

Computations performed by programs tend to be repetitious in the sense that most of the data consumed or produced by different dynamic instructions are often the same [35],[36],[37],[71],[72],[111],[112]. This is a result of the concise and structured way in which programs are written. For instance, different activations of the same routine may lead to the repetition of several instructions with the same source operands if just a subset of the input parameters varies. Techniques to exploit this phenomenon by reusing previously computed data instead of recomputing them will be referred to as *data-value reuse* techniques. *Instruction-level reuse* is a special implementation of data-value reuse intended to avoid the execution of single instructions that produce the same result as previous instructions.

Instruction-level reuse exploits this by buffering previous inputs and their corresponding outputs. When an instruction is fetched and decoded and its current inputs are found in that buffer, its execution can be avoided by obtaining the outputs from the buffer. A dynamic instruction can reuse the result of a previous instance of the same static instruction or an instance of any other static instruction. In the first case a *quasi-invariant*, defined as a computation that is repeated many times and often produces the same result, is removed. In the second case a *quasi-common subexpression*, defined as a computation that often produces the same result as another piece of code, is removed.

An example is given in Figure 3.1. If arrays `b` and `c` both have many repeated elements, the computation `b[i]+c[i]` will become quasi-invariant at run time (instruction *I1* in assembly code).

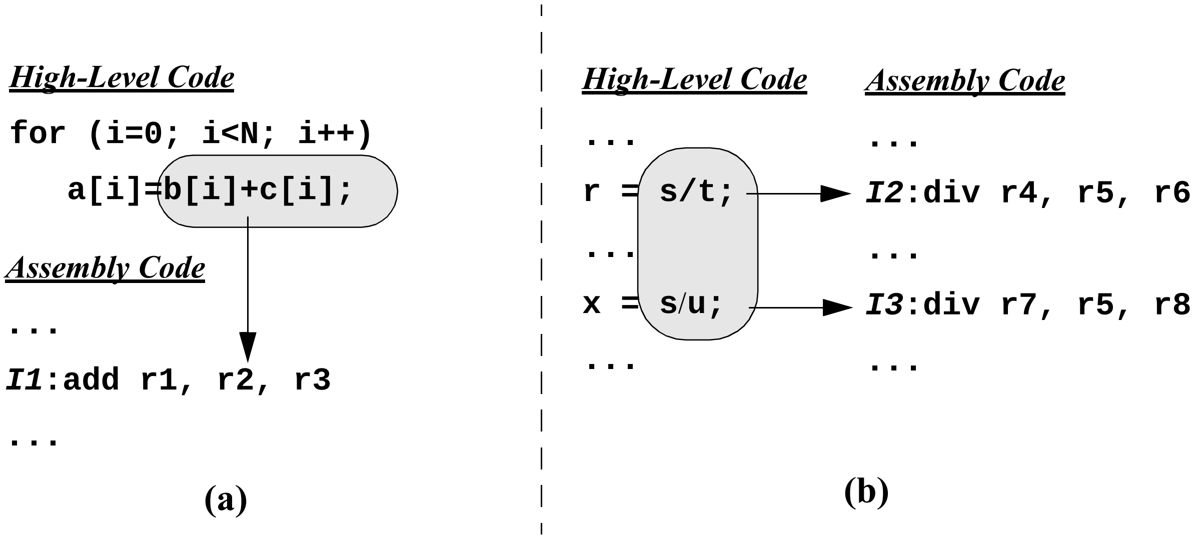


Figure 3.1. Examples of a) quasi-invariant and b) quasi-common subexpression.

Similarly, if `t` and `u` have many repeated elements, expressions `s/t` and `s/u` will be quasi-common subexpressions (instructions *I2* and *I3* in assembly code). Note that quasi-invariants refer to computation repetition produced by the same static instruction or program counter, while quasi-common subexpressions refer to computation repetition produced by several static instructions.

Instruction-level reuse has two positive effects: it lowers the latency of some instructions and, since reused instructions do not go through the issue and execution phases of the pipeline, reduces the contention of the processor resources. Note that instruction-level reuse has the same objective as some compiler optimizations. However, the compiler can only remove some redundant computations because it has limited knowledge of the data. Also, it usually cannot identify quasi-redundant computations, such as those in Figure 3.1.

In this chapter we analyse the concept of instruction-level reuse, discuss its implementation and its performance impact. In Section 3.2 we analyse the performance potential of instruction-level reuse under several scenarios. While ignoring aspects related to implementation, our aim is to understand the nature of this phenomenon and the potential benefits of this technique. In Section 3.3 we propose a mechanism to take advantage of computation repetition and therefore remove at run time quasi-invariants and eliminate quasi-common subexpressions. We show that this mechanism outperforms the execution time of previous proposals. We also propose extensions to previous schemes and evaluate them in detail in order to determine the best configuration for a given cost. Finally, in Section 3.4 we review related work and in Section 3.5 present our main conclusions.

3.2. The Performance Potential of Instruction Level Reuse

In this section, we aim to understand the instruction-level reuse phenomenon and investigate the performance potential of this technique. We will try to isolate the effect of instruction-level reuse from other microarchitectural aspects by putting it into an ideal machine context, with either a limited or unlimited instruction window. We are interested in evaluating the performance limits of instruction-level reuse rather than focusing on any particular implementation. To perform this study we need to clarify a couple of issues.

First, notice that instruction-level reuse exploits the fact that an instruction has appeared in the past with the same input values. Therefore, the maximum instruction-level reuse can be evaluated by analyzing the dynamic instruction stream and counting how many times a static instruction appears with the same inputs as before.

Second, notice also that instruction-level reuse is performed by caching the inputs and the results of previous dynamic instructions. Therefore, a reuse requires the following operations: a) read the input operands; b) perform a table lookup; and c) write the output operands. Even though some of these tasks can be overlapped (e.g. (a) and (b) if the table is indexed by the instruction address), the whole reuse operation will take some time, i.e. any reuse engine will have a given *latency*. To investigate this issue, we will consider an ideal reuse engine with parameterized latency.

3.2.1. Evaluation Methodology

We consider two different scenarios to evaluate the performance potential of instruction-level reuse: an ideal machine with an infinite window and an ideal machine with a limited window. Both scenarios assume a machine with an infinite number of functional units. In this way, we do not consider the benefit of reducing functional unit contention, which, due to the continuous increase in transistors per chip, will have a low impact in future high-performance processors. Moreover, when the number of functional units is a bottleneck, increasing the number of functional units is more cost-effective than implementing a reuse scheme.

For the infinite window scenario, the execution time is only limited by data dependences among instructions, both through register and memory. For the limited window scenario, the execution order inside each sequence of W instructions, where W is the instruction window size, is limited only by data dependences, whereas any pair of instructions at a distance greater than W are forced to be sequentially executed. We compute the IPC for each scenario as an extension of the approach by Austin and Sohi [6].

3.2.1.1. IPC for an Infinite Window Machine

The IPC for an infinite window machine is computed by analyzing the dynamic instruction stream. For each instruction, its completion time is determined as the maximum completion time of the producers of all its inputs plus its latency. The inputs of an instruction may be register or memory operands. Therefore, for each logical register and each memory location, the completion time of the latest instruction that has so far updated such storage location is kept in a table. The latency of the instructions has been borrowed from the latency of the Alpha 21164 instructions [32].

Once all the dynamic instruction stream has been processed, the IPC is computed as the quotient between the number of dynamic instructions and the maximum completion time of any instruction.

3.2.1.2. IPC for a Limited Window Machine

A limited instruction window of size W implies that a given instruction cannot start execution until the instruction that is W locations above in the dynamic stream is complete. The process of computing the IPC for this scenario is an extension of the unlimited window approach. The extension consists of computing the graduation time of each instruction as the maximum completion time of any previous instruction, including itself. The completion time of a given instruction is then computed as the maximum among the completion time of all the producers of its inputs and the graduation time of the instruction W locations above in the trace, plus the latency of the instruction. Note that only the graduation time of the latest W instructions must be tracked.

3.2.1.3. Experimental Framework

The compiled programs have been instrumented with the Atom tool [117] and their dynamic trace has been processed to obtain the IPC for each configuration. Results are shown for individual programs and, in some cases, we show the average for integer programs, FP programs or the whole set of benchmarks. Average speed-ups have been computed by harmonic means and average percentages have been determined by arithmetic means [56].

The benchmark programs are a subset of the SPEC95 benchmark suite, which comprises both integer and FP codes: *compress, gcc, go, ijpeg, li, perl* and *vortex* from the integer suite, and *applu, apsi, fpppp, hydro2d, su2cor, tomcatv* and *turb3d* from the FP suite.

The programs have been compiled with the DEC C and Fortran compilers with full optimization ("-non_shared -O5 -tune ev5 -migrate -ifo" for C codes and "`-non_shared -O5 -tune ev5`" for Fortran codes). Each program was run for 50 million instructions after skipping the first 25 millions. To study the maximum degree of reusability we need to store a huge amount of data, which limits the number of instructions that can be analyzed. In this way, the results provides a taste of the overall behaviour of the SPEC95 suite (see Section 1.4 for further details of tools and benchmarks).

3.2.2. Limits of Instruction Level Reusability

This section measures the potentiality of data value reuse at instruction level. Here we therefore consider the maximum instruction-level reuse that can be exploited. This value can be computed by counting, in the dynamic instruction stream, the percentage of instructions that have previously appeared with the same input operands. This is achieved by having a table that stores all the input values of its previously executed instances for each static instruction.The maximum percentage of reusable instructions will be referred to as the *instruction-level reusability* of a program. Notice that the reusability of a program takes into account any kind of instructions, including memory accesses.

Figure 3.2 shows that instruction-level reusability is very high. For most programs it is over 90% of all dynamic instructions and on average it is 87%. Reusability ranges from 53% to 99% and *applu* and *hydro2d* are the programs with the lowest and highest reusability respectively. This figure also shows that there are no huge differences between integer and FP codes (91% and 83% of instruction-level reusability, respectively). We can conclude that instruction-level reuse is abundant in all types of programs and therefore deserves further research.

3.2.3. Performance Improvement of Instruction Level Reuse

To analyse the performance of instruction-level reuse, we consider a reuse engine with infinite tables to keep a history of previous instructions and study the effect of several reuse latencies. Reuse latency corresponds to the length of time a reuse operation takes and usually involves a table look-up and some comparisons.

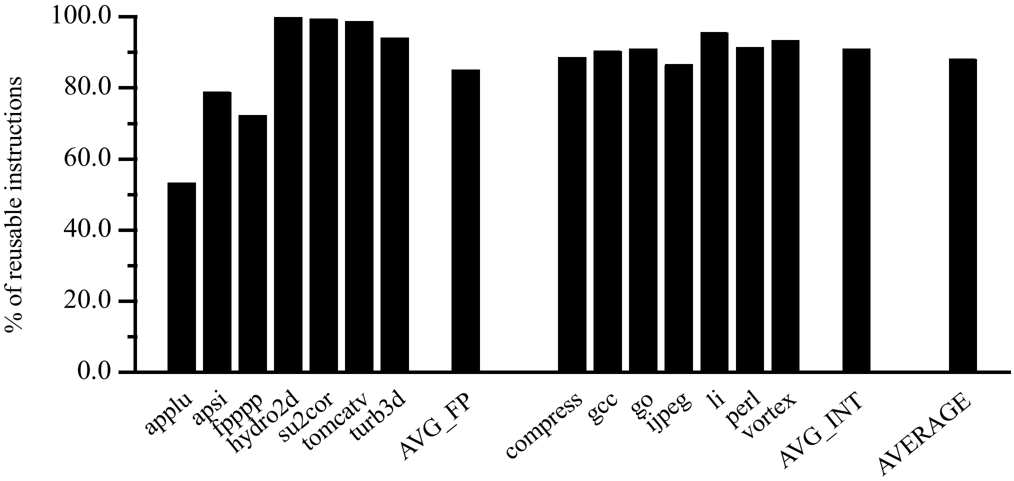


Figure 3.2. Instruction-level reusability for a perfect engine.

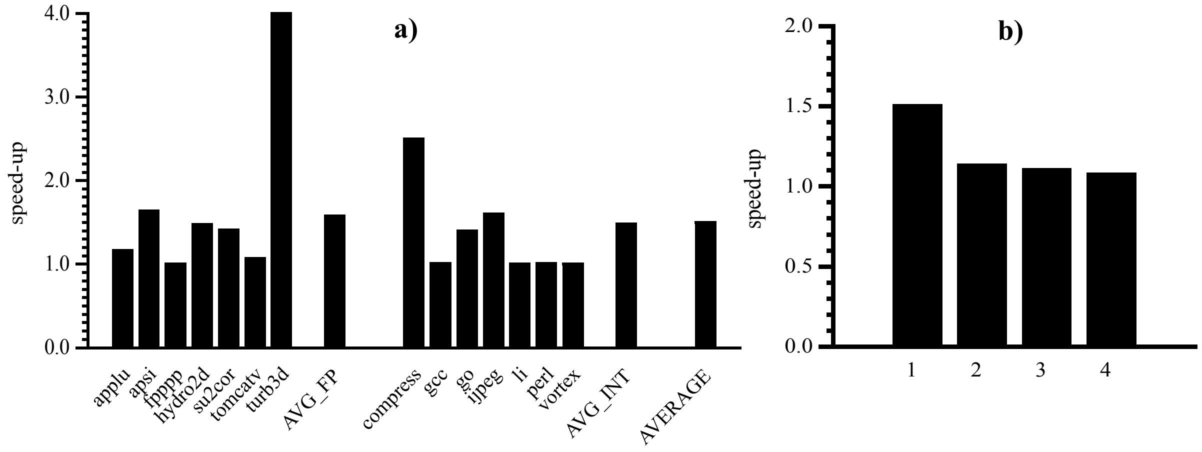


Figure 3.3. Speed-up of instruction-level reuse for an infinite instruction window: (a) for a 1-cycle reuse latency, (b) average speed-up for a reuse latency varying from 1 to 4 cycles.

Ultimately, we are interested in the effect of instruction-level reuse on execution time. For this scenario, the IPC is computed by extending the mechanisms described for an unlimited and a limited window configuration, respectively. The completion time of a non-reusable instruction is computed as in the base machine, whereas the completion time of a reusable instruction is computed as the maximum of the completion time of all the producers of its inputs (an instruction cannot be reused until all its inputs are available) plus a *reuse latency*. In any case, if the completion time of a reused instruction is higher than the completion time of the normal execution of that instruction, the latter will be chosen. This is equivalent to assuming that an oracle determines the best approach for each instruction.

Figure 3.3.a shows the speed-up provided by instruction-level reuse when the reuse latency is assumed to be 1 cycle. Note that the speed-ups depend strongly on the particular benchmark. On average, it is around 1.50 and is slightly higher for FP than for integer programs. However, some programs can benefit significantly from instruction-level reuse, e.g. as *turb3d* and *compress*, which show a speed-up of 4.00 and 2.50, respectively. On the other hand some programs, e.g. *fpppp* and *gcc,* hardly benefit from instruction-level reuse at all. In general, this performance may seem low if one takes into account the very high percentage of reusable instructions (see Figure 3.2).

Figure 3.3.b shows how, for a latency ranging from 1 to 4 cycles per reuse (only averages are shown), reuse latency affects performance. Note that the benefits of instruction-level reuse decrease significantly when more than a 1-cycle latency is assumed. This indicates that the instructions in the critical path are usually of low latency so the latency reduction achieved by instruction reuse is effective only if the reuse latency is very low. For a configuration with a limited number of functional units, the benefits will be slightly higher due to the reduction in functional unit contention. However, as we pointed out above, adding more functional units is a more cost-effective way of reducing contention than including a reuse scheme, which is significantly more complex.

The effect of instruction-level reuse does not depend only on the percentage of reusable instructions: it also strongly depends on the criticality of these instructions. In other words, if reusable instructions are concentrated on the critical path of the program, their benefit can be very high, but if they are located in the less critical sections of the program, their benefit can be negligible. Note that if reusable instructions were uniformly distributed over the whole dynamic instruction stream, independently of the criticality of the instructions and their latency, the theoretical speed-up achieved by instruction reuse would be given by 100/100-r, where *r* is the percentage of reusable instructions. If reusable instructions tend to be highly critical, this theoretical speed-up will be much lower than the actual one. On the other hand, if reusable instructions are not critical, the theoretical speed-up will be much higher than the actual one.

Figure 3.4 compares the theoretical speed-up with the actual speed-up when the reuse latency is assumed to be null. We can see that on average, the actual speed-up is slightly higher than the theoretical one (18.7 compared to 13.9), which suggests that reuse is slightly more frequent for critical instructions than for non-critical ones. The same trend is observed for integer and FP programs. Analyzing individual programs, for *fpppp, turb3d, ijpeg* and *li,* reusable instructions tend to be highly

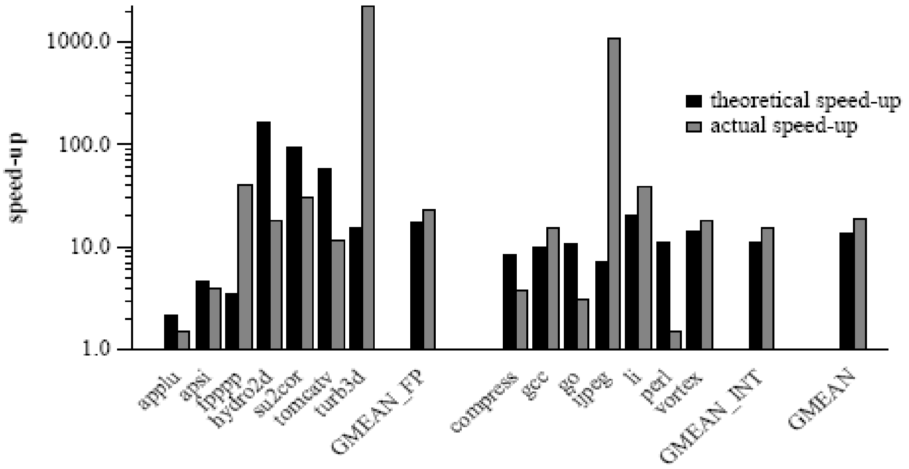


Figure 3.4. Theoretical versus actual speed-up.

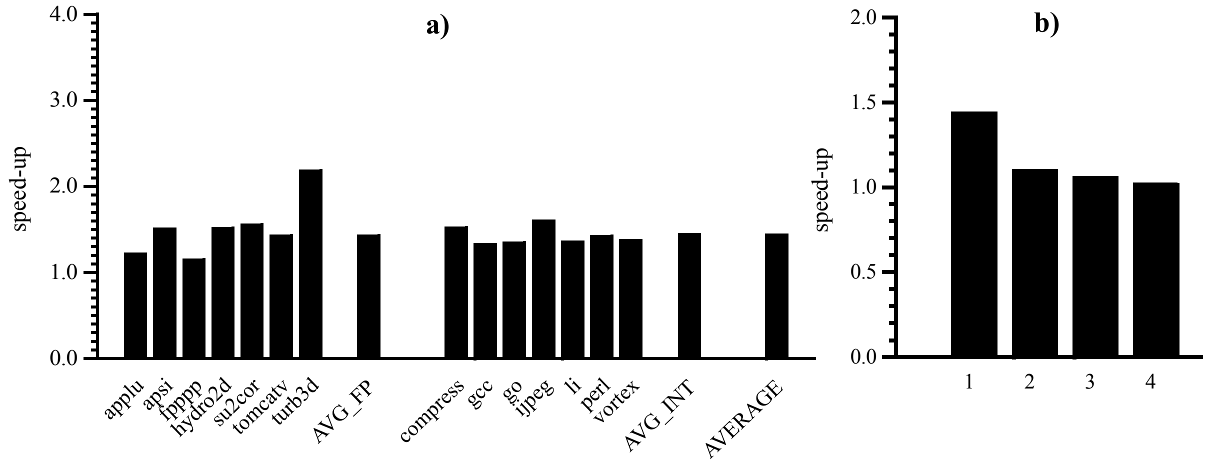


Figure 3.5. Speed-up of instruction-level reuse for a limited instruction window (256 entries): (a) for 1-cycle reuse latency, (b) average speed-up for a reuse latency varying from 1 to 4 cycles.

critical, whereas for *hydro2d*, *su2cor, tomcatv, compress, go* and *perl,* reusability tends to concentrate on non-critical instructions. For the other programs, *applu, apsi, gcc* and *vortex*, reusability and criticality seem to be quite independent phenomena.

Instruction-level reuse in the case of a limited instruction window behaves almost in the same way as in the case of an unlimited instruction window. This is shown in Figure 3.5.a, where we can see the speed-up for a 1-cycle reuse latency. The speed-up is 1.43, with minor difference between integer and FP averages (1.44 and 1.42, respectively). Differences between individual programs are smaller than for an infinite window. The benefits for programs with the highest speed-ups for an unlimited instruction window (*turb3d* and *compress*) are now reduced. Finally, Figure 3.5.b shows that, as in the infinite window configuration (see Figure 3.3.b), the benefits of instruction-level reuse when the reuse latency ranges from 1 to 4 cycles are also significantly reduced.

To summarize, the benefits of instruction-level reuse are significant for a 1-cycle reuse latency and very low for higher latencies, despite the fact that the percentage of reusable instructions is very high. This is because instruction reuse cannot be exploited until the source operands are ready, so the reuse of a chain of dependent instructions is still a sequential process.

3.3. Redundant Computation Buffer

In this section we present a mechanism for exploiting instruction-level reuse based on a buffer that stores information about which computations are likely to be redundant. This buffer is referred to as the *Redundant Computation Buffer (RCB)*. The main advantage of this proposal is that it exploits computation repetition to eliminate redundant computation from both quasi-invariants (repetitive behaviour of the same static instruction) and quasi-common subexpressions (repetitive behaviour of several static instructions).

3.3.1. General Description

The *Redundant Computation Buffer* is shown in Figure 3.6. It consists of three tables: one reuses arithmetic instruction results and memory addresses (Atable), one reuses load values (Mtable), and one identifies quasi-common subexpressions (Vtable). The Atable is indexed by the instruction address and each entry contains the following fields:

- The opcode of the instruction.
- Two operand values (opnd1 and opnd2).
- A result value, which corresponds either to the result of the latest arithmetic instruction or to the address of the latest memory operation that was mapped onto that entry.

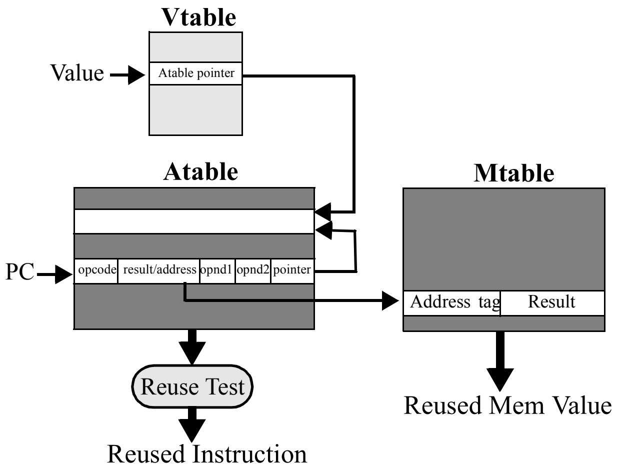


Figure 3.6. The Redundant Computation Buffer (RCB).

- A pointer to the entry that stores the results of the instruction that produces values that may be reused by the instruction mapped onto this entry; in other words, a pointer to the producer according to the terminology introduced.

Note that entries do not include a tag. An instruction can reuse the result of a given entry provided that the actual source operands and the opcode match those in the entry, no matter which static instruction the entry corresponds to. Therefore, the Atable does not include the PC tag but it does include the opcode, which is shorter than the tag. This not only reduces the table size but also takes advantage of interferences among instructions with the same opcode. Interfering instructions with different opcodes will not be able to exploit reuse, so they will have a negative effect (these interferences would also occur if PC tags were used instead of opcodes). We found that, although this optimization slightly improves the percentage of reused instructions (since there are more destructive interferences than constructive interferences) it comes at no cost. In fact, it actually reduces the cost of the Atable because the tags do not have to be stored.

The Mtable is indexed by the effective address of memory operations (obtained from the Atable), and each entry contains the latest value read from / written to that location plus a tag that identifies the effective address. In fact, this table acts in a similar way to a cache memory and is not an essential part of the mechanism. The main advantage of such a table rather than obtaining the same data from the memory hierarchy is its potential shorter access time (due to its small capacity) and a reduction in cache ports pressure.

The Vtable stores information about the latest results of instructions. For each recently produced result, this table provides an identifier of the producer instruction. The table is indexed by the result value and, in a first approximation, each entry would contain the opcode and the address of the instruction that produced that value. However, we have observed by experimental evaluation that just by storing the index of the Atable where the instruction that produces the value is mapped, performance is similar and a significant saving in storage is achieved.

The RCB works as follows. The Atable is accessed twice while the instruction is fetched and decoded. The first probe uses the instruction address and gets the potential result if the instruction happens to produce the same result as the last execution of the same static instruction (or another static instruction that is mapped onto the same entry). It also gets the pointer to the producer instruction in case it may reuse from another static instruction. The latest operands and result from that producer are obtained by accessing the table again. Once the instruction has been decoded and the source operands have been read, the actual source operands are compared with both the latest operands of the same static instruction and the latest operands and the opcode of the producer instruction. If any of the entries match, the instruction bypasses the issue and execute stages and goes directly to the write-back stage.

Load instructions may reuse the effective address using the Atable. They may also reuse their load values by means of the Mtable. This table is accessed by load instructions that have managed to reuse their addresses and is indexed by the reused address. Moreover, the conventional hardware disambiguation mechanism is used to enforce memory dependences. So, if there is a previous uncommitted store to the same address, the load does not access the Mtable. On the other hand, store instructions can also reuse their effective addresses by means of the Atable. Finally, loads update the Mtable when they are executed and stores do so when they are committed.

The Vtable, which is indexed by the result value, is updated by every instruction in the commit stage. Before writing to this table, every instruction first reads the corresponding entry and checks whether this entry was previously updated by another instruction with the same opcode. If this is the case, a pointer from the consumer (current instruction) to the producer (instruction found in the Vtable) is set in the Atable. Note that different approaches to assign confidence to the reuse among different static instruction may be devised. For instance, we could use saturating counters to establish a link from a producer to a consumer after a number of consecutive reuses.

3.3.2. A Working Example

In this section we illustrate how the mechanism achieves the reuse of quasi-common subexpressions.

To do so, let us assume the code in Figure 3.7. Let *I1* and *I2* denote the instructions that compute the values of *r* and *x*, respectively, which happen to be quasi-common subexpressions, and let us

```
while (cond) {

   r = s/t;      (I1)

   ...

   x= s/u;}      (I2)
```

Figure 3.7. A working example: code.

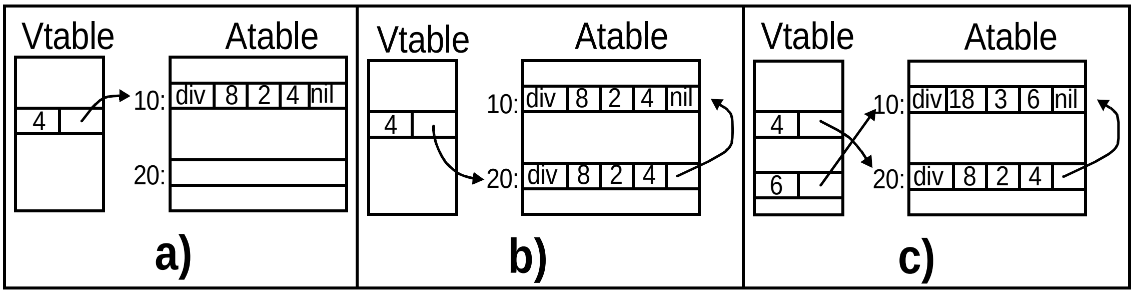


Figure 3.8. A working example: execution
a) after first execution of *I1*, b) after first execution of *I2*, and c) after second execution of *I1*.

assume that their PC is 10 and 20, respectively. The first time that *I1* is executed, the values of *s* and *t* are 8 and 2, respectively, so the result of the division is 4. The Atable entry indexed by 10 (PC of *I1*) is updated to reflect this operation and the Vtable entry indexed by the result is made to point to this Atable entry, (see Figure 3.8.a).

When *I2* is executed for the first time, it cannot be reused because the Atable entry indexed by 20 (PC of *I2*) is empty. Its source operands are also 8 and 2. Once the instruction is completed, the Atable is updated as explained for *I1*. Moreover, the Vtable is indexed by the result (4) and, since it is found in the table, the previous instruction that produced the same result (*I1*) is obtained following the pointer. At this point, a link from *I2* to *I1* in the Atable is established just by copying the pointer found in the Vtable into the Atable entry corresponding to *I2* (see Figure 3.8.b). This link indicates that *I2* has produced the same result as *I1*, so *I2* is a candidate for reusing results from *I1* in the future.

The next time that *I1* is executed, its source operands are 18 and 3. Since they are not the same as in the last execution, *I1* cannot be reused. When the instruction is completed, entry 10 in the Atable is updated to reflect the new source operands and the new result, and the Vtable is also updated to indicate the latest producer of the value 6 (see Figure 3.8.c). When instruction *I2* is encountered again, its source operands are equal to the last operands of *I1* (i.e. 18 and 3). In the decode stage, the corresponding Atable entry is looked up to check whether it can reuse the result of its previous execution. This is not so. Since a link to instruction *I1* is found, whether it can reuse the result of *I1* is also checked. This is so because current source operands of *I2* match the last source operands of *I1*. The Atable entry corresponding to *I2* is then updated with the current source and destination operands (not shown in Figure 3.8).

```
for (i=0; i<N; i++){
    ...
    x = a[i]+b[i];(S1)
    y = a[i-2]+b[i-2];(S2)
    ... }
```

Figure 3.9. Code example of reusing from non-latest result.

This process is repeated for following executions of *I1* and *I2*. Every execution of *I1* updates its Atable entry, whereas every execution of *I2* reuses the value it finds in the entry corresponding to *I1*. Note also that reuse can be exploited no matter how complex the quasi-common subexpression is. For complex subexpressions, reuse will occur for the instruction that computes the final result, as well as those instructions that compute intermediate values.

3.3.3. Reusing From Non-Latest Results

Note that the previous scheme allows the processor to reuse the result of the latest execution of instructions. However, in some cases, instructions can reuse results from previous executions of either the same or another static instruction. For instance, in the code of Figure 3.9, the result of statement S2 is the same as statement S1 of two iterations earlier (assuming that arrays *a* and *b* are not modified in the loop).

This can be exploited by storing the last *N* results for each instruction instead of the latest one. We will refer to *N* as the *history depth*. This can be implemented using a N-way set-associative buffer, as proposed by Sodani and Sohi [112].

3.3.4. Enhanced Result Cache and Enhanced Result Buffer

In Section 3.3.7, we will compare the RCB mechanism with the two main proposals of instruction-level reuse for superscalar processors: the *Result Cache* [91] and the *Reuse Buffer* [112]. These schemes are enhanced with several features of the RCB to provide a fair comparison. In this section we outline the main characteristics of these schemes and the enhancements applied.

The *Result Cache* and the *Result Buffer* preserve the results of instructions in hardware tables together with some information needed to establish their validity at a later time. When an instruction is fetched and its current inputs are found in that table, its execution can be avoided by obtaining the output from the table.

An important difference between the *Result Cache* and the *Reuse Buffer* is that the former requires the source operands to index the buffer, whereas the latter is indexed by the instruction address and its operands are only required once the entry is read, so that it can be compared with the operands in the entry. In other words, the *Reuse Buffer* can overlap the buffer lookup with the instruction fetch, register rename and register read operations, whereas the *Result Cache* cannot. So, if we assume that a table lookup takes one cycle, the *Reuse Buffer* mechanism can produce the result of a reusable instruction one cycle earlier than the *Result Cache* (see Section 3.3.5 for a detailed discussion of timing considerations). On the other hand, the main drawback of the *Reuse Buffer* is that it can only reuse dynamic instances of the same static instruction. In other words, the *Reuse Buffer* seeks to exploit quasi-invariants but does not take advantage of quasi-common subexpressions.

Moreover, the *Reuse Buffer* can reuse multiple inter-dependent instructions fetched simultaneously. Since the *Reuse Buffer* is indexed by the instruction address, the corresponding multiple entries can be read simultaneously and the reused result of an instruction can be used to check whether the following dependent instructions can be reused, just like register renaming of multiple dependent instructions can take place in the same cycle. We refer to this feature as *reuse chaining.* On the other hand, reuse chaining cannot be exploited by the *Result Cache*, since the buffer lookups would be sequential and could not be performed in a single cycle. This is because the *Result Cache* uses the source operand values to index the buffer, so if instruction B depends on instruction A, the buffer lookup of instruction B cannot be performed until the result of instruction A is available (i.e. until the reuse of A finishes and produces the source operand of B).

As explained earlier, unlike the *Reuse Buffer*, the Atable does not include the PC tag but it does include the opcode, which is shorter than the tag. This not only reduces the table size but also takes advantage of interferences between instructions with the same opcode and comes at no cost. When comparing the RCB and the *Reuse Buffer*, we will consider an enhanced *Reuse Buffer* that also includes this optimization.

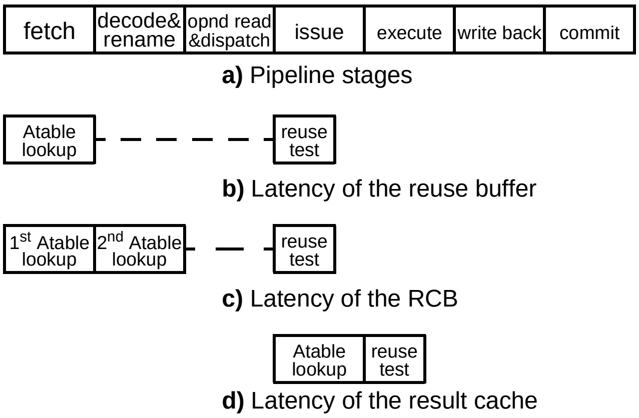
The RCB we propose seeks to eliminate redundant computation from both quasi-invariants and quasi-common subexpressions. Moreover, it has the same reuse latency as the *Reuse Buffer*, since it is indexed by the instruction address and can also exploit reuse chaining.

The *Result Cache* has been enhanced with a table to store memory values that has the same structure as the Mtable of the RCB scheme. We name this new *Result Cache* the *Enhanced Result Cache* (ERC). Also, in the original design, each index bit was obtained by X-oring one bit from the operand 1, one bit from the operand 2 and another bit from the opcode. We have evaluated several X-oring schemes and will present results for the most effective one. However, an interesting area for further research may be to analyse more sophisticated hashing schemes. Finally, as proposed in the original design, reuse test is not limited to long latency operations.

The *Enhanced Reuse Buffer* (ERB) consists of two tables. One table is targeted to reuse arithmetic instructions, conditional branches and memory addresses (Atable) and the other is targeted to reuse memory values (Mtable). In the original proposal [112], these two tables were merged into a single one but the authors suggested that a split implementation could be more cost-effective. However, they suggested a separate table for memory instructions that was indexed by the instruction address. That implementation needed store instructions to associatively search the table for a matching memory address, something which is avoided by the Mtable. Moreover, like the RCB, the instruction address tag is replaced by the opcode. As we discussed in Section 3.3.1, this take advantages of constructive interference and reduces the storage required.

3.3.5. Timing

As we indicated above, the performance of a reuse scheme is determined not only by the percentage of reused instructions but also by the reuse latency. Let us assume a dynamically scheduled processor with a microarchitecture that keeps speculative results in the reorder buffer or rename buffers and is pipelined in the stages shown in Figure 3.10.a (this example is based on the PowerPC 604 [116] but the conclusions are the same for other pipelines). Every instruction is fetched and then decoded and the physical location that holds the last definition of each source operand (if available) is identified. Available operands are read and the instruction dispatched to a reservation station. When all the operands are ready and a functional unit is available, the instruction is issued. It is then executed and the result is written back. Finally, instructions commit in order.



a) Pipeline stages

b) Latency of the reuse buffer

c) Latency of the RCB

d) Latency of the result cache

Figure 3.10. Reuse latency of the reuse buffer, the RCB and the result cache.

Since it is indexed by the PC, the *Reuse Buffer* (see Figure 3.10.b) accesses the Atable in parallel with the Icache lookup. However, the reuse is not effective until the reuse test is performed. This test cannot be done until the source operands are read. Since the reuse test only involves equality comparisons, it is reasonable to assume that it takes less than a cycle.

The RCB (see Figure 3.10.c) accesses the Atable in the fetch stage by indexing it through the PC. As a result of this access, it obtains a pointer to a potential producer of the same result (a single link is provided no matter which history depth is implemented). In the next cycle, it accesses the Atable entry corresponding to this pointer. The reuse test is performed once the operands are read, so the total reuse latency is the same as that of the *Reuse Buffer*.

The *Result Cache* (see Figure 3.10.d) indexes the Atable by means of the operand values, so it cannot perform the access until the operands are read. This implies that the latency of this mechanism is one cycle longer than that of the *Reuse Buffer* and the RCB.

For all reuse schemes, once the reuse test is successful, the instruction bypasses the execute stage and goes directly to the write-back stage. Note also that the *Result Cache* can exploit reuse from both quasi-invariants and quasi-common subexpressions. Moreover, it can exploit reuse from non-latest results. On the other hand, the *Reuse Buffer* can only exploit quasi-invariants. However, since it is indexed by the instruction address, its reuse latency is shorter than that of the *Result Cache*. The RCB has the best of both worlds: it exploits both quasi-invariants and quasi-common subexpressions and has the same reuse latency as the *Reuse Buffer*.

3.3.6. Hybrid Redundant Computation Buffer

A couple of issues may limit the performance potential of the instruction reuse schemes: the indexing method of the tables and the reuse latency. We have seen that the *Result Cache* indexing method provides a greater percentage of reuse but that its higher reuse latency means that its performance potential is limited. In this section we propose a slight modification of the RCB scheme that could take advantage of the best of both approaches. We therefore propose the *Hybrid Redundant Computation Buffer*. This new scheme stores the result of each instruction in two different entries. One entry is indexed by the instruction address, (in the same way as the RCB), and the other is indexed by hashing the source operand values and the opcode (in the same way as the *Result Cache)*. Note that the downside of this approach is that, as each dynamic instruction can use two entries of the buffer, it may produce more interference.

When instructions are fetched, they are tried to be reused first by indexing the Atable through the instruction address. If this is not successful, they are then tried to be reused by indexing the Atable again through the hashed operand values and opcode. The first probe provides low latency and the second provides higher reusability.

3.3.7. Performance Evaluation

This section evaluates the performance of the RCB and compares it with the performance of the *Enhanced Result Cache* and the *Enhanced Reuse Buffer*.

3.3.7.1. Experimental Framework

The different reuse schemes have been evaluated for a superscalar processor using the Alpha version of the Simplescalar toolset [15]. Simplescalar is an execution-driven simulator based on the Alpha ISA that models an out-of-order machine and has been modified to support data value reuse schemes. The base simulator models a 4-way dynamically scheduled superscalar processor based on the Register Update Unit [114]. The parameters of this simulator are shown in Table 1.2.

N-way set associative buffers, where N is the history depth, are considered for all the reuse mechanisms. The performance of each scheme is drawn in front of its required storage capacity, which is measured as the total number of bits required to implement it, including tags when used.

The following Spec95 benchmarks have been considered: *compress, go, gcc, li, m88ksim, perl* and *vortex* from the integer suite; and *applu, mgrid, swim* and *turb3d* from the FP suite. The programs have been compiled with the DEC C and F77 compilers with `-non_shared -O5` optimization flags (i.e. maximum optimization level). Each program was run with the test input set and statistics were collected for the first 125 million instructions after skipping the initial part corresponding to initializations (see Section 1.4 for further details of tools and benchmarks).

3.3.7.2. Basic Reuse Statistics

In this subsection we present statistics for the percentage of reuse that can be exploited by the various schemes for an ideal scenario in which all instructions are assumed to have their operands ready in the decode stage. For all figures, the percentage of reuse is shown over all the dynamic instructions, where arithmetic operations, conditional branches, memory addresses and load values are the instructions capable of being reused. Note that memory operations count twice (once for the address calculation and once for the memory value). The aims of this study are to determine the best configuration for each reuse scheme and then evaluate these best configurations for a superscalar processor in Section 3.3.7.3. In this latter scenario, the unavailability of the source operands will prevent the exploitation of reuse.

We first show that exploiting quasi-common subexpressions is a significant source of reuse. This is illustrated in Figure 3.11, which shows the percentage of dynamic instructions reused when the history depth is one and the Atable of each scheme has 1024 entries. The Mtable and the Vtable are assumed to have 512 and 1024 entries respectively, in all the experiments. The total size of the tables for the

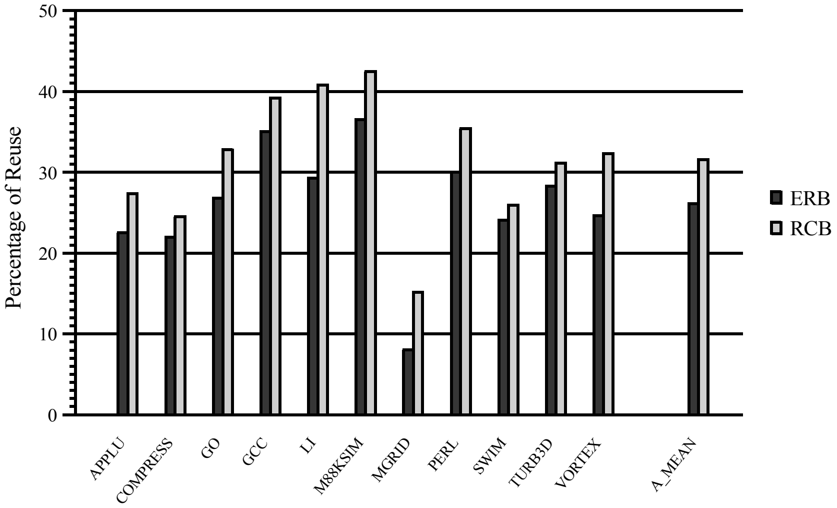


Figure 3.11. Enhanced Reuse Buffer vs. Redundant Computation Buffer for history depth of 1.

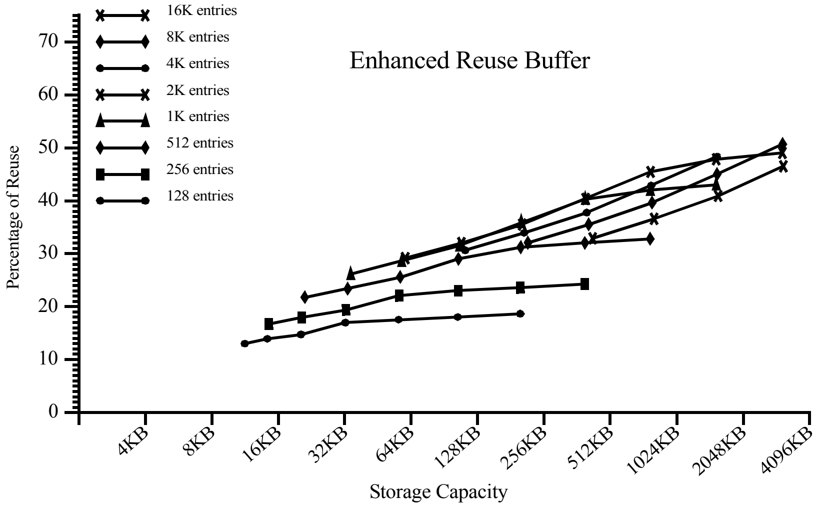


Figure 3.12. Reuse exploited by the Enhanced Reuse Buffer.

experiment in Figure 3.11 is approximately the same: 34 KB (KiloBytes) and 36 KB for the Enhanced *Reuse Buffer* and the *Redundant Computation Buffer*, respectively. Note that both schemes can reuse a significant number of instructions and that RCB numbers are always higher than ERB numbers. On average, the reuse due to quasi-common subexpressions is about 6% and is significant for all programs.

Figure 3.12 shows the percentage of dynamic instructions reused by the *Enhanced Reuse Buffer*. Only average numbers (arithmetic mean) for the set of benchmarks are shown. The X-axis shows the total storage capacity in KiloBytes (KB). Each line corresponds to a different number of entries in the Atable (the size of the Mtable is kept fixed and is also considered in the total capacity). The different dots in a line correspond to different history depths, starting from 1 at the leftmost point and increasing by a factor of two from one point to the other.

The results for 128 and 1024 entries with history depth of one are consistent with those of Sodani and Sohi [112]. For example, the 1024-entry configuration can reuse 26% of the dynamic instructions (Sodani and Sohi [112] reported 25.7%). Increasing the history depth provides a significant improvement, as is implied by the positive slope of the curves. For example, for the 1024-entry configuration, reuse grows from 26% to 41% when the history depth increases from 1 to 16. Note also that the benefits of increasing the history depth are more noticeable for larger numbers of entries. If we look at the best trade-off between number of entries and history depth, we can conclude that a history depth of one is the most effective configuration for small capacities, but not for large capacities. This is because for a small capacity the additional storage is best spent in reducing interferences (by increasing

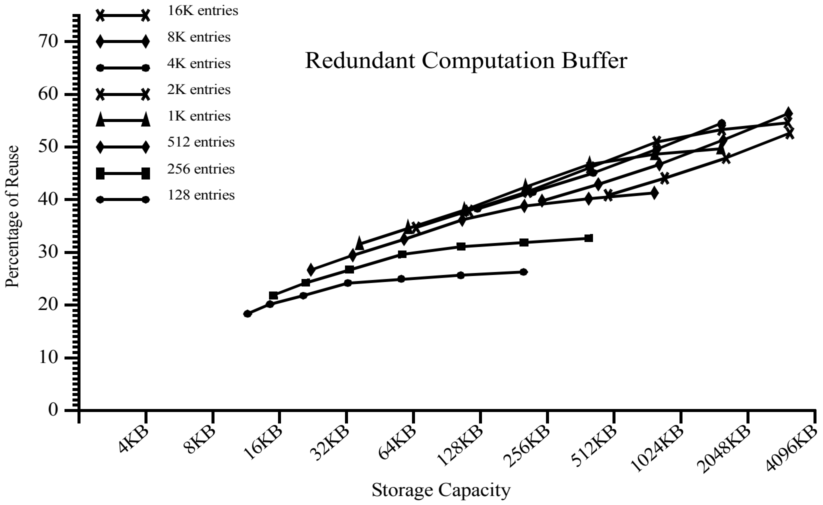


Figure 3.13. Reuse exploited by the Redundant Computation Buffer.

the number of entries), whereas interferences are very rare for large capacities. The trend is similar for the RCB in Figure 3.13 (again note that the size of the Mtable and Vtable are kept fixed and that they are considered in the total capacity).

Note that the concept of history depth does not make sense for the *Result Cache* since its entries are not associated to particular static instructions. In fact, the *Result Cache* can store multiple results of the same static instruction by placing each one in a different *Result Cache* entry.

Figure 3.14 compares the three reuse schemes in terms of percentage of reused instructions for different total storage requirements. For each scheme and capacity, only the best configuration is

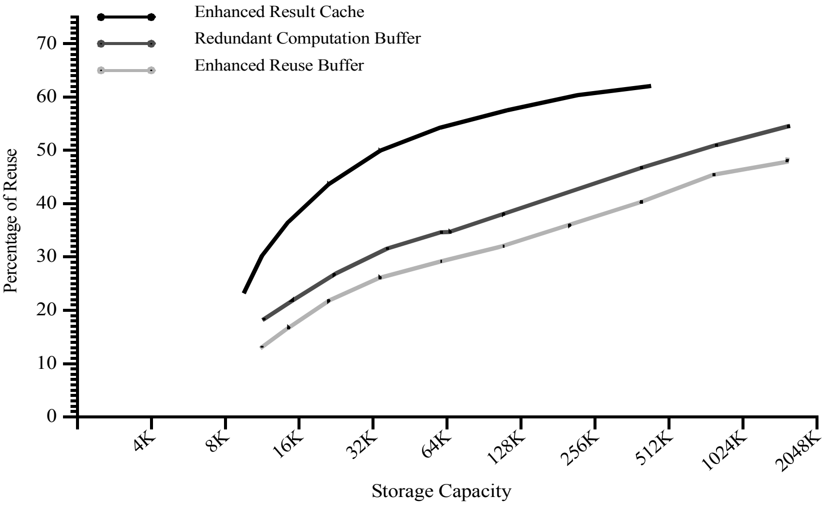


Figure 3.14. Reuse exploited by the best configurations of every scheme.

displayed. Note that in terms of reusability, the *Enhanced Result Cache* is the most effective scheme. This is followed by the *Redundant Computation Buffer* and the *Enhanced Reuse Buffer.* The effectiveness of the *Result Cache* is explained by the fact that this scheme allows a given instruction to reuse results from any other instruction with the same opcode as the RCB uses, but it does not use up any storage to store pointers and does not need to first establish pairs of producers-consumers before reusing among different static instructions. On the other hand, the *Reuse Buffer* can only reuse results from the same static instruction.

3.3.7.3. Performance Figures for a Superscalar Processor

In this subsection we present performance figures for the various reuse schemes for a superscalar processor. As we explained earlier, the base simulator models a 4-way dynamically-scheduled superscalar processor with the parameters shown in Table 1.2. Two configurations were evaluated for each reuse scheme, one for around 32KB of capacity and the other for around 200KB of capacity. For each capacity, the best configuration was chosen according to the analysis in Section 3.3.7.2. The precise values are 34KB, 36KB, 34KB and 217KB, 221KB, 217KB for the *Enhanced Reuse Buffer, Redundant Computation Buffer* and *Enhanced Result Cache,* respectively.

Figure 3.15 shows the speed-up achieved by the various reuse schemes for the base microarchitecture. The RCB scheme provides the highest speed-up and the *Result Cache* provides the lowest speed-up even though it exploits most reuse in the ideal case. However, the *Result Cache* is significantly penalized for not exploiting reuse chaining and for its higher reuse latency. We can see

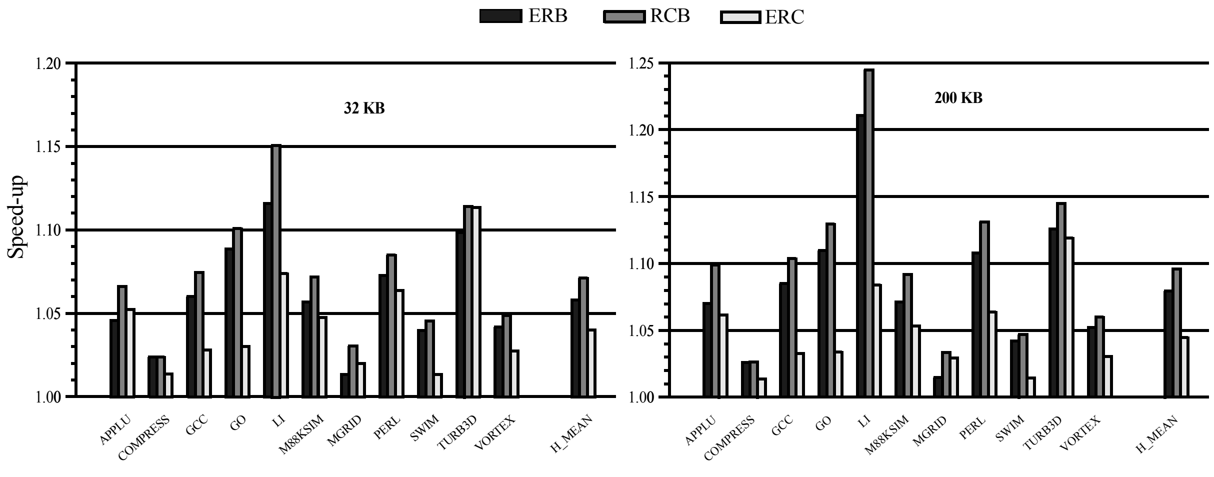


Figure 3.15. Speed-up for the base microarchitecture for each program and the harmonic mean.

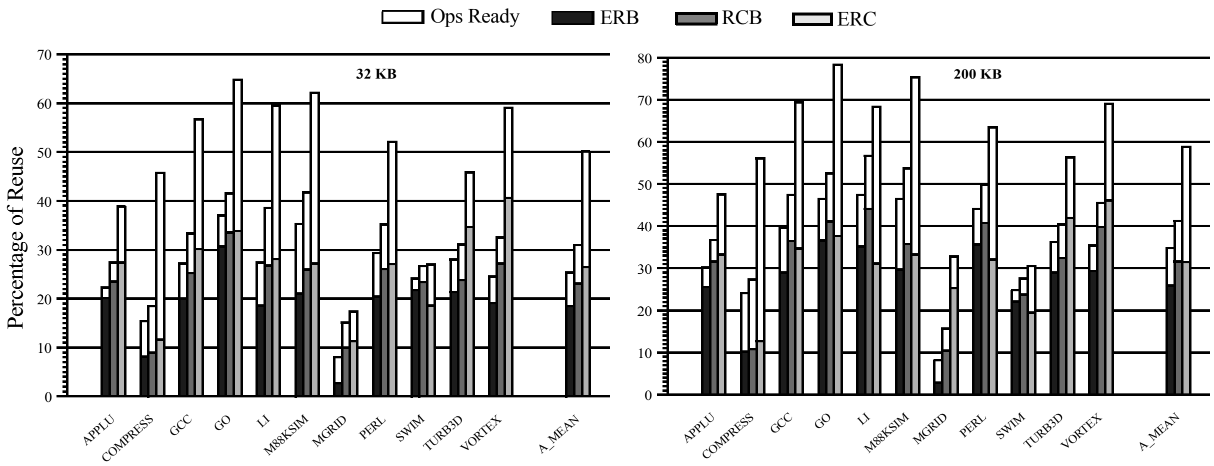


Figure 3.16. Reuse exploited in the superscalar microprocessor.

that the RCB provides significant speed-ups for all programs. The highest benefit is experienced by *li*, which shows a speed-up of 1.15 and 1.25 for the 32 KB and 200 KB configurations, respectively.

Figure 3.16 shows the reuse exploited by the various schemes in the base microprocessor. It also shows reuse exploited in the ideal case in which all the operands are ready. The actual reuse is lower than the ideal reuse despite including some squash reuse [112] (reuse from squashed instructions due to a control misspeculation), which the ideal case does not. The *Result Cache* loses most reuse in relation to the ideal case. An important reason for this is its inability to exploit reuse chaining. The *Result Cache* is therefore still the scheme that exploits the highest amount of reuse but the difference with the RCB is much smaller than in the ideal scenario.

We performed the same experiments for a more aggressive microarchitecture. In these experiments we have assumed an 8-way issue superscalar processor with twice as many functional units as our base microarchitecture and a more aggressive branch predictor based on an SAg [134] with 4096 branch history registers and a 4096-entry pattern history table. The averages are quite similar to those obtained for the base architecture, though there are some differences in individual programs.

Figure 3.17 shows the contribution of each instruction category to the percentage reuse shown in Figure 3.16. We can see that the contribution of each category to total instruction reuse is similar and that there are no significant differences when we compare the various instruction reuse schemes.

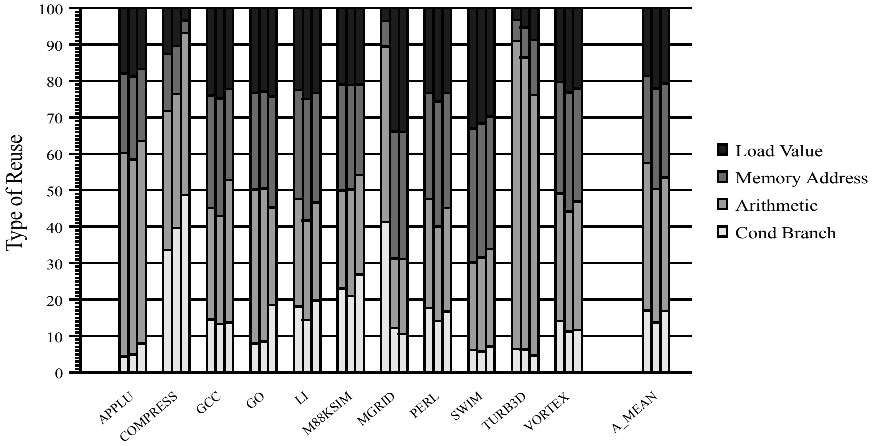


Figure 3.17. Contribution of each instruction category to total reuse. Each group of bars corresponds to ERB, RCB and ERC schemes respectively

3.3.7.4. Performance of the Hybrid RCB Scheme

The previous section shows that the *Result Cache* outperforms the RCB in terms of exploited reuse but, because of its higher reuse latency, provides a lower speed-up. The hybrid RCB scheme could take advantage of the best of both approaches. As we explained before, this scheme stores the result of each instruction in two different entries. One of the entries is indexed by the instruction address, as with the RCB, and the other is indexed by hashing the source operand values and the opcode. The first entry provides low latency while the second entry provides higher reusability. Figure 3.18 compares the RCB and the hybrid RCB schemes for a 32 KB capacity (the results for 200 KB are similar). Note that the hybrid scheme provides a slightly higher speed-up and increases the percentage of reused instructions by about 7% (from 23% to 30%).

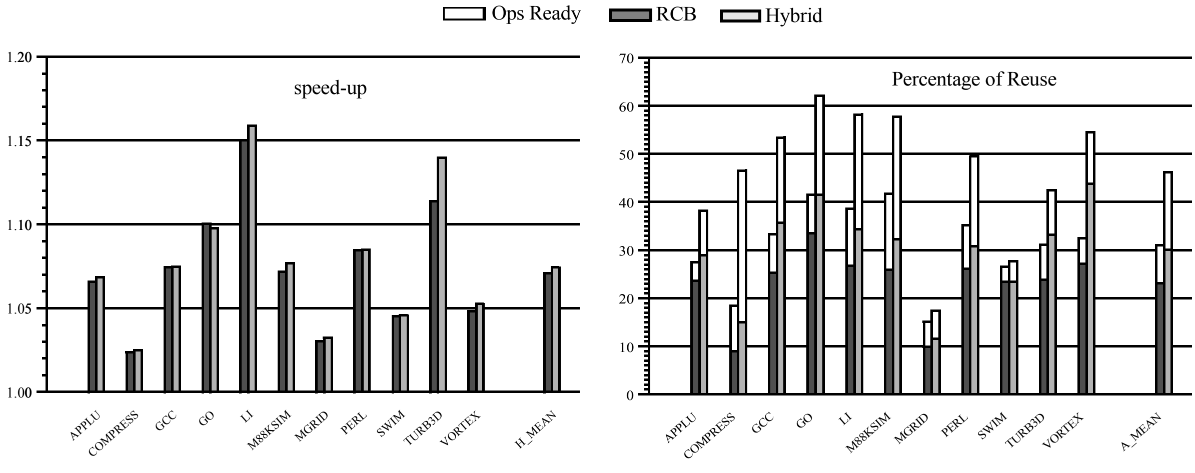


Figure 3.18. Performance of the RCB and the hybrid RCB scheme for a 32 KB capacity.

3.4. Related Work

Numerous data value reuse mechanisms based on both software and hardware approaches have been proposed. Software implementation is usually known as *memoization* or *tabulation.* Memoization [1],[11],[75] was proposed as a code transformation technique to take advantage of the redundant nature of computation by trading execution time for increased memory storage. The main idea is to store the result of functions together with their input values in a table that exists as a software data structure. This technique can also be extended to statements, groups of statements, or any given region that has limited side effects and a high degree of recurrence. Later invocations of these sections of code are preceded by a table look-up and, in case of hit, their executions are avoided.

A hardware implementation of data value reuse was first proposed by Harbison [46] for the Tree Machine. The Tree Machine has a stack-oriented ISA and the main novelty of its architecture was that the hardware assumed several compiler's traditional optimizations, such as common subexpression elimination and invariant removal. This is achieved using of a *Value Cache*, which stores the results of dynamic sequences of code (called *phrases*) together with information about the variable on which each sequence depends. New executions of the same phrases can be reused provided that none of the inputs variables have been written since the previous execution.

Another hardware implementation of data value reuse was suggested by Richardson [91],[92]. This author proposed a *Result Cache*, that seeks to speed-up some long latency floating point operations, such as multiplications and divisions. It consists of a buffer that is indexed by hashing the source operand values, and contains the last operation applied to such values and its result. Result caching has been further investigated by Oberman and Flynn [83], who proposed a specific buffer for each type of long-latency operations: *Division Caches, Square Root Caches* and *Reciprocal Caches.* These authors also investigated a shared cache for reciprocals and square roots. Citron *et al* [24] extended the *Result Cache* approach to multimedia applications. This approach is suited to bypassing the execution of multi-cycle computations (multiplication, division, square root) in a single cycle by means of distributed and specialized value reuse tables that are accessed in parallel with the functional units. Finally, result caching has also been evaluated by Azam *et al* [7] as a way to reduce power consumption.

Sodani and Sohi [112] invented the concept of dynamic instruction reuse and proposed the *Reuse Buffer*, which is indexed by the instruction address. They presented three schemes. In the first scheme, each entry of the buffer contains the source operands and the result of the last dynamic instance of the corresponding instruction. In the second scheme, each entry contains the source register identifiers and the result. In the third scheme, each entry contains pointers to the instructions that compute the source operands in addition to the fields of the second scheme. Experimental results show that the first scheme provides the highest speed-up for large reuse buffer sizes. A set-associative implementation of the *Reuse Buffer* allows it to store multiple dynamic instances of the same static instruction. Huang and Lilja [48] extended the first scheme of the *Reuse Buffer* by modifying the indexing mechanism. The idea is to index the contents of the table using a hash function of the values of the input operands of an instruction instead of indexing by the address of the instruction. Citron *et al* [26] revised the concept of instruction-level reuse by repeating and widening the scope of previous proposals, such as *Memo Tables* by Citron *et al* [24], *Redundant Computation Buffer* by Molina *et al* [77], and *Reuse Buffer* by Sodani and Sohi [112]. They also extended their analysis to consider the area, energy and timing overheads of maintaining such tables [25]. A different approach to filling the reuse tables was proposed by Yi *et al* [137]. These authors presented the *Instruction Precomputation* mechanism which uses profiling to determine the unique computations with the highest frequencies of execution. The reuse table is therefore loaded with these instructions before the program executes and, except for the fact that entries are not dynamically replaced, it is managed in a traditional way.

Specialization in load and store instruction reuse has been covered by Yang and Gupta [131],[133]. These authors presented reuse techniques for load redundancy removal that eliminate redundancy across different dynamic instances of the same static instruction and eliminate redundancy across dynamic instances of statically distinct instructions [133]. They also designed a carefully tuned load and store reuse mechanism to achieve net energy savings [131].

Although instruction reuse was initially designed to avoid the execution stage of instructions, some authors have explored several ways of applying it. Weinberg and Nagle [127] proposed a mechanism to eliminate the computation of high-level language pointer expressions. Basically, once the input operand set of an expression matches a previously executed instance of the same expression, the result is obtained from a table instead of recomputing it. This mechanism requires some compiler support to mark the instructions that are involved in the expressions. Jourdan *et al.* [59] proposed a renaming scheme that exploits the phenomenon of instruction-level reuse in order to reduce the register pressure. The basic idea is that several dynamic instructions that produce the same result share the same physical register. Onder and Gupta [84] also relied on the physical register file to provide data values corresponding to a subset of memory addresses whose values are currently resident in physical registers. Prefetching and load instruction reuse were combined by Surendra *et al* [121] to study their effectiveness in reducing data cache traffic in network processor units.

Squash reuse is another special implementation of instruction-level reuse in which the reused value comes from the same instance of the instruction that have been squashed. The concept was first introduced by Sodani and Sohi [112] as a way to reduce branch miss-speculation penalties. As we explained earlier, they proposed a table-based technique for avoiding the execution of an instruction that has been previously executed with the same inputs. As well as squash reuse, they also covered general reuse. A different implementation based on a centralized window environment was proposed by Chou *et al* [23]. These authors also introduced the idea of dynamic control independence and showed how it can be detected and exploited in an out-of-order superscalar processor to reduce the branch missprediction penalty. Roth and Sohi [96] proposed register integration as a simple and efficient implementation of squash reuse. This mechanism enables speculative results to remain in the physical register file after the producer instruction is squashed. Later, the speculative results may be reused through a modified renaming scheme. Finally, Petric *et al* [86] extended the concept of register integration by adding modifications to the original design that expanded its applicability and boosted its performance impact.

Sato and Arita [101] explored instruction-level reuse to enhance their *Variable Latency Pipeline*. This structure has proved effective for mitigating the constraints on the operand bypass logic. Sato [99] also presented a method for integrating fault-tolerance techniques into microprocessors by applying instruction redundancy as well as time redundancy.

Instruction reuse has also been combined with value prediction to improve its performance potential. Sodani and Sohi [113] investigated the various ways in which value prediction and instruction reuse interact with other microarchitectural features and the impact of such interactions on net performance. Choi *et al* [22] modified the *Result Cache* to enable value prediction. In their proposal, the *Result Cache* is managed in a traditional way to reuse instructions. The novelty is that the result of a previously executed instruction can be used as a prediction if the reuse test could not be performed because the operands of the instruction were not ready at decode time. Liao and Shieh [68] explored the combination of value reuse and value prediction in the *Reuse Buffer.* Manoharan and Narayanan [73] introduced a similar approach that managed confidence bits to minimize the misspredictions.

Finally, several studies have evaluated the sources of instruction repetition. Sodani and Sohi [111] investigated the source of the repetitive behaviour of the computations by categorizing dynamic program instructions into dynamic program slices at different levels. Yi and Lilja [135] defined the terms of *local* and *global level redundant computations* to refer the repetitive behaviour produced by the same static instruction and the repetitive behaviour of several static instructions, respectively. He also analysed the potential of reuse and compared the amount of redundant computation at the global level to the amount of redundant computation at the local level. Surendra *et al* [120] examined instruction reuse in network processing applications and showed that significant instruction reuse can be exploited in such applications. These authors also proposed a flow aggregation scheme that exploits packet correlation and uses multiply *Reuse Buffers* to further enhance the utility of instruction reuse.

3.5. Conclusions

Instruction-level reuse has proved to be an effective technique for avoiding the serialization caused by data dependences. In this chapter we have analysed instruction-level reuse in detail and shown that it can benefit from computation repetition to boost the execution of instructions.

First, we analysed the performance potential of instruction-level reuse under various scenarios. The evaluation was performed for an infinite resource machine and then for a machine with a limited instruction window. We conclude that instruction-level reuse is abundant in all types of programs and can provide very large speed-ups for an ideal machine. However, the benefits of instruction-level reuse are moderate for a 1-cycle reuse latency and low for higher latencies, even though the percentage of reusable instructions is high. This is because instruction reuse cannot be exploited until the source operands are ready, and so, the reuse of a chain of dependent instructions is still a sequential process.

In this chapter we have also presented a novel reuse mechanism that we called *Redundant Computation Buffer (RCB)*. This hardware mechanism can exploit reuse due to both quasi-invariants (repetition produced by the same static instruction) and quasi-common subexpressions (repetition produced by several static instructions) and also exhibits a low reuse latency. When we compared this mechanism with previous schemes, which we extended with novel features borrowed from the RCB, it provided the greatest benefits in terms of execution time reduction, though it did not achieve the highest reusability. On average, the RCB can reuse around 30% of all dynamic instructions, which implies a 1.10 speed-up. Improvements were experienced by all the programs. These ranged from 3% for *compress* to 25% for *mgrid*. We have also shown that storing multiple dynamic instances of the same static instruction can provide significant benefits, especially for large buffers.

“Those who can not remember the past are condemned to repeat it”
George Santayana, US (Spanish-born) Philosopher,1862-1952.

Chapter 4

TRACE LEVEL REUSE

Trace-level reuse is a data value reuse technique that avoids the execution of a dynamic sequence of instructions (traces). As long as executions have the same inputs, all changes in the processor state that would be produced by these instructions are done by reapplying the changes that were produced in a past execution of the same trace. Trace-level reuse is therefore based on the phenomenon that computations performed by programs tend to be repetitive.

In this chapter we introduce the concept of trace-level reuse and analyse its performance potential. We also address essential design issues for integrating this technique into a superscalar processor.

4.1. Introduction

Data value reuse techniques have been traditionally limited to the instruction level. Instruction-level reuse techniques are special implementations of data value reuse intended to avoid the execution of single instructions that produce the same result as previous instructions.

We have observed that some traces (dynamic sequences of instructions) are frequently repeated during the execution of a program and that the instructions that make up such traces often have the same source operand values. The execution of such traces will obviously produce the same outcome, so their execution can be skipped if the processor records the outcome of previous executions. We refer to this data value reuse technique that exploits the repetition of several consecutive computations as *Trace-Level Reuse.*

Typically, computation reuse works by storing the results of a previously seen computation in a reuse table. In this case, a computation refers to a set of consecutive instructions. When the computation occurs again, the outcomes of that computation are obtained from the reuse table and the reusable instructions are bypassed.

Trace-level reuse can improve performance by decreasing resource contention and the latency of some instructions in the same manner as instruction-level reuse. However, we will show that trace-level reuse is more effective than instruction-level reuse because it can avoid fetching the instructions of reused traces. This has two important benefits: it reduces the fetch bandwidth requirements and, since these instructions do not occupy window entries, increases the effective instruction window size. Moreover, trace-level reuse can compute the result of a chain of dependent instructions all at once, which may allow the processor to avoid the serialization caused by data dependences and therefore exceed the dataflow limit.

In this chapter we introduce the concept of trace-level reuse, discuss its implementation and analyse its performance impact. Section 4.2 analyses the performance potential of trace-level reuse under several scenarios. Section 4.3 address several design issues of trace-level reuse that covers aspects such as the memory required to store traces, a method for selecting traces, a mechanism for reusing traces and the process for updating the processor state. Section 4.4 reviews related work and Section 4.5 summarizes our main conclusions. Finally, Appendix A defines a set of theorems related to trace-level reuse that support the study of the performance potential.

4.2. The Performance Potential of Trace-Level Reuse

In this section, we analyse the trace-level reuse phenomenon and investigate the performance potential of this technique. We isolate the effect of trace-level reuse from other microarchitectural aspects by putting it into an ideal machine context, with either a limited or an unlimited instruction window. In summary, we are interested in evaluating the performance limits of trace-level reuse instead of focusing on any particular implementation.

4.2.1. Evaluation Methodology

The simulation environment assumed in this section is the same as the one described in Section 3.2.1. We considered two scenarios to evaluate the performance potential of trace-level reuse: an ideal machine with either an infinite window (see Section 3.2.1.1) or a limited window (see Section 3.2.1.2). Both scenarios assume a machine with an infinite number of functional units.

The benchmark programs are a subset of the SPEC95 benchmark suite comprising both integer and FP codes: *compress, gcc, go, ijpeg, li, perl* and *vortex* from the integer suite, and *applu, apsi, fpppp, hydro2d, su2cor, tomcatv* and *turb3d* from the FP suite. The programs have been compiled with the DEC C and Fortran compilers with full optimization ("-non_shared -O5 -tune ev5 -migrate -ifo" for C codes and "`-non_shared -O5 -tune ev5`" for Fortran codes). Each program was run for 50 million instructions after skipping the first 25 millions.

The compiled programs were instrumented with the Atom tool [117] and their dynamic trace was processed in order to obtain the IPC for each configuration. Results are shown for individual programs and in some cases we show the average for integer programs, FP programs or the whole set of benchmarks. Average speed-ups have been computed through harmonic means and average percentages have been determined through arithmetic means (see Section 1.4 for further details of tools and benchmarks).

4.2.2. Limits of Trace-Level Reusability

As we indicated earlier, the reuse of traces is an attractive technique since a single reuse operation may skip the execution of a potentially long sequence of dynamic instructions even if these are inter-dependent. To evaluate the performance limits of this technique, we need to compute the maximum reuse that can be attained for any possible partition of the dynamic instruction stream into traces. Since there is no constraint for the contents of each trace, the number of ways of partitioning a dynamic instruction stream into traces is practically unlimited, which prevents an exhaustive exploration of all of them.

Given that each reuse operation has an associated latency (e.g. table lookup), the most effective schemes will be those that reuse maximum length traces. Given a dynamic instruction stream that corresponds to the execution of a program, we are interested in identifying a set of reusable traces such that: a) the total number of instructions included in those traces is maximum and b) the number of traces is minimum. In other words, if a trace is reusable, it is more effective to reuse the whole trace in a single reuse operation than to reuse parts of it separately. However, finding maximum length reusable traces would be still a complex problem if all the possible partitions of a program into traces were explored.

We can, however, prove that if we consider just those traces that are formed by all maximum-length dynamic sequences of reusable instructions, we have an upper bound of the reusability that can be exploited by maximum-length traces (condition (a) above) and a lower bound of the number of traces required to exploit it (condition (b) above). This is supported by the theorems defined below and described in Appendix A. The performance provided by assuming that such traces are reusable will provide an upper bound of the performance limits of trace reuse.

Theorem TLR1. Let *T* be a trace composed of the sequence of dynamic instructions $<i_1, i_2, ..., i_n>$. If *T* is reusable, then i_k is reusable for every $k \in [1,n]$.

Theorem TLR2. Let *T* be a trace composed of the sequence of dynamic instructions $<i_1, i_2, ..., i_n>$. If i_k is reusable for every $k \in [1,n]$, then *T* is not necessarily reusable.

Theorem TLR1 implies that the number of instructions whose execution can be avoided by any trace reuse scheme is limited by the amount of individual instructions that are reusable. We can therefore compute an upper bound of the benefits of trace-level reuse by assuming that the amount of trace-level reusability is equal to the amount of instruction-level reusability, and the overhead of trace-level reuse is given by grouping reusable instructions into the minimum number of traces (i.e. assuming maximum-length traces). Theorem TLR2 states that this approach results in an upper bound that may not be reached.

4.2.3. Performance Improvement of Trace-Level Reuse

In this section we analyse the effect of trace-level reuse on execution time and focus on obtaining performance limits. We therefore consider a reuse engine with infinite tables to keep a history of previous instructions and analyse the effect of several reuse latencies. Reuse latency corresponds to the length of time a reuse operation takes. This usually involves a table look-up and some comparisons. In Section 4.3 we measure the amount of trace-level reusability when a finite reuse memory and a particular heuristic for trace collection are considered

The process for computing the IPC for this scenario is as follows. The completion time of every instruction that does not belong to a reusable trace is computed in the same way as in the base machine. For a reusable trace, the completion time of all the instructions that produce an output is computed as the maximum of the completion time of all the producers of its inputs plus the reuse latency. We also analysed two ways to consider the reuse latency. In one, the reuse latency is assumed to be a constant time per reuse operation. In the other, the reuse latency is assumed to be proportional to the number of inputs plus the number of outputs of the trace. Note that the first way is more appropriate when the reuse test just requires a valid bit to be checked, and the second way models the fact that reusing a trace requires the processor to read all its inputs and check that they are the same as in a previous execution.

For the limited window scenario, a reusable trace does not need to be fetched, and a single entry in the instruction window is allocated to represent the whole trace. Therefore, when computing the completion time of a given instruction, the graduation time of the instruction that is W locations above must be interpreted by taking into account only those instructions that must be brought into the processor. In other words, trace-level reuse has the additional advantage of artificially increasing the effective instruction window size, since some instructions are not even fetched and do not use any instruction slot.

In any case, if the completion time of an instruction in a reusable trace is higher than the completion time of the normal execution of that instruction, the latter will be chosen.

Performance figures of trace-level reuse are shown in Figure 4.1. Figure 4.1.a corresponds to an infinite window scenario and Figure 4.1.b corresponds to a finite window scenario. In both cases, a 1-cycle reuse latency has been considered. On average, the speed-up obtained for the infinite window scenario is around 3.03, and is slightly lower for FP than for integer programs. The greatest benefit is experienced by *ijpeg* (11.57). However, some programs have a negligible speed-up in this scenario

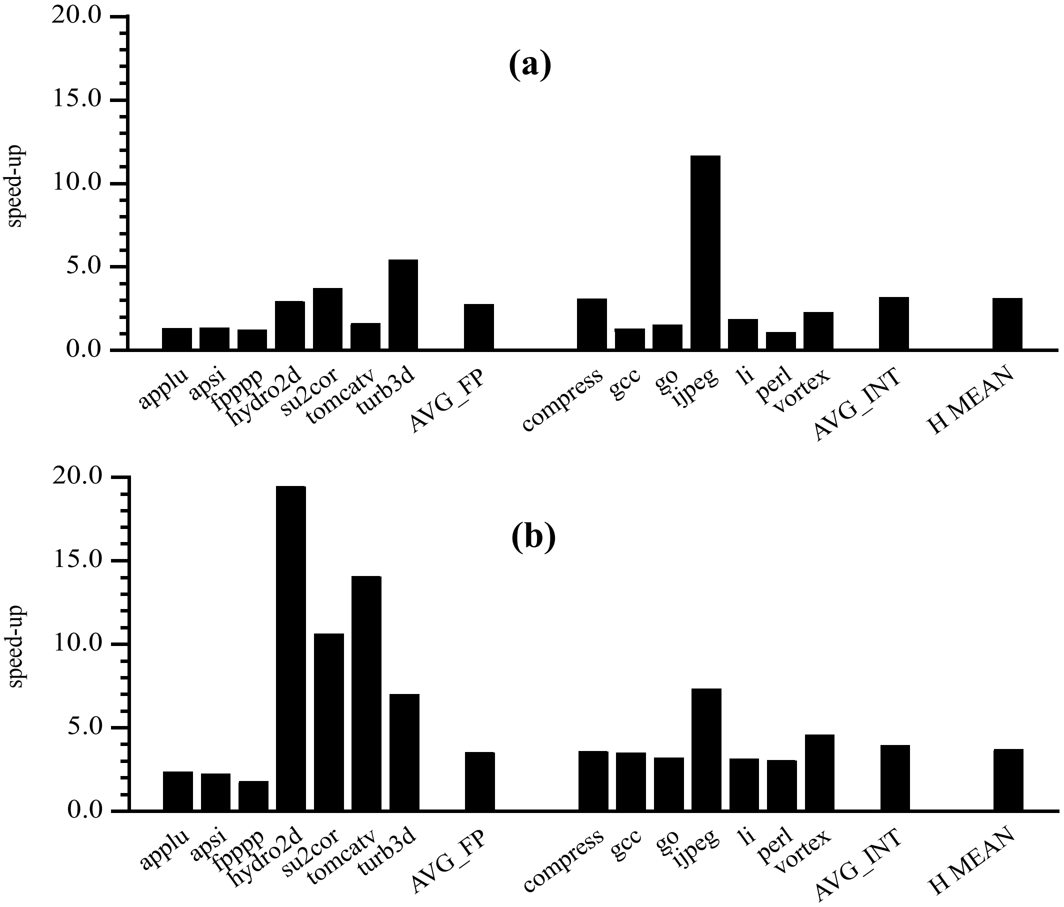


Figure 4.1. Speedup of trace-level reuse when considering a 1-cycle reuse latency: (a) for an infinite instruction window, (b) for a 256-entry instruction window.

(*perl* with 1.01). On the other hand, the average speed-up obtained for the limited window scenario is around 3.63. Once more, this speed-up is slightly lower for FP than for integer programs even though some FP programs, such as *hydro2d*, *su2cor*, *tomcatv* and *turb3d*, can significantly benefit from trace-level reuse.

We also found that the average speed-up is much higher than for instruction-level reuse in Chapter 3. For the infinite window scenario, speed-up increased from 1.43 (see Figure 3.5.a) to 3.03 (see Figure 4.1.a). This difference between trace-level and instruction-level reuse is even higher for the limited window scenario (Figure 3.3.a shows an average speed-up of 1.50 and Figure 4.1.b shows an average speed-up of 3.63). In this case, trace-level reuse may provide a very important extra advantage: it may avoid fetching instructions in reused traces and may increase the effective instruction window. Once the control flow reaches the initial address of a trace, if the trace is determined to be reusable the whole

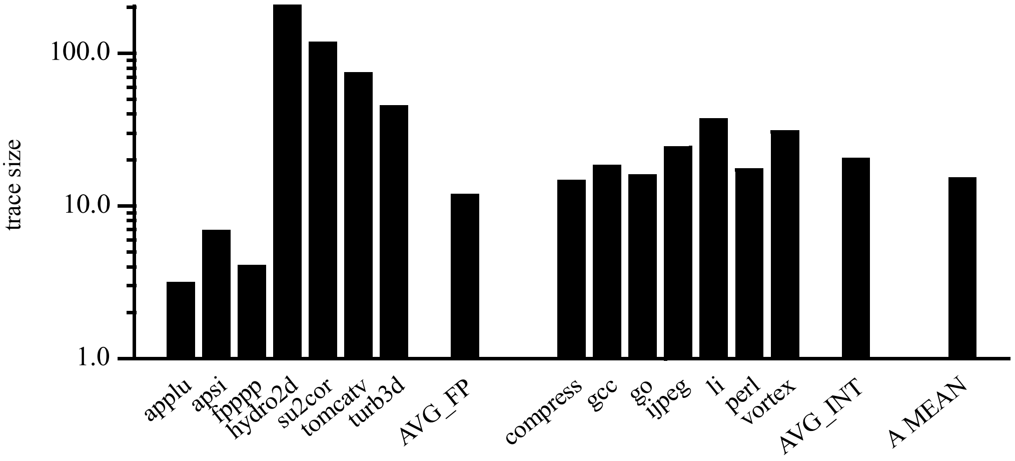


Figure 4.2. Average trace size.

trace can be reused without fetching or executing the instructions of the trace. Consequently, the speed-up of trace-level reuse for a limited instruction window is even higher than for an unlimited window (3.63 vs. 3.03), whereas for instruction-level reuse we observed the opposite trend.

Figure 4.2 shows the average trace size and correlates it with that in Figure 4.1. Note that, in general, larger traces imply higher speed-ups, which can be attributed to their greater potential to artificially increase the effective instruction window size. Integer programs have a quite uniform trace size, ranging from 14.5 to 36.7 instructions, and also exhibit a quite homogeneous speed-up. On the other hand, some FP programs, such as *applu, apsi* and *fpppp* have very short traces and exhibit very low speed-up, whereas others have large traces (up to 203 instructions for *hydro2d*) and high speed-ups.

Note also that trace-level reuse, unlike instruction-level reuse, provides significant speed-ups even if the reuse latency is higher than 1. This is shown in Figure 4.3.a, where we can see that the average speed-up for a reuse latency ranging from 1 to 4 cycles is not hardly degraded.

Note that a trace is reused provided that all input values are the same as in a previous execution. Therefore, a trace reuse operation may involve checking as many values as the number of inputs of the trace. Also, as the result of a trace reuse, all the output values of the trace must be updated. It may therefore be more realistic to assume that the reuse latency is proportional to the number of input and output values, i.e. it is equal to a constant *K* multiplied by the number of input/output values. *K* is the inverse of the read/write bandwidth of the reuse engine; for instance, *K*=1/16 implies that the reuse engine can read or write 16 values per cycle. In this scenario, the speed-up of trace-level reuse is shown

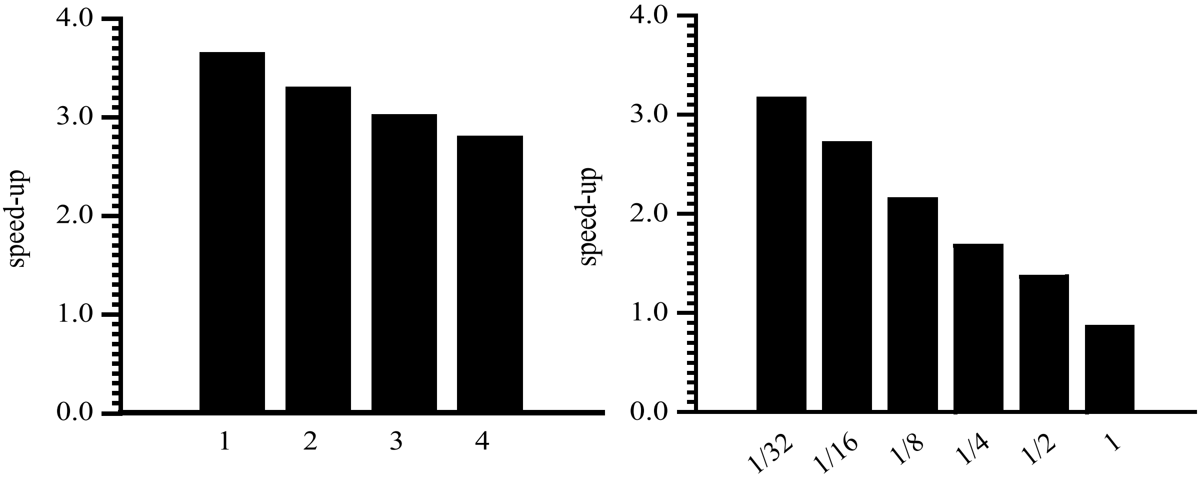


Figure 4.3. Speed-up of trace-level reuse for a 256-entry instruction window with a reuse latency that (a) varies from 1 to 4 cycles and (b) is proportional to the inputs plus outputs of the trace.

in Figure 4.3.b, where the X axis represents different values of *K*. We can also see that the speed-up of value reuse is high, even though it is significantly affected by the reuse latency. Note that it is reasonable to assume that microprocessors may have the capability to perform around 16 reads+writes per cycle, including register and memory values. In fact, current microprocessors such as the Alpha 21264 [44] can perform 14 reads+writes per cycle (8 register reads, 4 register writes and 2 memory references). If we look at the bar corresponding to *K*=1/16 in Figure 4.3.b, therefore, we can conclude that it is reasonable to expect a speed-up of around 2.7 from trace-level reuse. This also suggests that even the slowest approach to checking reusability, based on comparing all inputs, can significantly improve performance.

On average, we found that the number of input values per trace is 6.5 (2.7 register values and 3.8 memory values) and that the number of output values is 5.0 (3.3 register values and 1.7 memory values). Since the average number of instructions per trace is 15.0, this means that each reused instruction requires 0.43 reads and 0.33 writes, which is significantly lower than the number of reads and writes required by the execution of an instruction. We can therefore conclude that trace-level reuse also provides a significant reduction in the data bandwidth requirements, and can reduce the pressure on the memory and register file ports.

4.3. Design Issues of Trace-Level Reuse

In the previous section we demonstrated the high potential of trace-level reuse. The aim of this section is to identify design issues and propose alternatives for integrating a trace-level reuse scheme in a superscalar processor.

To reuse traces the processor needs to include some type of memory to store previous traces, decide which traces are worth storing, identify when the forthcoming trace can be reused and update the processor state if the trace is reusable. The techniques involved in these processes are addressed below in greater detail. Finally, we also measure the percentage reusability and the average trace size provided by trace-level reuse when a finite reuse memory and a particular heuristic for trace collection are considered.

4.3.1. Reuse Trace Memory

Reuse trace memory (RTM) is a memory that stores previous traces that are candidates to be reused. The RTM can be indexed by different schemes e.g. by PC, or by a hashing of the PC and the contents of a given register, etc. From the reuse point of view, a trace is identified by its input and its output (see Figure 4.4). The input of a trace is defined by:

- The starting address, i.e. initial program counter (PC).
- The set of register identifiers and memory locations that are live and their contents before the trace is executed. A register/memory location is live if it is read before it is written.

The output of a trace consists of:

- The set of registers and memory locations that the trace writes and their contents after the trace is executed.
- The address of the next instruction to be executed after the trace.

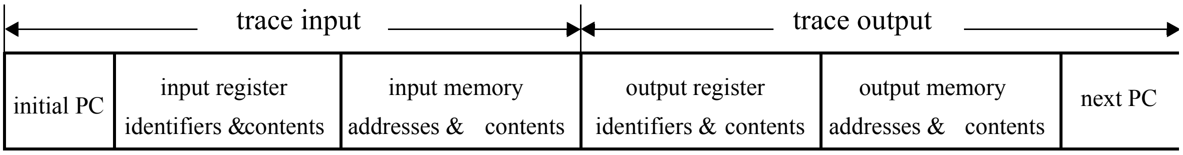

trace input
trace output
initial PC
input register identifiers &contents
input memory addresses & contents
output register identifiers & contents
output memory addresses & contents
next PC

Figure 4.4. A RTM entry.

4.3.2. Reuse Test and Processor State Update

We analyse the reuse test and processor state update in the pipeline. The reuse test refers to the validity check for establishing that the current operand values of a trace are the same as previously observed ones. The processor state update refers to the way in which output values of a trace are updated into the pipeline.

Note that, at some points of the execution (e.g. every cycle), the processor needs to check whether a trace that starts at the current PC can be reused. If it can, the processor uses the information about the trace obtained from the RTM to update its state as follows (see Figure 4.5):

- The PC is updated with the *next PC* field so that the fetch unit proceeds with the instructions that follow the trace. Instructions that belong to the trace do not need to be fetched.
- The output registers and output memory locations are updated with the values obtained from the RTM entry.

Basically, there are two ways to identify whether a trace is reusable. One is to read the current values of all input registers and memory locations and compare them with the values in any RTM entry associated with the current PC. Another is a valid bit to add to each RTM entry. When a trace is stored, its valid bit is set. For every register/memory write, all the RTM entries with a matching register/ memory location in its input list are invalidated. This approach requires a much simpler reuse test (just checking the valid bit). The final reuse process that updates the processor state can be implemented by inserting, in the instruction window, instructions that write the corresponding values in the trace output (registers and memory locations). In this way, precise exceptions could be guaranteed in an out-of-order processor following the conventional mechanism.

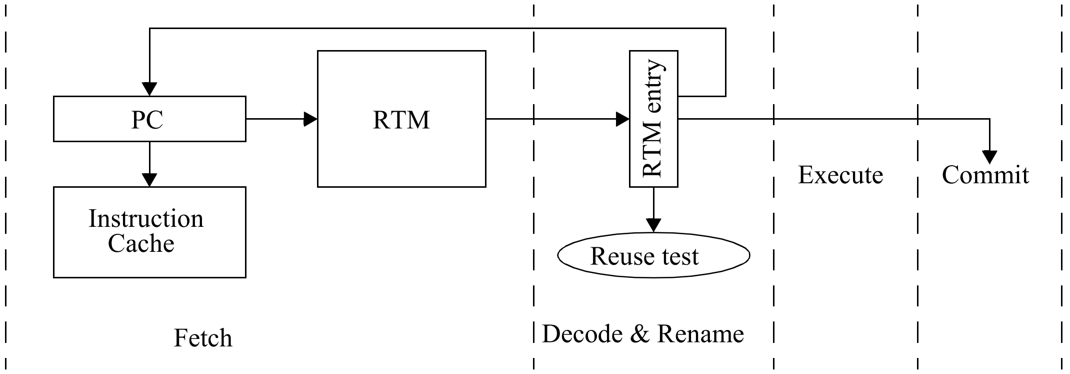
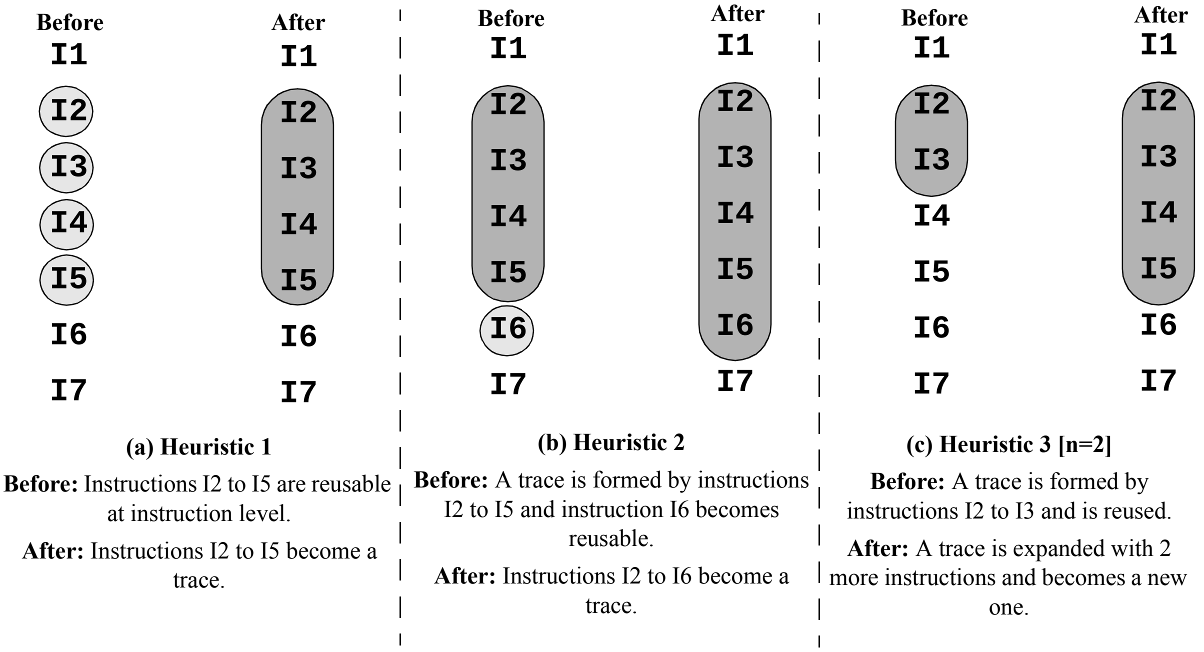


Figure 4.5. Trace-level reuse in the pipeline

4.3.3. Dynamic Trace Collection

The processor dynamically decides which traces of the dynamic stream are candidates to be reused. We propose three basic heuristics that can be considered to decide the start and end points of a trace. For example, a suitable criterion could be to start a new trace when a reusable instruction is encountered and to terminate the trace just before the first non-reusable instruction is found. Another may be to consider fixed-length traces that can be dynamically expanded once they are reused. Figure 4.6 shows examples of each dynamic trace collection heuristic.

- Heuristic 1 (H1): A trace consists of a sequence of dynamic instructions that are reusable at instruction-level. In this case, a different reuse memory for testing instruction-level reusability is also needed. This memory has as many entries as the RTM (see Figure 4.6.a).
- Heuristic 2 (H2): As before except that traces can be dynamically expanded when two consecutive traces are reused or instructions following a reused trace become reusable (see Figure 4.6.b).
- Heuristic 3 (H3[*n*]): A trace is formed by a fixed number of *n* instructions. When a trace is reused, it is expanded with *n* new instructions (see Figure 4.6.c).



(a) Heuristic 1

Before: Instructions I2 to I5 are reusable at instruction level.

After: Instructions I2 to I5 become a trace.

(b) Heuristic 2

Before: A trace is formed by instructions I2 to I5 and instruction I6 becomes reusable.

After: Instructions I2 to I6 become a trace.

(c) Heuristic 3 [n=2]

Before: A trace is formed by instructions I2 to I3 and is reused.

After: A trace is expanded with 2 more instructions and becomes a new one.

Figure 4.6. Examples of dynamic trace collection heuristics

Note that traces can have a variable number of instructions. In fact, the instructions that make up a trace are not stored in the RTM. Obviously, there may be implementation parameters that limit the size of a trace, e.g. the number of input and output values that can be stored in each RTM entry, but the number of instructions in a trace is not by itself a limitation.

4.3.4. Trace-Level Reusability with Finite Tables

The aim of this section is to measure the percentage of reusability and the average trace size that a trace-level reuse mechanism can provide when a finite reuse memory and a particular heuristic for trace collection are considered.

4.3.4.1. Experimental Framework

We have developed a functional parameterized simulator to evaluate trace-level reusability. To do so, we considered the same subset of the Spec95 benchmarks and compiler options as those in Section 4.2.1. The compiled programs were instrumented with the Atom tool [117] and their dynamic trace were processed in order to obtain statistics of reusability and trace sizes (see Section 1.4 for further details of tools and benchmarks).

We have evaluated several capacities for the Reuse Trace Memory (RTM):

- 512 entries: A 4-way set-associative memory (5-bit index) with 4 entries per initial PC. This means that up to 4 different traces starting at the same PC can be stored.
- 4K entries: A 4-way set-associative memory (7-bit index) with 8 entries per initial PC.
- 32K entries: An 8-way set-associative memory (8-bit index) with 16 entries per initial PC.
- 256K entries: An 8-way set-associative memory (11-bit index) with 16 entries per initial PC.

The reuse test is based on an associative search of traces that start at the same PC. In all cases, the memory is indexed by the least-significant bits of the PC register. Replacement policy is LRU, i.e. the oldest trace with the same PC that has been reused is the one that is replaced when a new trace is collected. For each trace, the number of inputs and outputs is limited to 8 registers and 4 memory values.

We also evaluated the three heuristics described in Section 4.3.3 for dynamic trace collection: H1, which builds traces with a sequence of dynamic instructions that are reusable at instruction-level; H2, which builds traces like H1 but these traces may be dynamically expanded; and H3[n], which builds traces with a fixed number of *n* instructions. Also, dynamic trace expansion (in H2 and H3[n]) is performed while the limits on the number of input and output locations are not exceeded. Note that, due to the LRU policy, the expanded traces replace the older trace with the same PC.

Finally, the reuse test is performed for every fetch operation and, when a trace beginning at a given PC contains the same values in its input locations as the current ones, the trace is reusable.

4.3.4.2. Analysis of Results

Figure 4.7 shows the percentage of reusable instructions and Figure 4.8 shows the average trace size for every evaluated heuristic.

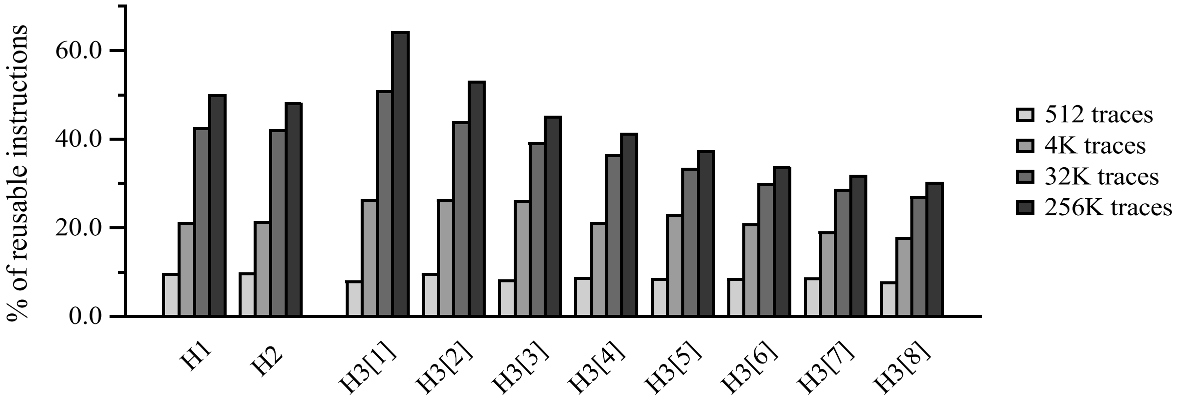


Figure 4.7. Percentage of reusable instructions of TLR with finite tables.

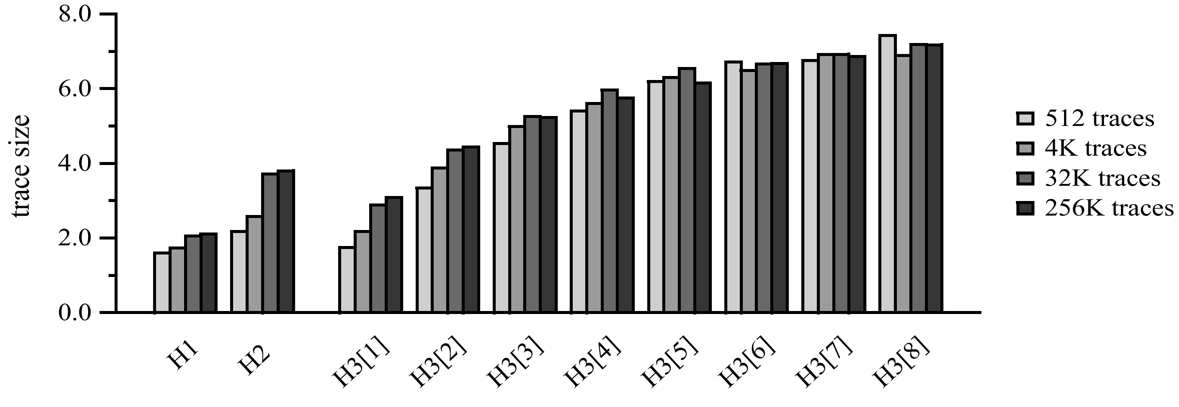


Figure 4.8. Average trace size of TLR with finite tables.

We can see that dynamic trace expansion is important for increasing the granularity of reusable traces while the total amount of reusability remains almost constant (see heuristics H1 and H2). Note also that heuristic H3[n] outperforms ILR, i.e. the best method of collecting traces should consider all kinds of instructions rather than just those that are reusable at instruction-level.

Also important is the relationship between the RTM size and the achieved reusability. For example, a 4K-entry RTM can reuse 25% of the dynamic instructions with an average trace size of 6 instructions. The percentage of reused instructions grows significantly as the RTM capacity increases. Finally, note the trade-off between the percentage of reused instructions and trace size. Increasing the trace size reduces the number of reused instructions. However, to achieve a given degree of reuse, the reuse overhead decreases when the trace size increases.

4.4. Related Work

Since we initially published the concept of trace-level reuse [37],[38] much work has been done in this area. Contemporary to our work, Huang and Lilja [49] proposed a scheme to reuse basic blocks. Basic block reuse is a particular case of trace-level reuse in which traces are limited to basic blocks. Note that trace-level reuse is more general and can exploit reuse in larger sequences of instructions, such as subroutines, loops, etc. Huang and Lilja also proposed a modification of their initial mechanism of basic block reuse called sub-block reuse [50]. The idea is to slice basic blocks into sub-blocks to balance the reuse granularity and the number of reuse opportunities. They also investigated compiler assistance [48] to carefully slice basic blocks that produce more value reuse opportunities while maintaining a reasonable granularity.

Sodani and Sohi proposed the *reuse buffer* [110], which is a hardware implementation of data value reuse at instruction level (or dynamic instruction reuse, as it is called in that study). The reuse buffer is indexed by the instruction address. They proposed three different reuse schemes. In the first scheme, for each instruction in the reuse buffer, it holds the source operand values and the result of the last execution of this instruction. In the second scheme, instead of the source operand values, the buffer holds the source operand names (architectural register identifiers). In the third scheme, in addition to the information of the second scheme, the buffer stores the identifiers of the producer instructions of the source operands. In this scheme, dependent instructions that are fetched simultaneously can be reused by chaining their individual reuses. However, the reuse of each individual instruction is still a sequential process since it must wait until the reuse of all previous instructions has been checked.

Connors and Hwu [29] introduced a novel approach that integrates architecture and compiler techniques to exploit value locality for large regions of code. The idea is to rely completely on the compiler to identify reusable regions. The compiler therefore performs analysis based on value profiling to identify code regions whose computations can be reused during dynamic execution. Later, they proposed a new hardware model that reduces the need for value profiling at compilation time [30]. With this model, the compiler is allowed to designate reusable regions that may prove to be inappropriate.

Costa *et al* [31] designed another hardware-based reuse technique that uses memoization tables to skip the execution of sequences of redundant instructions. The main differences between this and other reuse schemes are the management of individual redundant instructions and the exclusion of memory access instructions from the validity domain.

Sastry *et al* [98] investigated the properties of reuse in the context of a dynamic optimization setting by characterizing the available computation reuse in programs at coarse granularities and determining the relative applicability of specialization and memoization.

Sazeides [104] introduced the concept of *Instruction Isomorphism*. An instruction instance is said to be isomorphic if its component, which is the information derived from the instruction and its backward dynamic data dependence graph, is identical to the component of an instruction executed earlier. By definition, an isomorphic instruction will produce exactly the same output with the earlier instruction.

Value reuse and value prediction have also been combined at trace level. Wu *et al* [129] resembled the compiler-directed computation reuse scheme of Connors and Hwu [29]. These authors proposed a speculative multithreading architecture that consisted of two execution cores with dedicated functionality to support integrated region-level computation reuse and value prediction. Pilla *et al* [87],[88] added value prediction and memory reuse to extend the work of Costa *et al* [31]. Basically, these authors increased the number of traces that can be reused by predicting the values of trace inputs that are not available when reuse is applied.

4.5. Conclusions

In this chapter we have introduced the concept of trace-level reuse, demonstrated its great potential, and identified design issues for integrating this technique into a superscalar processor. The underlying concept of trace-level reuse is similar to that of instruction-level reuse. Both are data value reuse techniques but trace-level reuse handles dynamic sequences of instructions rather than single instructions. It is therefore based on the observation that several computations of the programs tend to be repetitive, but we only consider repetitive computations at local level as a source of trace-level reuse.

We found several differences between this technique and instruction-level reuse. Instruction-level reuse can exploit a higher degree of reuse than trace-level reuse and may provide very large speedups for an ideal machine. However, instruction-level reuse also has more overheads because it requires more reuse operations. When reuse latency is considered, trace-level reuse is much less degraded and may even outperform instruction-level reuse. We have shown that trace-level reuse has many positive effects: a) it reduces the fetch bandwidth requirement by avoiding fetching instructions of reused traces; b) it increases the effective instruction window size by avoiding storing instructions of reused traces in the instruction window; c) it has fewer overheads because it requires fewer operations per reused instruction.

Simulation results show that when the reuse latency is 1 cycle, the instruction window has 256 entries and history tables are unbounded, trace-level reuse provides an average speed-up of 3.6 and ranges from 1.7 to 19.4 for individual programs. Results are similar when reuse latency is considered to be proportional to the number of inputs and outputs of a trace. These results were obtained by performing a detailed analysis of the performance potential of trace level reuse.

Finally, we have identified several design issues for integrating a trace-level reuse scheme into a superscalar processor. We addressed essential issues such as memory for storing previous traces, approaches for deciding which traces are useful, a mechanism for identifying reusable traces and how to update the processor state. We have also evaluated the impact of a limited-capacity history table with different trace collection heuristics in a trace-level reuse scheme. For example, for a 4K-entry Reuse Trace Memory we found that percentage reusability is around 25% of all dynamic instructions and that the average trace size is around 6 instructions. For a 256K-entry Reuse Trace Memory, around 50% of instructions can be reused.

“Prediction is very difficult, especially about the future”
Niels Bohr, Danish Physicist, 1885-1962.

Chapter 5

TRACE-LEVEL SPECULATION

Trace-level speculation is a data value speculation technique that avoids the execution of a dynamic sequence of instructions by predicting the values produced by those instructions. This technique exploits the high percentage of repetition in the computations of conventional programs to increase the instruction-level parallelism.

In this chapter we propose a novel microarchitecture to exploit trace-level speculation that has a low misspeculation penalty and can speculate on any dynamic trace of instructions. We also propose a trace selection method based on a static analysis that uses profiling data to determine large regions of code whose live-output values can be successfully predicted.

5.1. Introduction

Data dependences are one of the most important hurdles that limit the performance of current microprocessors. Two techniques have so far been proposed to avoid the serialization caused by data dependences: data value speculation [70] and data value reuse [110]. Both techniques exploit the high percentage of repetition in the computations of conventional programs. Speculation predicts a given value as a function of past history. Value reuse is possible when a given computation has already been made exactly. Both techniques can be considered at two levels: the instruction level and the trace level. The difference is the unit of speculation or reuse: an instruction or a dynamic sequence of instructions.

Reusing instructions at trace level means that the execution of a large number of instructions can be skipped in a row. More importantly, as these instructions do not need to be fetched, they do not consume fetch bandwidth. Unfortunately, trace reuse introduces a live-input test that it is not easy to handle. Especially complex is the validation of memory values. Speculation may overcome this limitation but it introduces a new problem: penalties due to a misspeculation.

Trace-level speculation avoids the execution of a dynamic sequence of instructions by predicting the set of live-output values based, for instance, on recent history. There are two important issues with regard to trace-level speculation. The first of these involves the microarchitecture support for trace speculation and how the microarchitecture manages trace speculation. The second involves trace selection and data value speculation techniques.

Several thread-level speculation techniques [9], [28], [93], [94], [97], [140] have recently been explored to exploit parallelism in general-purpose programs. We lay on the same trend and propose a microarchitecture called Trace-Level Speculative Multithreaded Architecture (TSMA), which is tolerant to misspeculations in the sense that it does not introduce significant trace misprediction penalties and does not impose any constraint on the approach to building or predicting traces.

Traces are identified by an initial point and a final point in the dynamic instruction stream, and data speculation refers to the prediction of a trace's live-output values. Traces can be built according to various heuristics such as basic blocks and loop bodies, etc [29], [38], [49]. Once a trace is built, live-output values can be predicted in several ways, including using conventional value predictors such as last value, stride, context-based and hybrid schemes [74], [105]. We also focus on developing effective trace selection schemes for TSMA processors. In this way, we propose a trace selection method based on a static analysis that uses profiling data.

In this chapter we analyse trace-level speculation, discusses its implementation and analyse its performance impact. Section 5.2 reviews some approaches to exploit trace-level speculation. Section 5.3 describes in detail the microarchitecture proposed to exploit trace-level speculation. Section 5.4 presents extensions to TSMA that minimize misspeculation penalties and develop trace selection schemes. Section 5.5 reviews related work and, finally, Section 5.6 summarizes our main conclusions.

5.2. Approaches to Trace-Level Speculation

Trace-level speculation is a dynamic technique (although the compiler may help) that requires a live-input or live-output test. In this section, we describe both approaches.

5.2.1.Trace-Level Speculation with Live-Input Test

This approach (see Figure 5.1) is supported by means of a multithreaded architecture. The underlying concept is to have a couple of threads working cooperatively. One thread is in charge of trace speculation while the other is in charge of live-input validation. When the main thread speculates a trace, another thread is spawned. The main thread skips the trace and performs the live-output update. It then begins to execute instructions speculatively. Meanwhile, the spawned thread validates live-input values. If validation succeeds, the spawned thread dies. If validation does not succeed, recovery actions are required and the main thread returns to the point where the trace was speculated.

We can also consider another version of this approach that reduces misspeculation penalty. With this approach, the spawned thread executes the skipped code in parallel with live-input validation. Then, if validation fails, the main thread dies and the spawned thread becomes the main thread.

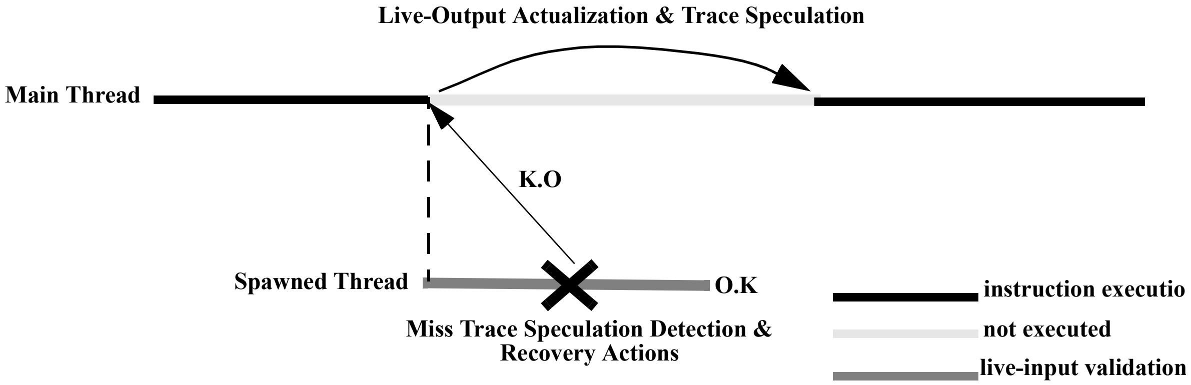

Live-Output Actualization & Trace Speculation
Main Thread
K.O
Spawned Thread
O.K
Miss Trace Speculation Detection & Recovery Actions
instruction executio
not executed
live-input validation

Figure 5.1. Trace-level speculation with live-input test

5.2.2. Trace-Level Speculation with Live-Output Test

This approach was introduced by Rotenberg *et al* [90], [93], [94], [119] as the underlying concept behind *Slipstream Processors*. This approach is supported by means of a couple of threads (a speculative thread and a non-speculative one) working cooperatively to execute a sequential code.

Let us consider the program of Figure 5.2.a that is composed by three pieces of sequential code or traces. Figure 5.2.b shows the execution of the program from the point of view of code and Figure 5.2.c shows the execution of the program from the point of view of time.

The speculative thread executes instructions and speculates on the result of whole traces. The non-speculative thread verifies instructions that are executed by the speculative thread and executes speculated traces. Each thread maintains its own state but only the state of the non-speculative thread is

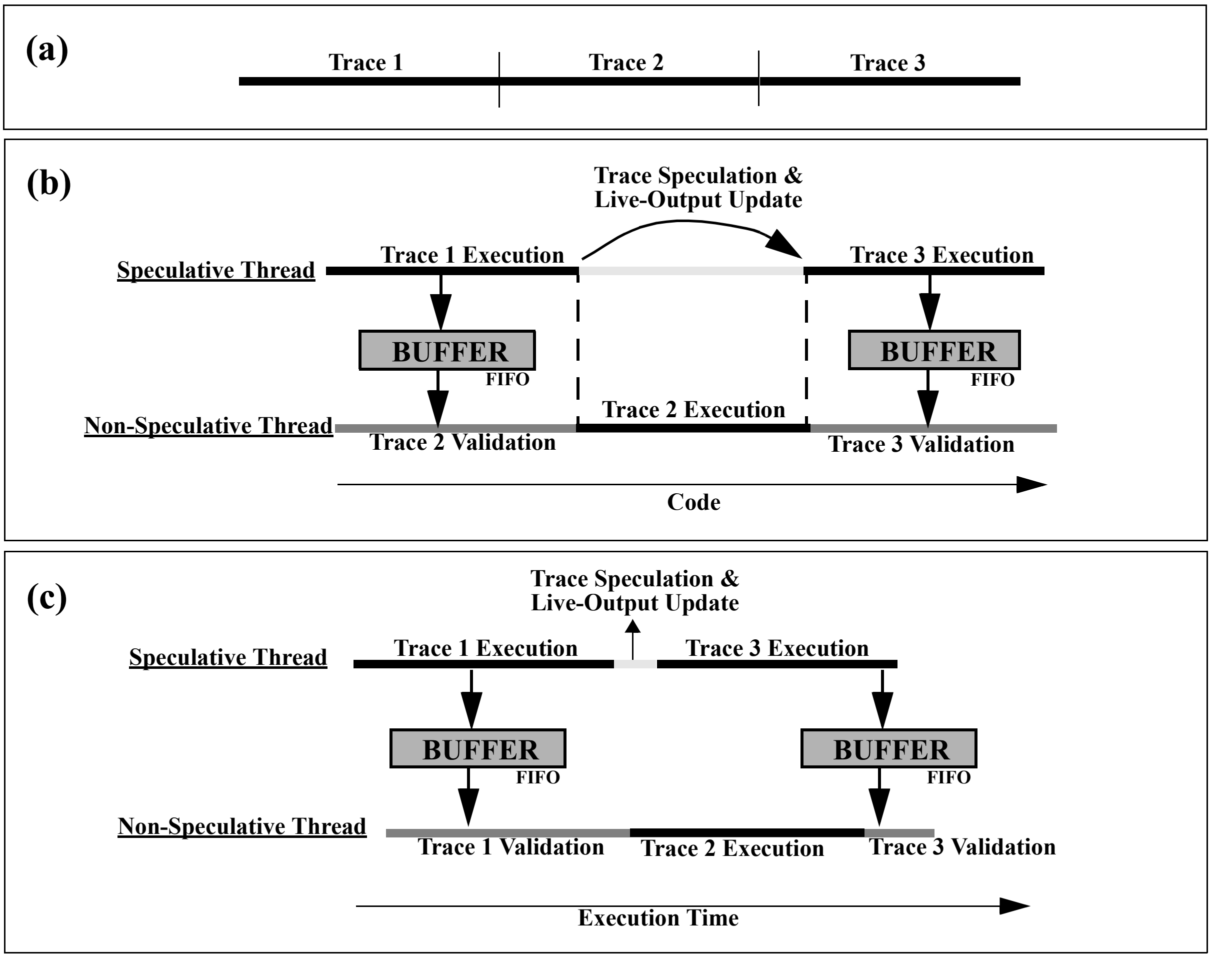


Figure 5.2. Trace-level speculation with live-output test
(a) program (b) point of view of code (c) point of view of time

guaranteed to be correct. Communication between threads is done by means of a buffer that contains the executed instructions by the speculative thread. Once the non-speculative thread executes the speculated trace, instruction validation begins. This is done by verifying that source operands match the non-speculative state and updating the state with the new result. If validation does not succeed, recovery actions are required.

Note that speculated traces are validated by verifying their live-output values. Live-output values are those that are produced and not overwritten within the trace. The advantage with this approach is that only live-output values that are used are verified. Moreover, verification is fast because instructions consumed from the buffer have their operands ready (see trace3 execution and validation in Figure 5.2.c). Finally, speed-up is obtained when both threads execute instructions at the same time and then, validation does not produce a misspeculation which implies to set some recovery actions (see trace2 and trace3 execution in Figure 5.2.c).

The microarchitecture presented in this chapter focuses on this approach: trace-level speculation with live-output test.

5.3. Trace-Level Speculative Multithreaded Architecture

This section outlines the main characteristics of Trace-Level Speculative Multithreaded Microarchitecture (TSMA). We first present an overview and, in the following subsections, describe in more detail the main components of the proposed microarchitecture. In the last subsection we present a working example.

5.3.1. Overview

The underlying concept of our proposal is based on a couple of threads that work cooperatively to perform trace-level speculation with live-output test. Figure 5.3 shows the proposed microarchitecture based on the approach described in Figure 5.2.

A TSMA processor can simultaneously execute a couple of threads (a speculative one and a non-speculative one) that cooperate to execute a sequential code. The speculative thread is in charge of trace speculation. The non-speculative thread is in charge of validating the speculation. This validation is performed in two stages: (1) executing the speculated trace and (2) validating instructions executed by the speculative thread. Speculated traces are validated by verifying their live-output values. Live-output

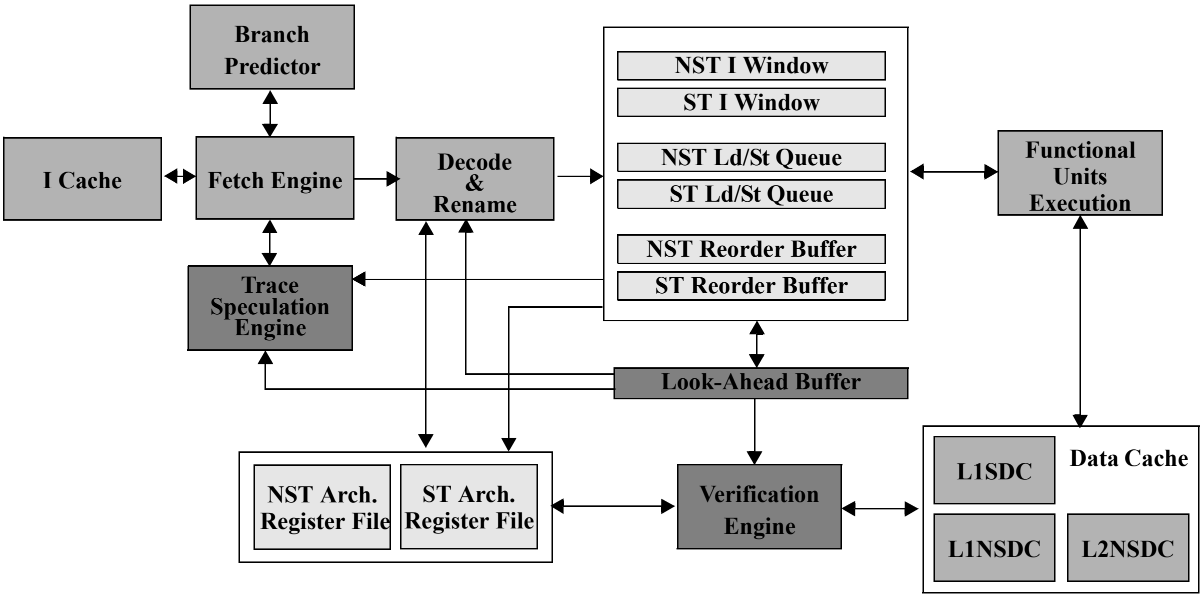


Figure 5.3. Trace-level speculative multithreaded microarchitecture.

values are those that are produced and not overwritten within the trace. In the rest of the paper we will use the terms ST and NST to refer to the speculative thread and the non-speculative thread, respectively. Note that ST runs ahead of NST.

Both threads maintain their own architectural state by means of their associated architectural register file and a memory hierarchy with some special features. NST provides the correct and non-speculative architectural state, while ST works on a speculative architectural state. Note that each thread maintains its own state but that only the state of NST is guaranteed to be correct.

Additional hardware is required for each thread. ST stores its committed instructions to a special FIFO queue called *Look-Ahead Buffer*. NST executes the skipped instructions and verifies instructions in the look-ahead buffer executed by ST. Note that verifying instructions is faster than executing them because instructions always have their operands ready. In this way, NST catches ST up quickly.

ST speculates on traces with the support of a *Trace Speculation Engine* (TSE). This engine is responsible for building traces and predicting their live-output values. NST, on the other hand, uses special hardware called a *Verification Engine.* The NST executes the skipped instructions and verifies instructions in the look-ahead buffer executed by ST. This is done by verifying that source operands match the non-speculative state and by updating the state with the new result in case they match. If there is a mismatch between the speculative source operands and the non-speculative ones, a trace misspeculation is detected and a thread synchronization is fired. Basically, this recovery action involves flushing the ST pipeline and reverting to a safe point in the program. An advantage with this approach is that any live-output values used are the only ones that are verified. Note also that the verification of instructions is faster than their execution because instructions always have their operands ready. A critical feature of this microarchitecture is that this recovery is implemented with minor performance penalties.

The hardware of TSMA (see Figure 5.3) can be divided into three categories:

- *Local*: each thread maintains a logical register file, an instruction window, a load store queue and a reorder buffer. All this hardware is replicated for both threads (light grey in Figure 5.3).
- *Shared*: non-replicated hardware is shared by both threads. These resources are the instruction cache, the fetch engine, the branch predictor, the decode and rename logic, functional units, a modified data value cache and logical control (grey in Figure 5.3).
- *Additional*: hardware requirements to support trace-level speculation. These resources are the look-ahead buffer, the verification engine and the trace speculation engine (dark grey in Figure 5.3).

The main parts of the *Trace-Level Speculative Multithreaded Microarchitecture* are described below in greater detail.

5.3.2. Trace Speculation Engine

The trace speculation engine (TSE) is responsible for two issues: (1) implementing a trace-level predictor and (2) communicating a trace speculation opportunity to the fetch engine. In this work we assume that the trace predictor maintains a simple PC-indexed table with N entries. Each entry contains live-output values and the final program counter of the trace. TSE receives from NST and the verification engine the information required to build traces and determine live-output values. This information comes from correctly executed instructions at commit time.

To determine trace speculation opportunities, TSE scans the current program counter of ST. This value is provided by the fetch engine. If TSE determines that the current PC is the beginning of a potentially predictable trace, it provides some trace information for the fetch engine. This information consists of a special INI_TRACE instruction and some MOV instructions. The INI_TRACE instruction contains the final program counter of the trace and the number of times that this PC is repeated inside the trace (this allows the TSE to construct traces that consists of multiple loop iterations). Additional MOV instructions inserted into the ST pipeline update live-output values of a trace. ST then continues with normal instruction fetch from the final point of the speculated trace. In the next cycle, NST wakes-up and begins to fetch and execute instructions of the speculated trace.

To ensure the correctness of the architectural state, ST may only speculate a new trace when the look-ahead buffer is empty. This means that TSMA has only a single unverified trace speculation at any given time.

5.3.3. Look-Ahead Buffer

The aim of this structure is to store instructions executed by ST. NST will later validate these instructions. The look-ahead buffer is just a first-input first-output queue, so a huge look-ahead buffer can be managed easily. ST introduces instructions at commit time whereas the verification engine takes these instructions and test their correctness. The fields of each entry of the look-ahead buffer are:

- Program counter
- Operation type: indicates memory operation
- Source register ID 1 & Source value 1
- Source register ID 2 & Source value 2
- Destination register ID & Destination value
- Memory address

5.3.4. Verification Engine

The verification engine (VE) is responsible for validating speculated instructions and, together with NST, maintains the speculative architectural state. Instructions to be validated are stored in the look-ahead buffer. Verification involves testing source values of the instruction with the non-speculative architectural state. If they match, the destination value of the instruction can be updated in the non-speculative architectural state (register file or memory). Memory operations require special considerations. First, the effective address is verified and, after this validation, store instructions update memory with the destination value. On the other hand, loads check whether the value of the destination

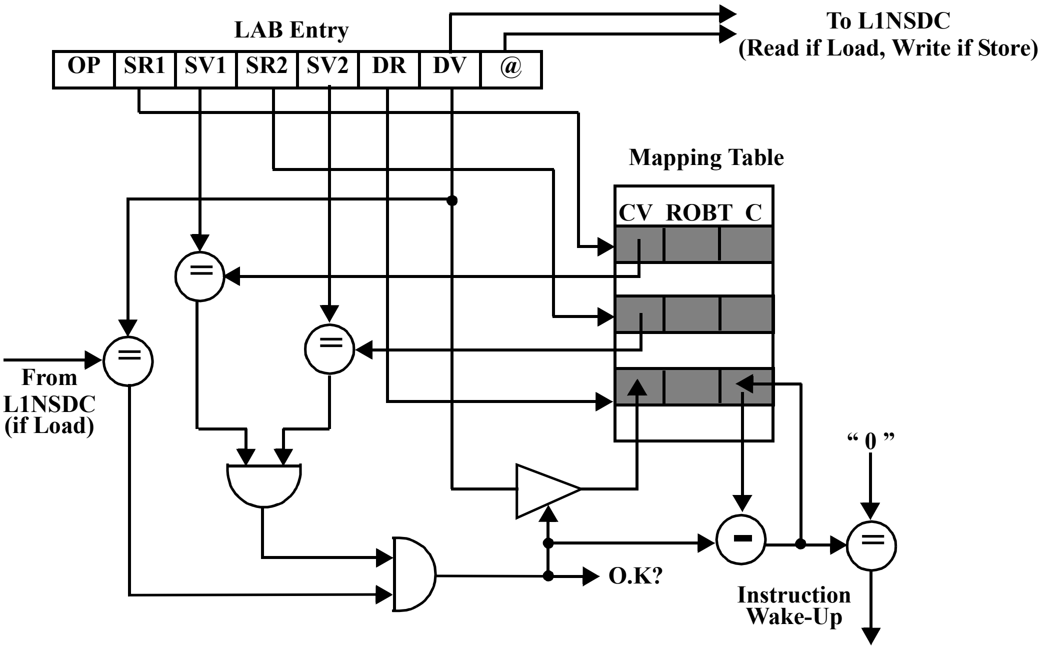


Figure 5.4. Verification engine block diagram.

register matches the non-speculative memory state. If it does, the destination value is committed to the register file.

This engine is independent of both threads but works cooperatively with NST to maintain the correct architectural state. Figure 5.4 shows a simple implementation of the verification engine. Note that the hardware required to perform the verification is minimal.

5.3.5. Thread Synchronization

Thread synchronization is required when a trace misspeculation is detected by the verification engine. Basically, this involves flushing the ST pipeline and returning to a safe point in the program. The recovery actions involved by a synchronization are simple:

- Instruction execution is stopped.
- ST structures are emptied (instruction window, load store queue, reorder buffer and look-ahead buffer).
- Speculative data cache and logical register file associated with ST are invalidated.

NST executes traces speculated by ST and the verification engine validates ST executed instructions once ST puts them in the look-ahead buffer. NST maintains execution beyond the final point of the speculated trace but commitment of these instructions is disabled in order to significantly reduce the penalties caused by synchronization. In this way, two types of synchronizations (total and partial) may occur.

Total synchronization occurs when a misspeculation is detected by the verification engine and NST is not executing instructions after the end of the trace. This implies squashing ST and paying the penalty of starting to fill its pipeline from the point it detected the misspeculation. On the other hand, partial synchronization occurs when a misspeculation is detected and NST is already executing instructions. In this way, the ST pipeline does not need to be refilled. NST takes the role of ST, enabling the commitment of the already executed instructions after the end of the speculated trace. Meanwhile recovery actions are taken to initialize ST with a correct architectural state at the failure point. After this synchronization, the roles of the threads are interchanged.

This partial synchronization avoids the pipeline refill penalty at the expense of the constraint that while NST is executing instructions beyond the end of a speculated trace, ST cannot speculate on a new trace. This number of additional executed instructions should therefore be quite small. On the other hand, it is important to minimize the number of total synchronizations without losing speculation opportunities. Empirically we have observed that trace misspeculations are detected relatively early. The processor dynamically determines the number of instructions to be executed after a speculated trace from on the number of verified instructions before a misspeculation is detected.

5.3.6. Memory Subsystem

We propose a new first level data cache architecture (see Figure 5.5). This cache architecture is responsible for maintaining the speculative memory state of ST. The first level of the memory hierarchy comprises two modules: the level 1 speculative data cache (L1SDC) and the level 1 non-speculative data cache (L1NSDC).The second level only contains non-speculative data and will be referred to as level 2 non-speculative data cache (L2NSDC).

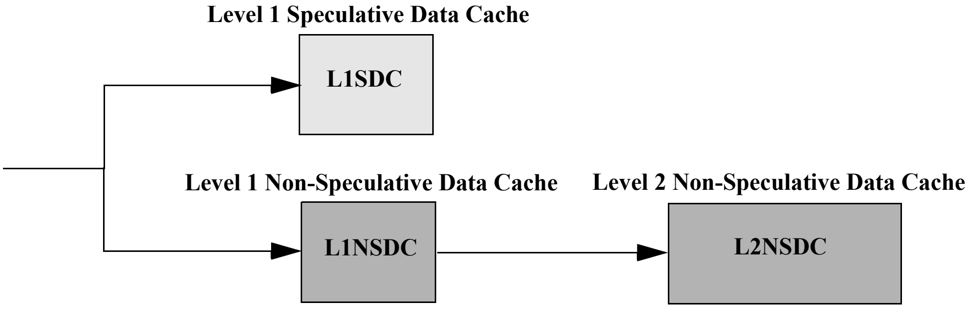

Level 1 Speculative Data Cache
L1SDC
Level 1 Non-Speculative Data Cache
L1NSDC
Level 2 Non-Speculative Data Cache
L2NSDC

Figure 5.5. Memory subsystem.

This new organization is guided by the following rules:

1. ST store instructions update values in L1SDC only.

2. ST load instructions get values from L1SDC. If a value is not in L1SDC, it is obtained from L1NSDC or L2NSDC in a traditional way. Access to L1SDC and L1NSDC is done in parallel. No line from L1SDC is copied back to L2NSDC.

3. NST store instructions (executed by NST or verified by the verification engine), update values and allocate space in the non-speculative caches only.

4. NST load instructions (executed by NST or verified by the verification engine), obtain values and allocate space in the non-speculative caches only.

5. A line replaced in L1NSDC is copied back to L2NSDC.

Note that rules 3 to 5 correspond to the normal management of traditional caches, while rules 1 and 2 describe the behaviour of the new speculative cache. Simulations show that a very small L1SDC may be enough to provide good performance.

The following figures present different scenarios depending on the order of the actions involved in the speculation, execution and verification of a trace. Note that this example considers a correct trace speculation. Figure 5.6 shows ST performing a trace speculation that includes a store instruction (1). After speculation, ST executes a load that refers to the memory location of the speculated store (2). On the other hand NST executes the speculated store (3) and later the load instruction is verified (4). Note that this example considers a correct trace speculation. On the other hand, Figure 5.7 shows the behaviour of the memory system for an incorrectly speculated trace.

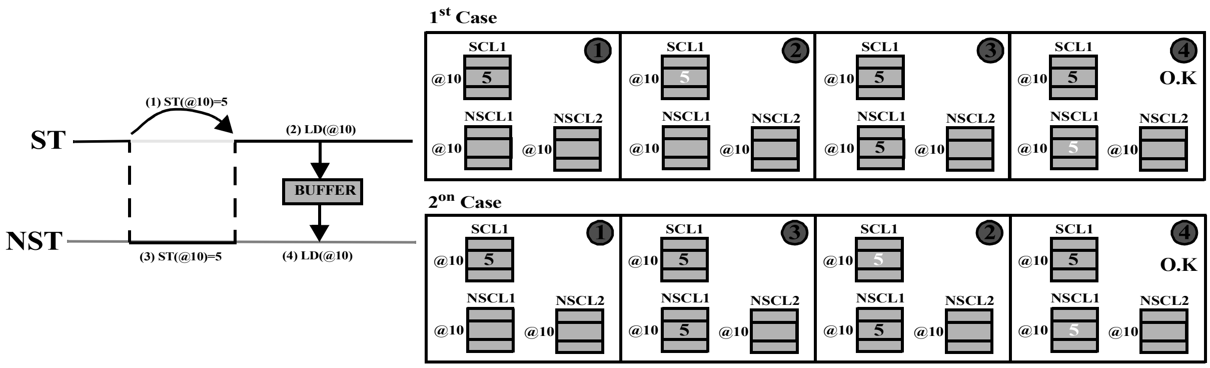


Figure 5.6. Example: correct trace speculation.

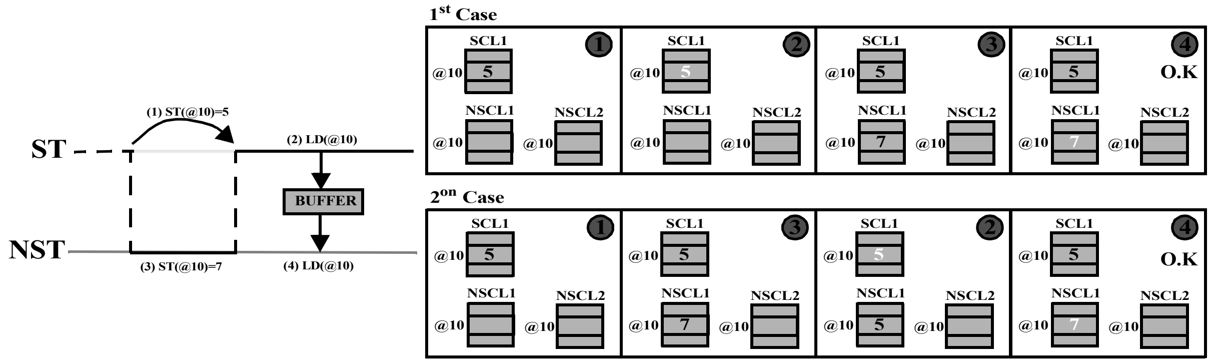


Figure 5.7. Example: incorrect trace speculation.

5.3.7. Register File

The proposed microarchitecture assumes a register renaming mechanism in which speculative register values are kept in the reorder buffer. Figure 5.8 illustrates the register map table, which for each entry contains the following fields:

- Committed Value: this contains the last committed value of the register.
- ROB Tag: this points to the ROB entry that has (or will have) the latest value of the register.
- Counter: this determines the number of instructions in ST that are using this register after a trace speculation.

Note that the difference between this and traditional structures is the new counter field. This field is used to provide NST with the ability to begin the execution of speculated traces as soon as possible. In

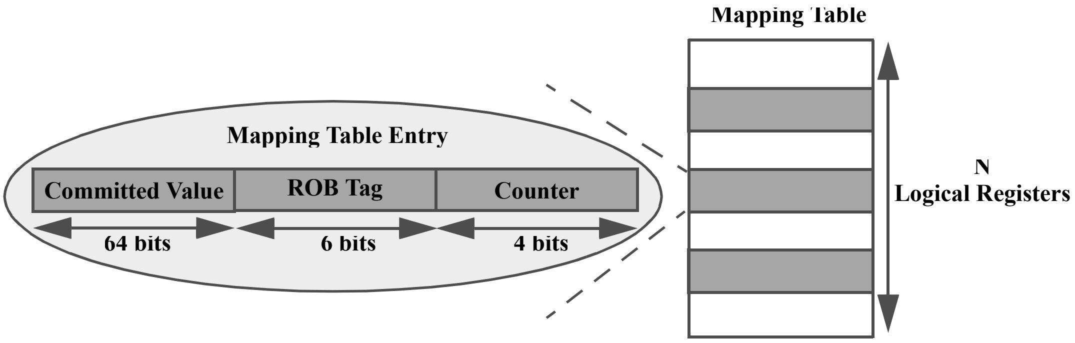


Figure 5.8. Register map table block diagram.

particular, once ST speculates a trace, NST immediately begins the execution of that trace. Notice also that there may be instructions from ST before the speculation point that still have not been executed. These instructions may produce values to be consumed by NST instructions. In this way, these dependent NST instructions have to wait for ST completion and for this reason, NST needs to know whether its instructions have source operands that are still not ready because ST has not finished their execution.

The counter field is maintained as follows:

1. When a ST instruction enters the instruction window, the counter associated to its destination register is increased.

2. When an instruction is committed to the look-ahead buffer by ST, the destination register counter is decreased.

3. After a trace speculation, the counter is no longer increased. It is just decreased until it reaches zero. If an instruction decoded by the NST encounters a source operand with a counter field equal to zero, this indicates that the instruction is younger than the speculation point.

ST passes a copy of the counters to NST when the special INI_TRACE instruction that determines trace speculation is renamed. The other fields of the register map table, i.e. the committed values and ROB tags do not need to be communicated. The verification engine then decreases the counters as it validates instructions. This is done until a special mark in the look-ahead buffer that determines the start of a trace speculation is reached. In this way, a counter greater than zero indicates that the register value is not ready because it has not been verified by the verification engine. This does not prevent NST from executing instructions that do not depend on this value. On the other hand, to guarantee correctness of memory state, memory instructions are stalled until the verification engine reaches the starting point of the trace speculation.

Note that there may be speculated traces in a path that is incorrectly speculated by a branch. In this case, NST begins execution but when ST determines an incorrect path and recovers, it stops and empties its thread private structures.

5.3.8. Working Example

To understand the behaviour of the microarchitecture, we present a detailed working example. Figure 5.9 shows the key steps of a trace speculation. Below is a detailed explanation of each step.

1. ST begins the execution of the program and commits the instructions to the look-ahead buffer.
2. The trace speculation engine identifies a trace speculation opportunity, notifies the fetch engine and provides ST with the information required through a special INI_TRACE instruction. At this point, the program counter is modified and additional instructions to update live-output values are provided. When the INI_TRACE instruction is renamed, NST receives a copy of the ST mapping table. Now ST maintains a speculative architectural state using its mapping table and the memory hierarchy in a speculative way.

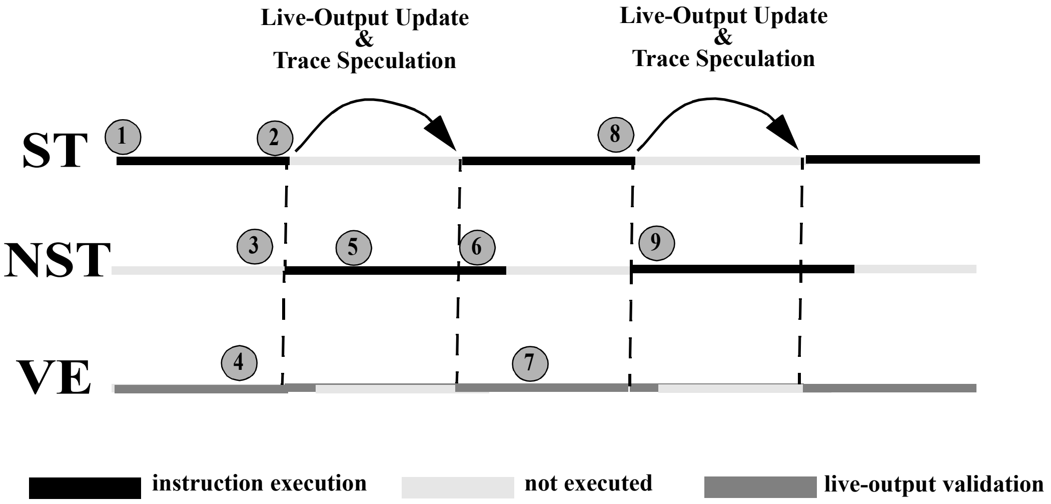

Live-Output Update & Trace Speculation
Live-Output Update & Trace Speculation
ST
NST
VE
1
2
3
4
5
6
7
8
9
instruction execution
not executed
live-output validation

Figure 5.9. TSMA behaviour: a working example.

3. NST begins to execute the ST skipped instructions immediately. This prompt execution is done through the support of the special mapping table, as described above.

4. VE consumes instructions from the look-ahead buffer and updates the architectural register file shared with NST. It decreases the register map counters and stores the committed value in the mapping table. This is done until an INI_TRACE instruction is reached and guarantees that NST does not execute instructions with values that are still not produced by ST.

5. NST executes instructions normally. It commits instructions and maintains the correct state.

6. NST detects the final point of the speculated trace. The verification engine begins to validate instructions and update the architectural state. NST continues executing instructions but commit is disabled. Memory instructions are stalled.

7. The verification engine validates ST executed instructions after the trace speculation. The verification engine guarantees the correct state and verifies N instructions from the look-ahead buffer. This number is set dynamically to be slightly larger than the average number of verified instructions that precede a misspeculation detection.

    - If verification fails, recovery actions are taken. If NST is still executing instructions, it takes the role of ST. This is known as partial synchronization. At this point, the state is safe so the verification engine becomes idle.

    - If there is no misspeculation among the N verified instructions, the NST structures are flushed and it becomes idle. The verification engine continues verifying instructions and maintaining the correct architectural state. If the verification engine finds a misspeculation when NST is idle, a total synchronization occurs. ST is squashed and refilled starting from the incorrect instruction.

8. ST may speculate on a new trace when the look-ahead buffer is empty. This ensures the correctness of the architectural state. In this way, NST is guaranteed to receive a correct copy of the mapping table.

9. NST executes the trace as described in point 2.

5.3.9. Performance Evaluation

In this section we evaluate the performance potential of the *Trace-Level Speculative Multithreaded Architecture*.

5.3.9.1. Experimental Framework

The simulation environment is built on top of the Simplescalar [15] Alpha toolkit. Simplescalar models an out-of-order machine and has been modified to support trace-level speculative multithreading. Table 1.2 shows the parameters of the baseline microarchitecture. The *Trace-Level Speculative Multithreaded Architecture* does not modify sizes of baseline structures: it just replicates for each thread unit the instruction window, reorder buffer and logical register mapping table. It also adds some new structures, as shown in Table 5.1.

Speculative data cache	1 KB, direct-mapped, 8-byte block
Verification engine	Up to 8 instructions verified per cycle. Memory instructions block verification if fail in L1. Number of additional instructions verified after average number to find an error is 16
Trace speculation engine	128 history table, 4-way set associative,.
Look-ahead buffer	128 entries

Table 5.1. Parameters of TSMA additional structures

The following Spec95 benchmarks were considered: *compress, gcc, go, li, ijpeg, m88ksim, perl and vortex* from the integer suite; and *applu, mgrid and turb3d* from the FP suite. The programs were compiled with the DEC C and F77 compilers with *-non_shared -O5* optimization flags (i.e, maximum optimization level). Each program was run with the test input set and statistics were collected for 125 million instructions after skipping an initial part of 250 million instructions (see Section 1.4 for further details of tools and benchmarks).

5.3.9.2. Analysis of Results

One of the main objectives of this section is to show that trace misspeculations cause minor penalties in the microarchitecture. We propose a simple mechanism for building traces and determining live outputs. Traces are built following a simple rule: a trace starts at a backward branch and terminates at the next backward branch. Traces are also terminated at calls and returns, and have a minimum and maximum size (8 and 64 instructions respectively). On the other hand, live-output values are predicted by means of a hybrid scheme composed of a stride predictor and a context-based predictor. This

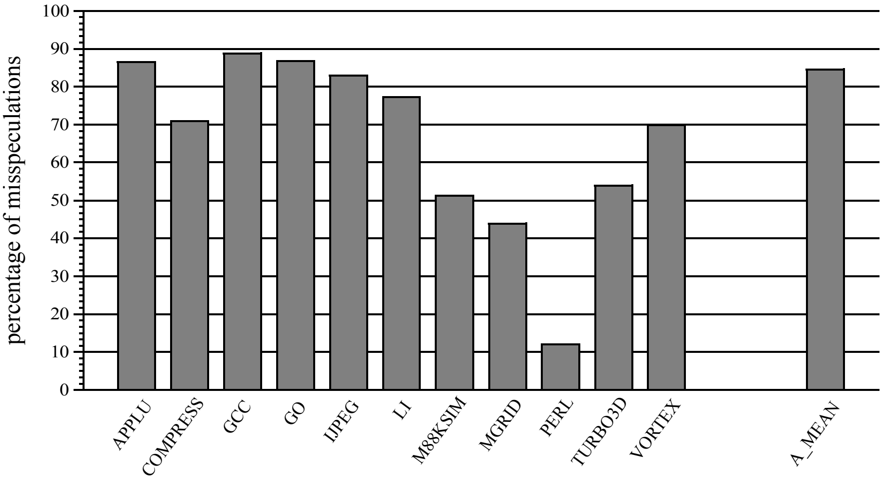


Figure 5.10. Percentage of misspeculations.

mechanism maintains, in each entry of the history table and in an ordered way, the last 9 dynamic instances of a trace. At prediction time, if the last instance of the trace appears among the previous 8 instances, the next trace is predicted. Otherwise, stride prediction is performed.

Figure 5.10 shows the percentage of misspeculations of the above mechanism. As we can see in Table 5.1, the capacity of prediction tables is relatively small. Note that this mechanism produces a huge percentage of misspeculations (close to 70% on average).

Figure 5.11 shows the percentage of speculated instructions. On average this is close to 40%, so speculation is relatively frequent. Note that the ideal scenario is when the percentage of speculated

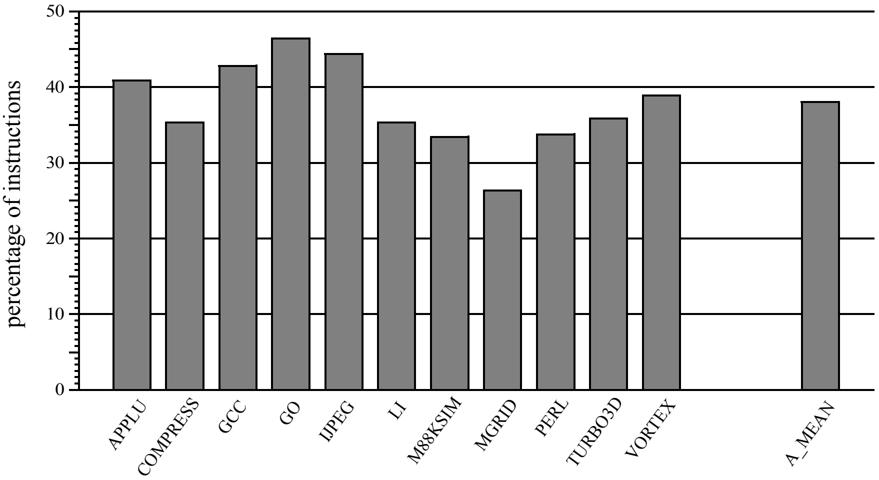


Figure 5.11. Percentage of predicted instructions.

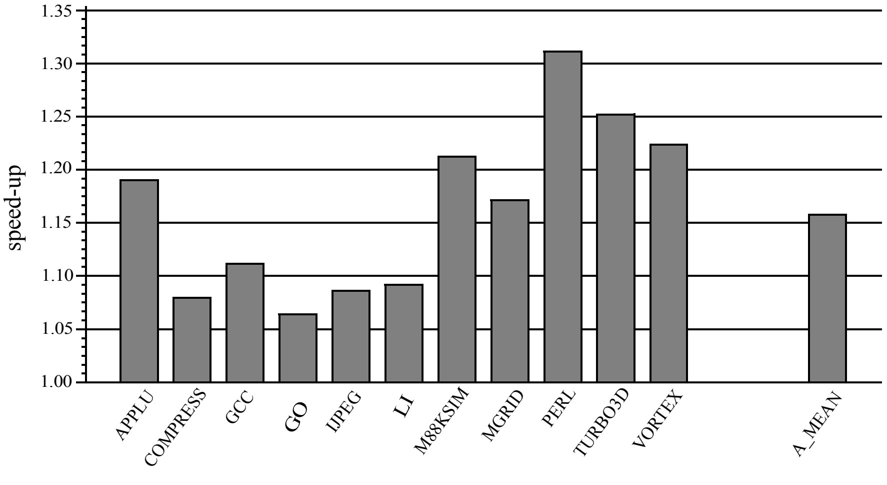


Figure 5.12. Speed-up.

instructions is around 50% because the microarchitecture has two threads and only allows a single unverified trace speculation at any given time (ST may only speculate a new trace when the look-ahead buffer is empty).

Figure 5.12 shows the speed-up obtained over the baseline model. Note that no slow-down is presented in any of the analysed benchmarks though the percentage of misspeculations is huge. In fact, significant speed-ups are obtained for most of them. Our results show that, despite speculating a small percentage of instructions correctly and misspeculating on average close to 70% of the traces, the average speed-up is 16%. These results demonstrate the tolerance to misspeculations of the proposed microarchitecture. Moreover, it encourages further work to develop more accurate trace prediction mechanisms. Some previous studies [29], [38], [49], [93] have demonstrated a significant potential for trace repeatability/predictability, which suggests that there may be effective schemes to significantly increase the accuracy of trace predictors.

Table 5.2 shows the results of other simulations. Statistics of speculated traces, average number of verified instructions before a misspeculation is detected, percentage of the time the look-ahead buffer is full and percentages of synchronizations are shown. Note that the number of total synchronizations is much smaller than the number of partial synchronizations. This is the key to minimizing misspeculation penalty. Our results also show that a look-ahead buffer with 128 entries almost never stalls ST. On the other hand, the average number of verified instructions before a misspeculation is encountered is very low. This number is dynamically computed by the processor and increased by a fixed amount (16 in our case) to dynamically set the number of instructions validated by the verification engine. In this way, the verification engine needs to verify 20 instructions to guarantee almost 96% of partial synchronizations. Table 5.2 also shows that there are no significant differences with respect to the baseline model in terms of the performance of the shared structures such as branch predictor and caches. Other results in the table refer to the average trace size and the number of live-output values.

Average size of speculated traces	30.31
Average number of live-output values	14.70
Average number of verified instructions after an error is encountered	4.26
Percentage of the time look-ahead buffer is full	0.01%
Percentage of total synchronizations	4.42%
Percentage of partial synchronizations	95.58%
Difference over the baseline mode of branch prediction hit rate	0.63%
Difference over the baseline mode of L1 data cache miss rate	0.21%
Difference over the baseline mode of instruction cache miss rate	0.44%

Table 5.2: Additional simulation results

5.4. Extensions to TSMA

We have shown that there are two important issues in trace-level speculation. The first involves the microarchitecture support for trace speculation, and the second involves trace selection and data value speculation techniques. This section covers both issues. First, we propose an advanced thread synchronization scheme based on the observation that there is a significant number of instructions whose control and data are independent of the mispredicted instruction. Second, we propose a static program profiling analysis for identifying candidate traces to be speculated.

5.4.1. Reducing Misspeculation Penalty in TSMA

TSMA uses special hardware called a *Verification Engine* that verifies instructions in the look-ahead buffer previously executed by ST. If source values of the instructions in the look-ahead buffer do not match the non-speculative architectural state, a thread synchronization is required in the original TSMA architecture. Basically, recovery actions involve flushing the ST pipeline and reverting to a safe point in the program. A critical aspect of the TSMA is to implement this recovery with the fewest performance penalties. Unfortunately, every thread synchronization throws away execution results of instructions that are independent of the mispredicted ones. This wastes useful computation and fetch bandwidth.

Note that a misspeculation in one instruction causes younger instructions to be discarded from the look-ahead buffer, though some may be correctly executed. Consider, for instance, a speculative trace in which just a single live-output value of the whole set is incorrectly predicted. Only the instructions dependent on the mispredicted one will be incorrectly executed by ST.

This section extends the previous TSMA microarchitecture with an advanced verification engine that significantly improves performance. This advanced engine reduces the number of thread synchronizations and the number of penalties due to misspeculations. The main idea is that it does not throw away execution results of instructions that are independent of the mispredicted speculation, which reduces the number of instructions fetched and executed again.

5.4.1.1. Thread Synchronization Analysis

We analyse the number of correctly executed instructions that are squashed from the look-ahead buffer when a thread synchronization has been fired. See Section 5.4.3.1 for details of the experimental framework.

Figure 5.13 shows the number of instructions that are squashed from the look-ahead buffer every time a thread synchronization was fired. Note that the number of discarded instructions is significant for all benchmarks. On average, up to 80 instructions were squashed from the look-ahead buffer in each thread synchronization irrespective of whether they were correctly or incorrectly executed.

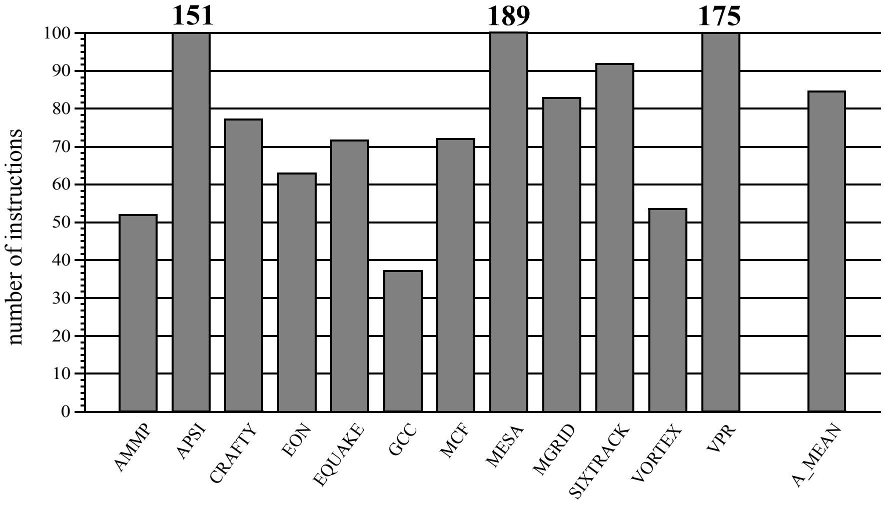


Figure 5.13. Number of squashed instructions from the LAB in each thread synchronization.

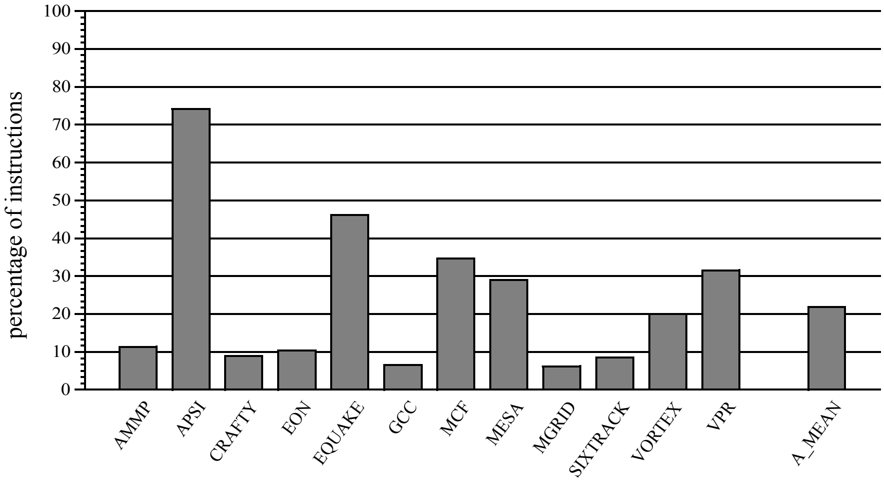


Figure 5.14. Percentage of squashed instructions from the LAB that were correctly executed.

Figure 5.14 shows that on average the percentage of squashed instructions from the look-ahead buffer that were correctly executed by ST was over 20%. If we combine this with the previous results, this means that on average 16 instructions that were correctly executed were discarded every time a thread synchronization was performed. We therefore decided to reconsider thread synchronizations to try and avoid this waste of activity and reduce the number of fetched and executed instructions.

5.4.1.2. Advanced Verification Engine

The conventional verification engine is in charge of validating speculated instructions. Together with NST, it maintains the speculative architectural state. Instructions to be validated are stored in the look-ahead buffer by ST. The verification involves comparing source values of the instruction with the non-speculative architectural state. If they match, the destination value of the instruction can be updated in the non-speculative architectural state (register file or memory). If they do not match, a thread synchronization is performed. Memory operations require special considerations. First, the effective address is verified. Then, store instructions update memory with the destination value. On the other hand, load instructions check whether the value of the destination register matches the non-speculative memory state. If it does, the destination value is committed to the register file. Note that this validation is fast and simple. Memory instructions stall verification if there is a data cache miss.

In this section, we modify the verification engine to improve the performance potential of the architecture. The underlying concept is based on the idea that a misspeculation in one instruction does not necessarily cause valid work from sequential younger computations to be aborted. Thread synchronization can therefore be delayed or even avoided. Below we describe how this advanced verification engine behaves depending on the type of instruction that is validated:

- **Branch instructions:** These operations do not have an explicit destination value. Implicitly, they modify the program counter according to the branch direction that is taken or not taken. The idea is to validate the branch target instead of the source values. So, if source values are incorrectly predicted but the direction of the branch is correct, a thread synchronization is not fired.
- **Load instructions:** First, the effective address is verified. If validation fails, the correct effective address is computed. Therefore, load instructions do not check whether the value of the destination register matches the non-speculative memory state. Simply, the destination value obtained from memory is committed to the register file. Note that an additional functional unit is required in order to compute the effective address.
- **Store instructions:** As with load instructions, the effective address is first verified. If validation fails, store instructions update memory with the destination value obtained from the non-speculative architectural state instead of with the value obtained from the instruction. Note that only one functional unit is required to compute the effective address.
- **Arithmetic instructions:** As with the conventional engine, the verification of arithmetic operations involves comparing the source operands of the instruction with the non-speculative architectural state. If they match, the destination value of the instruction can be committed to the register file. If they do not, the verification engine re-executes the instruction with values from the non-speculative state. In this case, verification is stalled and instructions after the re-executed one cannot be validated until the next cycle. Moreover, to maintain a high validation rate, this re-execution is only considered for single-cycle latency instructions. An additional functional unit is required in order to re-execute the instruction.

Note that only branch instructions with a wrong target and non-single-latency instructions with wrong source operands fire a synchronization.

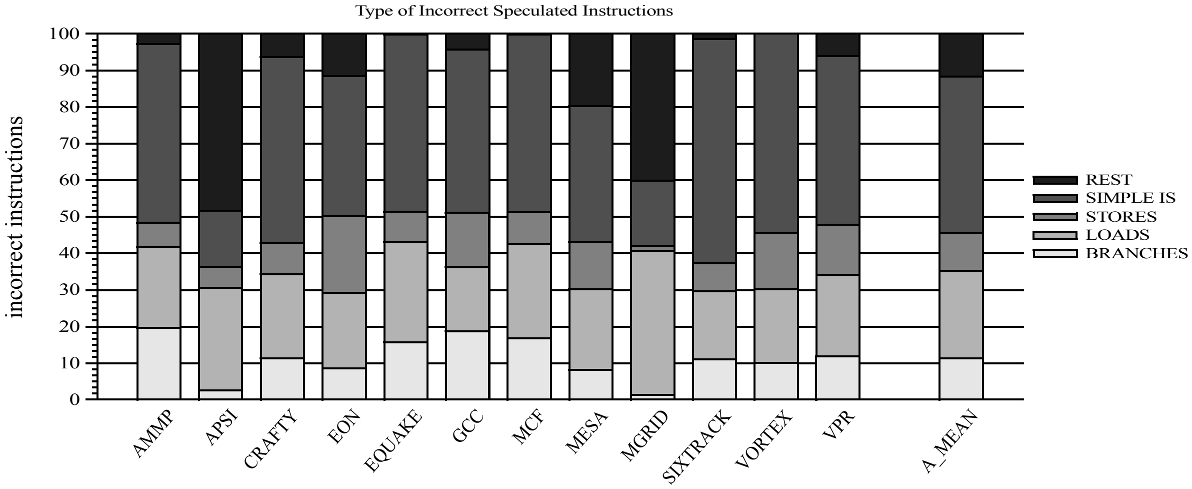


Figure 5.15. Type of incorrect speculated instructions

Figure 5.15 shows the breakdown of instructions in the look-ahead buffer that fail validation for the original validation engine. From bottom to top the categories are: branch instructions, load instructions, store instructions, instructions with single-cycle execution latency, and finally, other instructions. Note that branches, memory operations and instructions with a single-cycle latency account for 90% of the total incorrectly executed instructions. This means that there is a huge potential benefit for the advanced verification engine. Also, simulation results show that on average just 1% of the instructions inserted in the look-ahead buffer are incorrectly predicted. This suggests that the advanced verification engine may not need to re-execute many instructions, so the validation rate will not be greatly affected. Therefore, we assume that the number of functional units is not affected Finally, we also assume that the maximum number of instructions validated per cycle is the same and that no more than one instruction is re-executed per cycle.

5.4.2. Compiler Analysis to Support TSMA

Program profiling analysis is an effective technique for determining code regions whose live-output values may be reused or speculated at run-time [29],[34],[48],[69],[82]. In the following subsections we describe a profile-guided analysis for selecting the traces to be speculated by a TSMA processor.

5.4.2.1. Graph Construction

Trace selection is performed using an abstract data structure that is built from information obtained from the control flow graph, the data dependence graph and the predictability of values. The abstract data structure is a graph in which each node provides useful information for a static instruction. This information is obtained by running the test input set of the analysed benchmarks. The information maintained in each node or static instruction is:

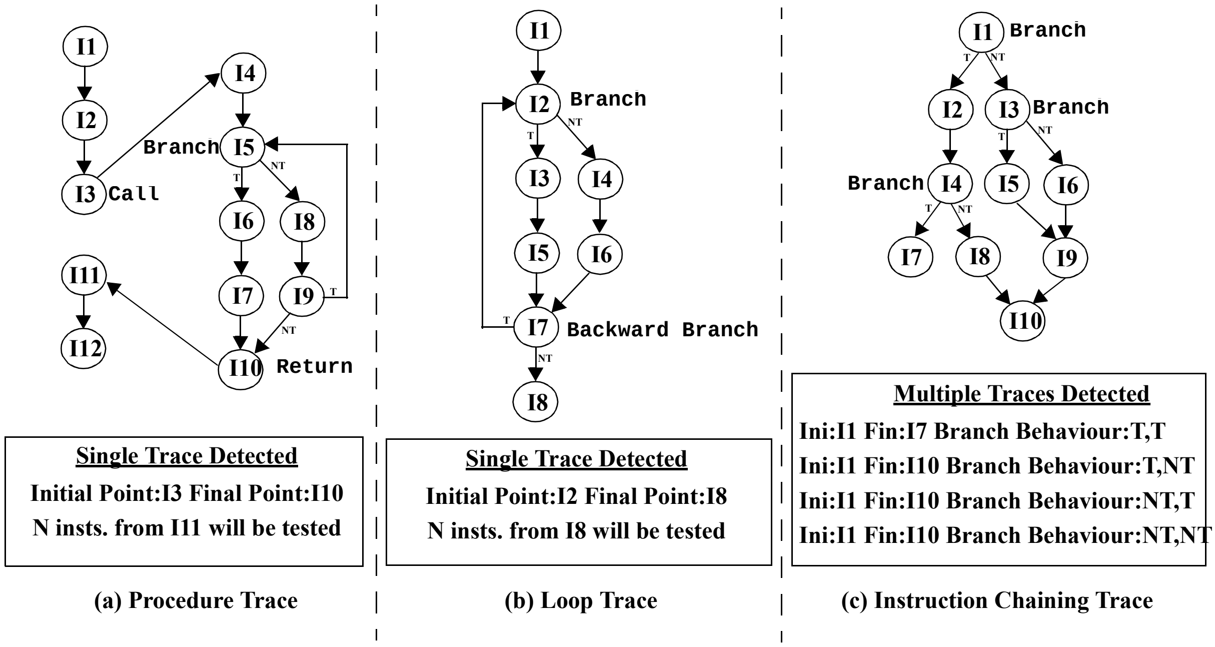
- The type of instruction and number of dynamic executions.
- The pointers to succeeding instructions in the dynamic execution stream with their corresponding frequencies (a single pointer in the case of arithmetic or memory instructions and multiple pointers in the case of conditional instructions and indirect jumps).
- The pointers to instructions that produce values that are consumed by the current instruction, pointers to instructions that consume values that are produced by the current instruction and their corresponding frequencies.
- The predictability of live-output values for different value predictors (stride and context-based predictors are considered).
- The percentage of times that the value produced by the current instruction is never used. Even with aggressive compiler optimizations, there are opportunities for removing code that may only be dead on a specific path [18].

5.4.2.2. Graph Analysis

Once the graph is built, several heuristics are applied to identify large regions of code that are suitable for traces. Several issues can be considered in the process of trace selection. These are related to the method for selecting the initial point of a trace, the final point and the predictability of live-output values.

A trace is considered a good candidate for speculation if the predictability of the live-output values achieves a certain threshold. Once live-output values are identified, their predictability has to be checked. Two types of statistics are analysed: prediction accuracy and utilization degree, which refers to the percentage of times that the value produced by an instruction is not consumed by any other instruction. If a live-output value does not achieve a certain threshold in terms of value predictability but is not frequently consumed, it is considered predictable.

The initial and final points of a trace are the other important issues to be determined. Note that misspeculations occur when live-output values are mispredicted or the actual control flow does not


I1
I2
I3
Call
I4
Branch
I5
T
NT
I6
I8
I7
I9
T
NT
I10
Return
I11
I12
Single Trace Detected
Initial Point:I3 Final Point:I10
N insts. from I11 will be tested
I1
I2
Branch
T
NT
I3
I4
I5
I6
T
I7
Backward Branch
NT
I8
Single Trace Detected
Initial Point:I2 Final Point:I8
N insts. from I8 will be tested
I1
Branch
T
NT
I2
I3
Branch
T
NT
Branch
I4
I5
I6
T
NT
I7
I8
I9
I10
Multiple Traces Detected
Ini:I1 Fin:I7 Branch Behaviour:T,T
Ini:I1 Fin:I10 Branch Behaviour:T,NT
Ini:I1 Fin:I10 Branch Behaviour:NT,T
Ini:I1 Fin:I10 Branch Behaviour:NT,NT

(a) Procedure Trace (b) Loop Trace (c) Instruction Chaining Trace

Figure 5.16. Trace Recognition Heuristics Examples

reach the trace termination point. The trace termination point selected must try to maximize the trace length and minimize control flow misspeculations. Below we describe three basic heuristics for building traces: *procedure trace*, *loop trace* and *instruction chaining trac*e.

5.4.2.3. Procedure Trace Heuristic

Procedures are potential sources for trace speculation. They are relatively frequent in a program execution and the computations that follow a subroutine return are fairly independent of the subroutine, except for return values and some memory locations. This means that just a few values should be predicted. Also, the control return point is normally reached despite the complexity of the control flow inside the procedure, which means that it is quite easy to predict the end of the trace.

This heuristic tries to identify some procedures as traces. In this way, a call instruction is marked as the initial point of a trace, and the return address is set as its final point. Figure 5.16.a shows an example of procedure trace detection. Note that the whole subroutine is considered as a single trace regardless of the control flow followed at each invocation.

To determine the predictability of live-output values, a given number of instructions belonging to all significant paths after the execution of the procedure are checked. A path is considered to be significant if its frequency of execution is above a certain threshold. For each instruction in a significant path it is checked whether any of its operands are produced by any instruction of the procedure. If this is the case, the predictability of the producer instructions is checked (through profiling) and if a certain threshold is not achieved, the trace is discarded. It has been empirically observed that there is no need to check too many instructions after the trace to identify good procedure traces. Moreover, binaries assumed in this analysis (Alpha under Unix) help this validation because only a couple of registers are used to return values other than memory locations.

5.4.2.4. Loop Trace Heuristic

Loops are a traditional source of parallelization and speculation. This heuristic considers the whole execution of a loop as a trace. The aim of this heuristic is to detect loops whose live-outputs after their whole execution are predictable (in fact, we are only concerned with outputs that are consumed relatively early).

This heuristic sets the initial point of a trace as the target of a backward branch and the final point of the trace is the fall-through instruction of the same backward branch. Figure 5.16.b shows an example of a loop trace. Note again that the whole loop is considered as a single trace regardless of the control flow followed at each invocation. As for subroutines, the predictability of the live-output values is checked by analysing a given number of instructions belonging to the significant paths after the execution of the loop. The trace is selected only if the predictability of the producer instructions is above a certain threshold.

5.4.2.5. Instruction Chaining Trace Heuristic

The aim of this heuristic is to identify large sequences of dynamic instructions, besides procedures and loops (and not necessarily contiguous in the static binary), with potential for speculation.

First, the initial point of a trace is selected. The taken and non-taken targets of all conditional branches are considered as initial points of a trace. The trace is then extended by adding successive instructions until a final point of the trace is reached. A trace reaches its final point when a new instruction already belongs to the same trace, the trace reaches a maximum size, or the new instruction is an indirect jump.

A trace in this case corresponds to a single control-flow path. Therefore, every time a conditional branch is found, a trace is split into two, one for each potential path. Figure 5.16.c shows an example of various traces with the same initial point. Each trace is identified by its initial point, its final point and the behaviour of the conditional branches within the trace. To limit the number of different traces with the same initial point, paths whose frequency of execution is below a given threshold are ignored.

Once a candidate trace has been identified, its live-output values are determined and its predictability is checked. For each live-output value, the highest value between its prediction accuracy and its utilization degree is chosen. The percentages of different live-outputs are then multiplied to estimate the probability that the trace is correctly speculated (a value is correctly speculated if it is correctly predicted or if it is not frequently used). If this probability is above a certain threshold, the trace is considered predictable and the process finishes. Otherwise, the final instruction of the trace is removed and the process starts again. This process is repeated until the trace achieves the defined threshold or the size of the trace reaches a minimum. Note that this process tries to select the longest predictable traces.

5.4.2.6. Hardware Modifications into the TSE

In this subsection we discuss the hardware modifications that are considered in the *Trace Speculation Engine* (TSE) to support the trace selection method based on compiler analysis.

We have shown how the off-line profile-guided analysis determines trace candidates to be speculated. These selected traces are communicated to the hardware at program loading time by filling the special hardware structure called trace table. We assume a simple 4-way set associative PC-indexed table with 128 entries. We have empirically observed that this number of entries and this degree of associativity leads to a good distribution of traces along the structure, and minimizes aliasing.

Now, each entry of the trace table contains the following information:

- *PcIni*: the initial program counter of the trace.
- *PcFin*: the final program counter of the trace.
- *BranchHist*: some bits that encode the history of some preceding branches.
- *LOValues*: value prediction information of N live-output values.
- *FreqCount*: a counter that determines the number of times that the trace has been found.

Live-output values are predicted by means of a hybrid scheme comprising a stride predictor and a context-based predictor. Based on the data in the trace table, the TSE is responsible for detecting initial and final points of a trace, maintaining value prediction information to compute live-output values, updating branch history and incrementing frequency counter. When the frequency counter of a trace reaches the maximum value, all frequency counters of traces with that initial program counter are initialized to zero.

As explained in Section 5.3.2, the TSE also has to determine trace speculation opportunities by scanning the current program counter of the speculative thread and checking it against the trace table. In this way, if the current PC is the beginning of a potentially predictable trace, the trace is speculated since the architecture is very tolerant to trace mispredictions. As we discussed in Section 5.4.2.5, multiple traces with the same initial program counter may be stored in the trace table. In this case, the trace predictor selects a trace from those with the same initial point based on the history of the preceding branches. If the current branch history matches that of a stored trace, this trace is selected for speculation. If no branch history matches the current one, the most frequent trace is selected from all those with the same initial program counter by checking frequency counters.

Finally, the TSE behaves in the conventional way when it determines that the current PC is the beginning of a potentially predictable trace. That is, it provides the final program counter for the fetch engine and generates some MOV instructions in order to initialize the live-outputs with the predicted values.

5.4.3. Performance Evaluation

In this section we discuss the experimental framework and analyse the performance of the advanced verification engine and the compiler analysis. First, we evaluate compiler analysis assuming the advanced verification engine in the TSMA. The advanced verification engine is then analysed by assuming the trace selection method based on compiler analysis.

5.4.3.1. Experimental Framework

The TSMA simulator is built on top of the Simplescalar Alpha toolkit [15]. The following Spec2000 benchmarks were considered: *crafty, eon, gcc, mcf, vortex,* and *vpr* from the integer suite and *ammp, apsi, equake, mesa, mgrid,* and *sixtrack* from the FP suite. The programs were compiled with the DEC C and F77 compilers with `-non_shared -O5` optimization flags (i.e. maximum optimization level). See Section 1.4 for further details of tools and benchmarks.

Table 1.2 shows the parameters of the baseline microarchitecture. The TSMA assumes the same sizes as the baseline configuration and for each thread unit replicates the instruction window, reorder buffer and logical register mapping table. It also adds some new structures (see Table 5.3). For the simulation, each program was run with the ref input set and statistics were collected for 250 million instructions after skipping initializations.

Speculative data cache	1 KB, direct-mapped, 8-byte block
Verification engine	Up to 8 instructions verified per cycle. Memory instructions stall verification for L1 misses. The conventional engine verifies an average number of 16 instructions to find an error. The advanced verification engine only re-executes a single instruction per cycle.
Trace speculation engine	128 history table, 4-way set associative. Hybrid predictor (stride + context)
Look-ahead buffer	512 entries

Table 5.3. Parameters of TSMA additional structures

Value predictors considered	stride and context
Minimum size of trace	16
Maximum size of trace	1024
Maximum number of live-output values	32
Minimum combined percentage to consider a set of live-output values predictable	25%
Minimum frequency to consider a path as significative	10%
Minimum accumulative frequency to consider multiple paths	1%

Table 5.4: Profiling analysis parameters

Table 5.4 shows the main parameters used in the program analysis phases. These values have been empirically checked to represent a good design point. First, it is important to minimize the number of misspeculations without losing speculation opportunities. In this way, the percentage of speculated traces is higher when the trace recognition heuristics are less conservative, but this also increases the percentage of misspeculation. However, the percentage of speculated traces and, therefore, the opportunities for speculation decrease when the trace recognition heuristics are more conservative. Second, it is important to maximize the number of speculated instructions and minimize the number of trace speculations. This means speculating traces as long as possible since every speculation introduces a minor penalty. Unfortunately, speculation accuracy decreases when the traces are larger because a huge number of live-output values have to be predicted. For the profiling data, each program was run with the test input set and statistics were collected for 250 million instructions after skipping initializations.

5.4.3.2. Analysis of Results of the Compiler Support

In this section, we analyse the performance of the compiler analysis by means of several figures that show the percentage of speculated instructions, the distribution of speculated traces, the speed-up obtained, the activity of both threads and the branch behaviour distribution.

Speculated instructions: Figure 5.17 shows the type of speculated instructions corresponding to instruction chaining traces, call traces and loop traces. Note that almost 45% of the speculated instructions are due to speculation of instruction chaining traces, 40% are due to speculation of call traces and the remaining 15% correspond to speculation of loop traces. Although the numbers of speculated call and loop traces are relatively small, they are significantly larger than instruction chaining traces. Table 5.2 shows that loop traces have an average trace size of 215.8 instructions, while instruction chaining traces have an average size of 36.4 instructions. Other statistics, such as the average number of live-output values and average numbers of branches within a trace, are also shown in Table 5.5

.

Average size of speculated traces per type (Instruction Chaining, Calls and Loops)	36.4, 97.3, 215.8
Average size of speculated traces	65.7
Average number of live-output values	16.4
Average number of branches within a trace (Instruction Chaining Heuristic)	5.3
Average number of traces with the same initial point (Instruction Chaining Heuristic)	1.57

Table 5.5: Additional simulation results

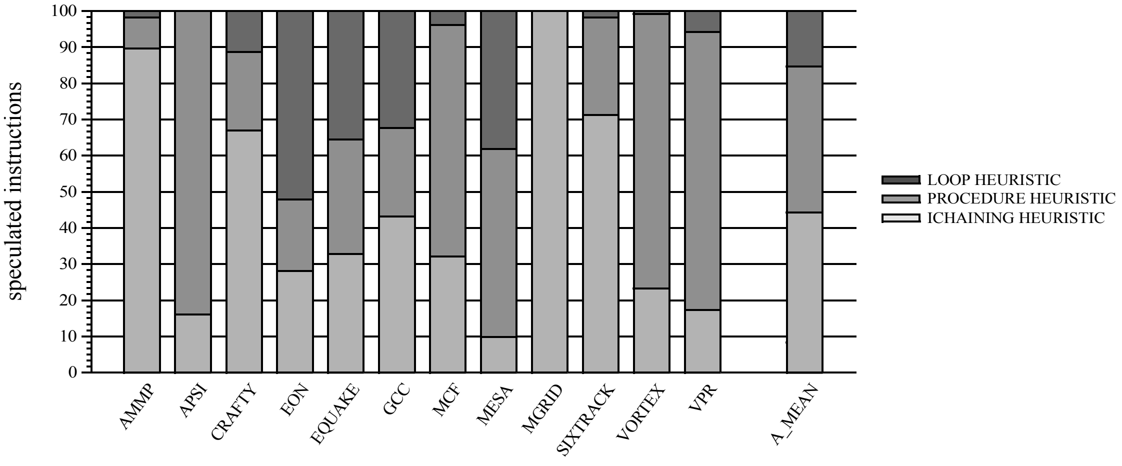


Figure 5.17. Type of speculated instructions

Note that the number of skipped instructions is larger when the traces are larger. However, this also implies a larger number of live-output values and therefore increases the probability of a live-output misprediction. The best performance depends on finding the best trade-off between the size of the traces and the predictability of their live-output values.

Distribution of speculated traces: A trace misspeculation can be produced by incorrectly predicting a live-output value or incorrectly predicting the final point of a trace. Also, the final point of a trace may be correctly predicted but paths between the initial and the final point of a trace may be incorrectly predicted. Note that this does not necessarily produce a misspeculation. For example, if-then-else structures that do not generate different live-output values may produce different traces with the same initial and final points. Figure 5.18 shows the distribution of speculated traces divided into four categories: (1) correct trace speculation and correct path speculation, (2) correct trace speculation despite incorrect path speculation, (3) incorrect trace speculation but correct path speculation, and finally (4) incorrect trace speculation and incorrect path speculation. We observe a significant percentage of correctly speculated traces (almost 70%). Note that the contribution of traces that do not produce misspeculation, even though the paths between the initial and the final point of the trace were not correctly predicted, is around 7%. On the other hand, the percentage of misspeculations is close to 30% (21% for correctly predicted paths and 9% for mispredicted paths or misprediction of the final point of a trace). These results confirm that the proposed mechanism for predicting paths and final points of traces provides a significant level of accuracy.

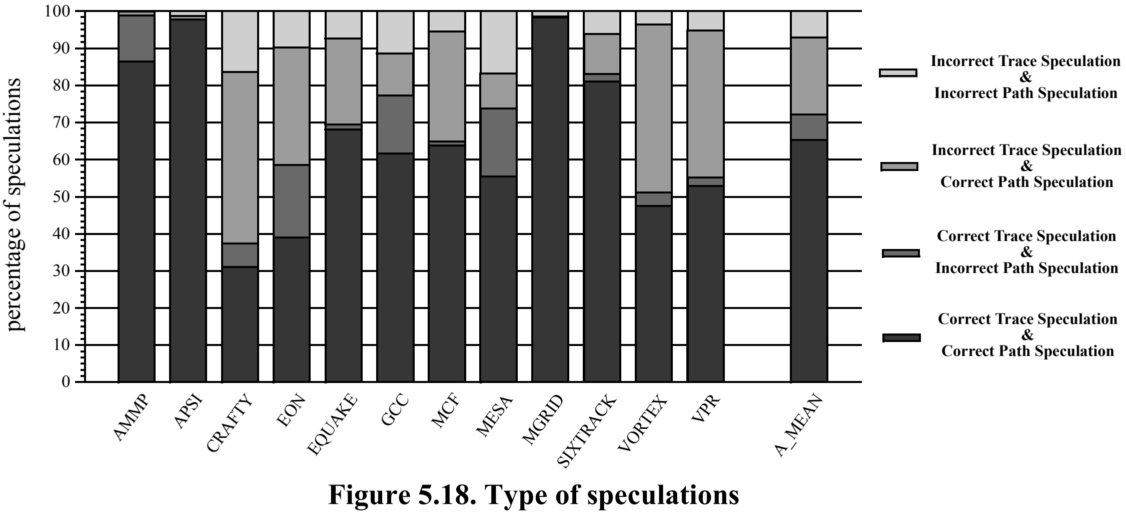


Figure 5.18. Type of speculations

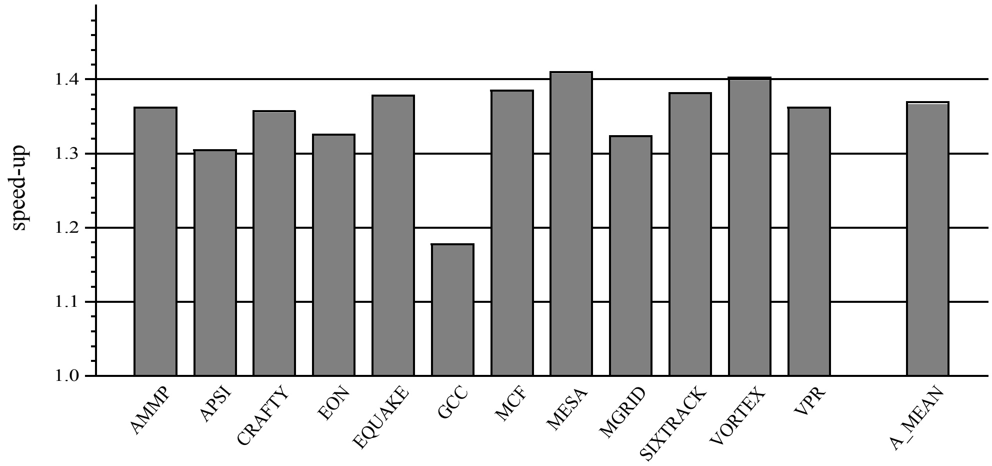


Figure 5.19. Speed-up

Speed-up: Figure 5.19 shows the speed-up obtained by the TSMA processor over the baseline superscalar configuration. Our results show that the average speed-up was almost 38% and very high speed-ups were achieved for all benchmarks. Note that significant speed-up was obtained despite misspeculating an average of 30% of the traces. These results also confirm that the proposed microarchitecture is tolerant to misspeculations and encourage further work to develop more aggressive trace prediction mechanisms.

Activity of both threads: Figure 5.20 and Figure 5.21 provide several statistics about the activity of the speculative thread and the non-speculative thread, respectively. The dark-grey bar in Figure 5.20 represents the percentage of time that ST can speculate but does not find a trace to be speculated, while

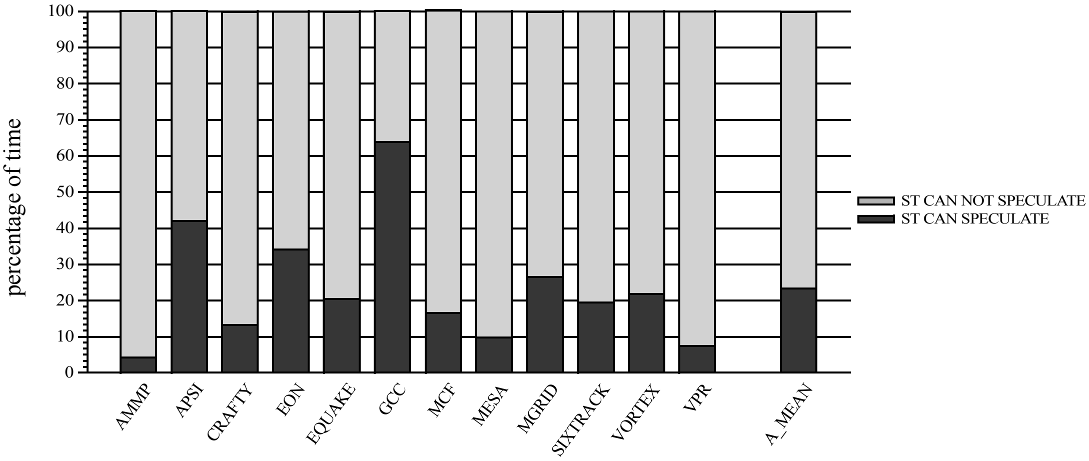


Figure 5.20. Type of cycles of the Speculative Thread (ST)

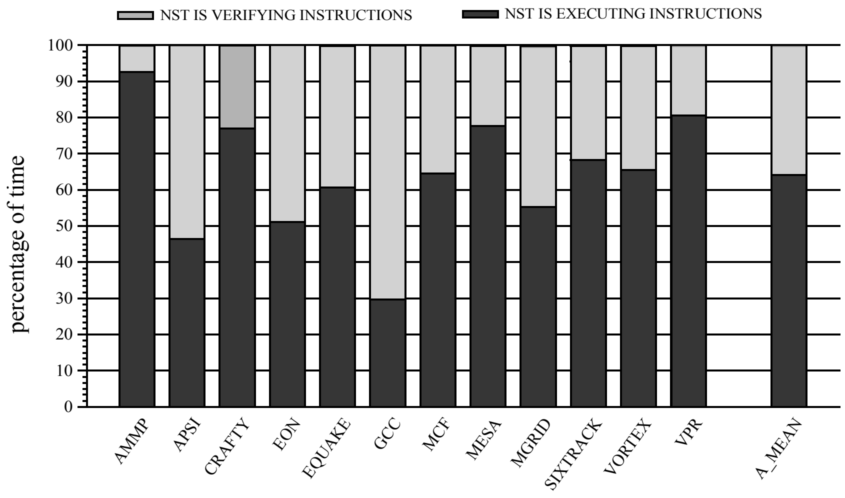


Figure 5.21. Type of cycles of the Non Speculative Thread (NST)

the light-grey bar represents the percentage of time that ST cannot speculate traces because NST is executing and verifying a speculated trace. Note that speculation may be performed only when NST catches up to ST. On average, almost 25% of the time the trace speculation engine did not communicate a trace speculation opportunity to the fetch engine because of this reason, which again confirms that performance may be improved by further analysing the impact of the trace size. Note that the ideal scenario is when the ST finds a point to speculate right after the NST has caught up to it. The dark-grey bar in Figure 5.21 represents the percentage of time that the NST executes traces speculated by ST, while the light-grey bar represents the percentage of time that the NST verifies instructions from the

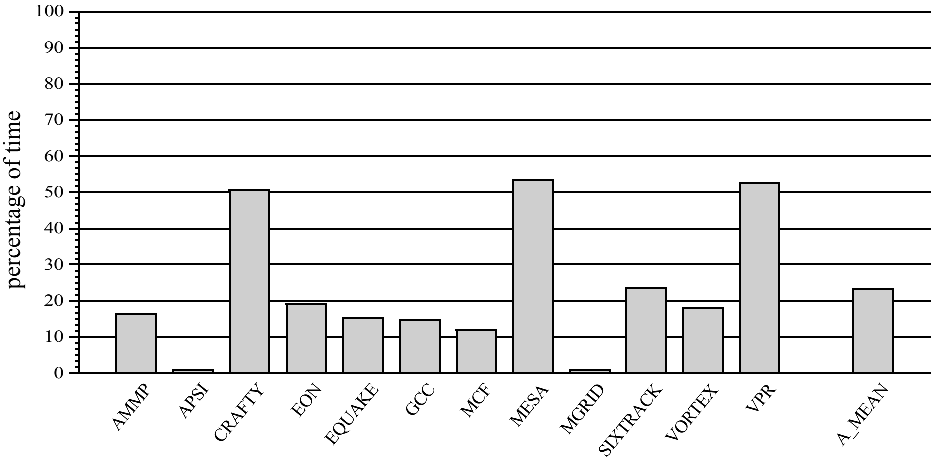


Figure 5.22. Useless cycles of the Speculative Thread (ST)

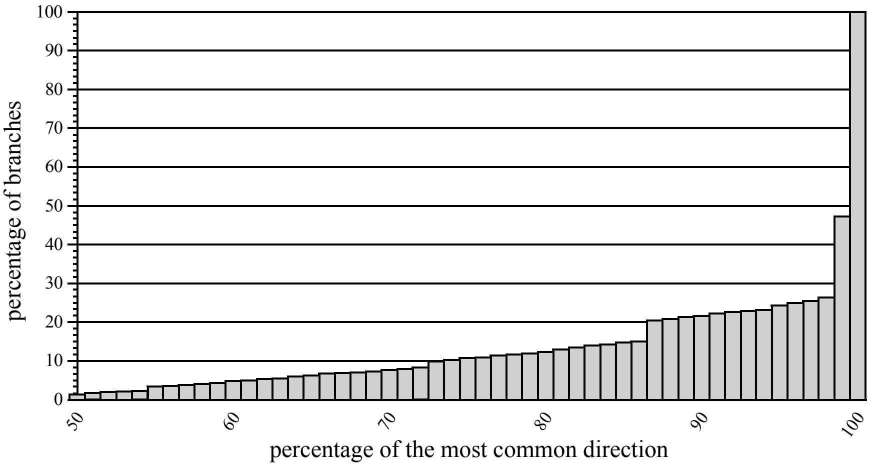


Figure 5.23. Branch Behaviour Distribution

look-ahead buffer. In general, more speculated instructions (see Figure 5.20) imply more time executing instructions for the NST and, since verifying instructions is faster than executing them, this follows a superlinear relation. Figure 5.22 shows the percentage of time that ST executes instructions beyond a misspeculation point. On average ST wastes up to 20% of the time executing instructions that will be discarded. Note that the ideal scenario would be when this percentage is negligible, which also implies a minimal number of trace misspeculations.

Branch behaviour distribution: Finally, we also observed that, despite the significant number of branches within the trace, the instruction chaining heuristic does not provide many traces with the same initial point (see Table 5.5). In this way, we studied branch behaviour and concluded that the majority of branches almost always take the same direction. Figure 5.23 shows the accumulated distribution of the branch behaviour for all the benchmarks used in this analysis. The X-axis represents the percentage of times that a branch takes the most common direction (50% means that the branch takes the taken and the not-taken paths the same number of times and 100% means that the branch always takes the same path). The Y-axis represents the accumulated number of dynamic branches. Note that almost 80% of the branches take the same direction more than 90% of the times. This result, combined with the parameters used for the analysis phase (listed in Table 5.4), significantly limits the number of traces with the same initial point.

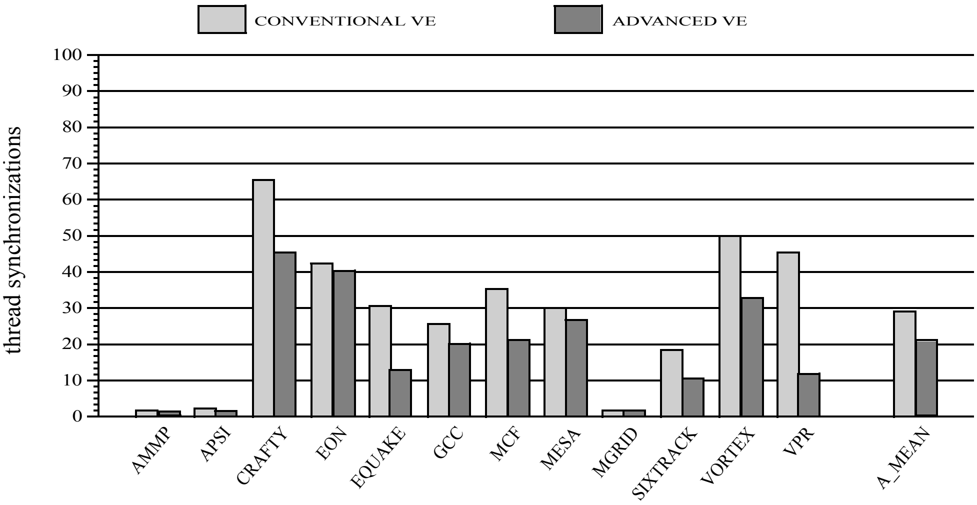


Figure 5.24. Percentage of thread synchronization

5.4.3.3. Analysis of Results of the Advanced Verification Engine

The main aim of this section is to show that the number of thread synchronizations is lower when the advanced verification engine is used.

Figure 5.24 plots the percentage of thread synchronizations against the number of trace speculations. Figure 5.25 shows the speed-up of TSMA over the baseline architecture. The first bar in each figure represents TSMA with the conventional engine and the second bar represents TSMA with the advanced verification engine.

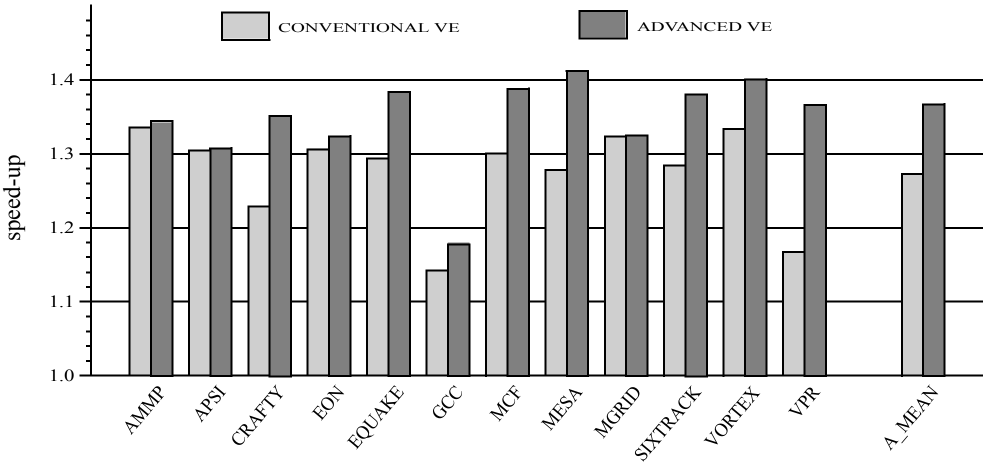


Figure 5.25. Speed-up

Our results show that the average speed-up for TSMA was 27 with the conventional verification engine. As expected, these speed-ups were significant for all the benchmarks despite a thread synchronization rate close to 30%.

On the other hand, the number of thread synchronizations was about 10% lower (from 30% to 20%) with the advanced verification engine than with the conventional scheme. Note that this engine did not always fire a thread synchronization to handle a miss trace speculation. It also provided a higher speed up (close to 38%), which implies that the average performance improvement was 9%. Note that the performance of most benchmarks improved significantly. Only benchmarks such as *ammp*, *apsi* or *mgrid,* whose misspeculation with the traditional verification engine was negligible, hardly improved since thread synchronizations were already low with the original verification engine.

These results demonstrate the tolerance to misspeculations of the proposed microarchitecture and encourage further work to develop more aggressive trace prediction mechanisms. Note that the advanced verification engine opens up a new area of investigation i.e. aggressive trace predictor mechanisms that do not need to accurately predict all live output values.

Figure 5.26 shows the reduction in executed instructions with the advanced verification engine. On average, this reduction is almost 8%. Note that this also reduces memory pressure since these instructions do not need to be fetched all together. Again, benchmarks whose percentage of synchronization was negligible experienced a very small reduction in executed instructions. For the other benchmarks, on the other hand, the number of executed instructions and the number of thread synchronizations decreased, which led to significant speed-ups.

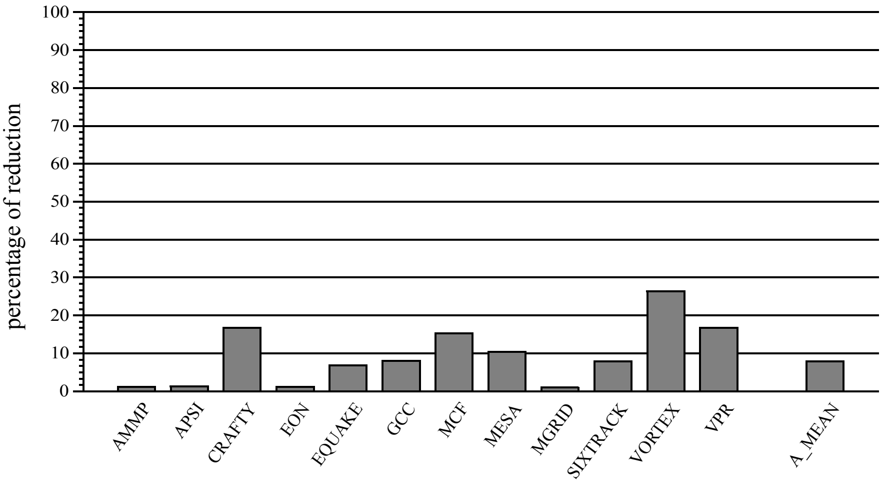


Figure 5.26. Reduction in executed instructions

5.5. Related Work

Rotenberg [93], [94] lay the basis of *Slipstream Processors* by proposing to run two partially redundant threads on either a symmetric multiprocessor or a simultaneous multithreading processor. His technique dynamically avoided the execution of non-essential computations of a program by creating a shorter version of the original program that removed ineffectual computations. One thread executes the shorter program and runs ahead of the thread that executes the full program. Communication between threads is done by means of a special structure called delay buffer. Recovery actions are done through the delay buffer which may cause a significant penalty. Koppanali and Rotenberg [61] examined in depth the slipstream component responsible for detecting past-ineffectual instructions. A detailed study of slipstream processors was done by Purser *et al* [90]. These authors discussed the sources of slipstream performance and its limitations. Slipstream processors were also studied by Sundaramoorthy *et al* [119] to combine the improvement of single program performance and the recover from transient hardware faults. Ibrahim *et al* [55] applied slipstream execution mode in a chip multiprocessor to enable the construction of a program-based view of the future to attack coherence, communication and synchronization overheads. Finally, Austin [5] proposed the DIVA checker architecture that manages dynamic verification to reduce the burden of verification in complex microprocessor designs. The idea was to manage two heterogeneous internal processors that execute the same program.

Several thread-level speculation techniques have been explored to exploit parallelism in general-purpose programs. *Speculative Multithreading* [3], [74] is a well-known technique based on the concurrent execution of speculative threads. *Simultaneous Multithreading* [122] allows independent threads to issue instructions to multiple functional units in a single cycle. *Multiple Path Execution* [2], [126] permits the speculative execution of multiple paths in parallel. S*imultaneous Subordinate Microthreading* [20] was proposed in order to execute subordinate threads that perform optimizations on a primary thread.

Other recent studies have also focused on the pre-execution of critical instructions by means of speculative threads. Collins *et al* [28] explored *Speculative Precomputation,* which uses a multithreaded architecture to improve the performance of single-threaded applications by pre-computing future memory accesses in available thread contexts. Roth and Sohi [97] proposed *Speculative Data-Driven Multithreading* as a general purpose mechanism to overcome the retirement stalls of mispredicted branches and loads that miss in cache. Zilles and Sohi [140] presented a technique to construct pieces of code (called slices) that contain the subset of the program that relates to critical instructions. They also proposed *Master/Slave Speculative Parallelization* [141] to improve the execution rate of sequential programs by parallelizing them speculatively for execution on a multiprocessor. Finally, a novel microarchitecture that dynamically allocates processor resources between a primary and a future thread was proposed by Balasubramonian *et al* [9]. The future thread executes instructions when the primary thread is limited by resource availability which therefore warms up certain microarchitectural structures.

Several studies [29],[38],[49],[94] have shown that programs usually have a significant degree of repeatability/predictability, which suggests that there may be effective schemes to significantly increase the accuracy of trace predictors. Oplinger *et al* [85] identified the potential sources of speculative parallelism in programs and concluded that a combination of loop and procedural speculation is a promising parallelization scheme for speculative thread-level parallel machines.

Value profiling has also been studied as a mechanism to assist value prediction or value reuse schemes. A value prediction scheme guided by value profiling is presented by Gabbay and Mendelson [34]. Connors and Hwu [29] and Huang and Lilja [48] proposed compiler-directed approaches for identifying code regions whose computation can be reused during dynamic execution. A code specialization approach that uses value profiling was presented by Muth *et al* [82]. Lin *et al* [69] proposed a compiler framework that includes analysis for speculative optimizations. These authors used profiling information and simple heuristics to supplement traditional non-speculative compile-time analysis.

Several techniques for reducing recovery penalties caused by speculative execution have been proposed. Instruction reissue, or selective squashing, was first proposed by Lipasti [70]. The idea was to retain instructions dependent on a predicted instruction in the issue queue until the prediction is validated. If the prediction is wrong, all dependent instructions are issued again. This technique trades the cost of squashing and re-fetching instructions for the cost of keeping instructions longer in the issue queue. A similar scheme that focused on load instructions was presented by González and González [39]. Tyson and Austin [123] thoroughly investigated the performance potential of the instruction reissue mechanism. Later, Sato and Arita [100] proposed a practical and simple implementation of instruction reissue based on a very slight modification of the register update unit.

Squash reuse has also been proposed as a way to reduce branch misspeculation penalty. This concept was first introduced by Sodani and Sohi [112]. These authors proposed a table-based technique for avoiding the execution of an instruction that has previously been executed with the same inputs. As well as squash reuse, they also covered general reuse. A different implementation based on a centralized window environment was proposed by Chou *et al* [23]. These authors also introduced the idea of dynamic control independence and showed how it can be detected and used in an out-of-order superscalar processor to reduce the branch misprediction penalty. Finally, Roth and Sohi [69] proposed register integration as a simple and efficient implementation of squash reuse.

5.6. Conclusions

In this chapter we have presented TSMA (*Trace-Level Speculative Multithreaded Architecture*). This novel microarchitecture is designed to exploit trace-level speculation with special emphasis on minimizing misspeculation penalties. Initial results based on a simple mechanism to build traces and predict its live outputs show that the microarchitecture is very tolerant to trace misspeculations. In fact, significant speed-up is presented in the majority of the analysed benchmarks in spite of the relatively poor accuracy of the assumed trace predictor. On average, a speed-up of 16% is achieved with a trace predictor that misses in 70% of the cases.

We have also proposed an advanced hardware technique to enhance TSMA processors. This hardware improvement focuses on the verification engine. The idea is to avoid the re-execution of instructions even when source values are incorrectly predicted. Instead of firing a thread synchronization that wastes useful computations, the correct value is re-computed and used to update the architectural state. The advanced engine reduces the number of thread synchronizations and the penalty due to misspeculations. This avoids discarding instructions that are independent of a mispredicted one, thus reducing the number of fetched and executed instructions and cutting energy consumption and contention for execution resources.

We have shown that, as well as the microarchitecture support for trace-level speculation, trace selection and data value speculation techniques are also an important design issue. In this way, we propose a profile guided analysis for identifying highly predictable, large traces to be speculated by a TSMA processor. We propose three basic heuristics to determine opportunities for speculation. This analysis substitutes the dynamic process of detecting speculative traces and their corresponding live-output values, which considerably reduces hardware complexity. Our simulation results show that these techniques achieve an average speed-up of almost 38%.

Future areas for investigation include generalising the architecture to multiple threads in order to perform sub-trace speculation during the validation of a trace that has been speculated. The relatively low penalty of misspeculations means that another area for future work is to investigate more aggressive speculation schemes.

“I see my path, but I don't know where it leads. Not knowing where I'm going is what inspires me to travel it”
Rosalia de Castro, Galician Writer, 1837-1885.

Chapter 6

CONCLUSIONS AND FUTURE WORK

This chapter summarizes the main conclusions of this work and outlines areas for future work.

6.1. Conclusions

The instruction-level parallelism of current microprocessors designs is seriously limited by control, data dependences and memory performance. This thesis has described several microarchitectural techniques that boost the execution of instructions to alleviate the serialization caused by data dependences and improve the memory system. All these techniques are based on exploiting the high percentage of repetitive behaviour exhibited by real-world programs. The techniques proposed for improving the memory system are based on exploiting the value repetition produced by store instructions, while the techniques proposed for boosting the execution of instructions are based on exploiting the computation repetition produced by all the instructions.

In Chapter 1, we first analysed repetition in programs and concluded that it was extremely high. On average, more than 80% of a program's computations have been done exactly in the past and over 90% of the values have been produced by an earlier computation. We have also shown that just a few computations and values are responsible for most of the repetitive behaviour of real-world programs.

In Chapter 2, we described a couple of microarchitectural techniques for exploiting repetition in data caches. First, we identified *Redundant Store Instructions* as memory writes that, because the value they write is equal to the existing value, do not change their contents. We analysed this value repetition in the same location of the memory hierarchy and presented a simple mechanism for reducing the memory traffic between levels of the memory hierarchy. In particular, we showed that we can achieve a significant memory traffic reduction between data cache levels with minimal hardware support. In this chapter we also showed the high degree of value repetition into several storage locations of conventional data caches at any given time. Based on this observation, we presented a novel data cache design, called *Non-Redundant Cache,* that avoids the replication of values. This novel data cache outperforms conventional caches in terms of power dissipation, access time and die area at the expense of a very minor increase in miss ratio.

We then applied three techniques that alleviates the serialization caused by data dependences by reducing the execution latency of instructions: *Instruction-Level Reuse*, *Trace-Level Reuse* and *Trace-Level Speculation*.

In Chapter 3, we analysed instruction-level reuse in detail and showed that it can benefit from computation repetition to boost the execution of instructions. We concluded that instruction-level reuse is abundant in all types of programs and can provide very large speed-ups for an ideal machine.

However, the benefits of instruction-level reuse are significant for lower latencies and minimal for higher latencies. In this chapter we also presented a novel reuse mechanism called *Redundant Computation Buffer*. This instruction-level reuse mechanism can exploit reuse due to both quasi-invariants and quasi-common subexpressions and also exhibits a low reuse latency. When we compared this mechanism with previous schemes, we found that it provided the greatest benefits in terms of execution time reduction.

In Chapter 4, we introduced the concept of trace-level reuse and demonstrated its great potential. Trace-level reuse is a data value reuse technique that handles dynamic sequences of instructions rather than single instructions. It is therefore based on the observation that several computations of the programs tend to be repetitive. We showed that trace-level reuse has many positive effects: a) it reduces the fetch bandwidth requirement by avoiding fetching instructions of reused traces; b) it increases the effective instruction window size by avoiding storing instructions of reused traces in the instruction window; and c) it has fewer overheads because it requires fewer operations per reused instruction. Finally, we addressed essential issues for integrating a trace-level reuse scheme into a superscalar processor, such as memory for storing previous traces, approaches for deciding which traces are useful, a mechanism for identifying reusable traces and how to update the processor state.

In Chapter 5 we covered trace-level speculation by proposing the *Trace-Level Speculative Multithreaded Architecture* (TSMA). This approach is supported by means of a couple of threads, a speculative thread and a non-speculative one, working cooperatively. This architecture has two main benefits: (a) no significant penalties are introduced in the presence of a misspeculation and (b) any type of trace predictor can work with this proposal. We have shown that, despite speculating a small percentage of traces correctly, the speed-up is significant. These results demonstrate the tolerance to misspeculations of the proposed microarchitecture. We also showed that, as well as the microarchitecture support for trace-level speculation, trace selection and data value speculation techniques are also important design issues. In this chapter, we proposed a profile-guided analysis for identifying highly predictable, large traces to be speculated by a TSMA processor. This analysis substitutes the dynamic process of detecting speculative traces and their corresponding live-output values, which considerably reduces hardware complexity. In this analysis, it is important to minimize the number of misspeculations without losing speculation opportunities and to maximize the number of speculated instructions while minimizing the number of trace speculations. Our simulation results show that the speed-up is high in most of the analysed benchmarks. Moreover, a detailed analysis of the results show that there is still room for improvement.

6.2. Future Work

Moore's law has held up over the past several decades and microprocessor transistor density has doubled every two years since 1965. Similar growth is projected for the near future. Thereby, designers will be able to fit more components onto a single die such as chip level multiprocessing and larger caches. However, this flexibility introduces certain problems, such as power consumption, on-chip interconnection delays and memory latency and bandwidth. Much effort should therefore be made to study methods that can mitigate such restrictions.

6.2.1. Value Repetition in Instruction Caches

Several studies have recently focused on the phenomenon of value replication in data caches. Surprisingly, instruction caches have not been considered to exploit the repetitive behaviour in caches. We believe that the codification of instructions also exhibits a high percentage of repetition that can be exploited in the same way as the *Non-Redundant Cache.* Moreover, the complexity of the mechanism can be reduced because write operations are not performed in instruction caches.

It would also be interesting to apply the *Redundant Store Mechanism* and the *Non-Redundant Cache* in all levels of the memory hierarchy. For instance, memory bandwidth can be significantly reduced not only by redundant stores, but also by inlined values that can be exchanged between levels in a compressed form.

6.2.2. Program Profiling to Support Data Value Reuse Schemes

It has been shown that just a few computations are responsible for the majority of the computations performed by conventional programs. An interesting focus of research would be to determine at compilation time the most frequent computations based on program profiling information. These computations and their corresponding results could be loaded into the reuse tables at the beginning of the execution of programs. This would considerably reduce hardware complexity and power consumption because the dynamic process of detecting instructions/traces and their corresponding live-output values could be avoided. Note that the profile-guided analysis for identifying highly predictable traces in trace-level speculative multithreaded architectures can be easily adapted to support data value reuse schemes at instruction and trace level.

6.2.3. Quasi-Common Traces

To the best of our knowledge, all trace-level reuse approaches are based on tables that are indexed by the program counter of the first instruction of the trace. This is because it is difficult to find in a program different sets of consecutive dynamic instructions that match exactly, especially when traces get bigger. It would be interesting to identify pieces of code or traces that are exactly the same in several parts of the program. Note that the *Redundant Computation Buffer* exploits reuse due to both quasi-invariants (produced by the same static instruction) and quasi-common subexpressions (produced by several static instructions). Therefore, the idea is not only to exploit quasi-invariant traces, but also quasi-common traces.

6.2.4. Value Prediction in TSMA

Simulation results for the novel verification engine of the *Trace-Level Speculative Multithreaded Architecture* show that it can significantly improve performance without increasing complexity. These results encourage further work to develop more aggressive speculation schemes based on the idea that not all live-output values need to be highly predictable. This is also motivated by the relatively low penalty of misspeculations achieved by the *Trace-Level Speculative Multithreaded Architecture*.

6.2.5. Multiple Speculations and Threads in TSMA

TSMA processor can simultaneously execute a couple of threads that cooperate to execute a sequential code. To ensure the correctness of the architectural state, ST may only speculate a new trace when the look-ahead buffer is empty. This means that TSMA has only a single unverified trace speculation at any given time. Future work includes to modify the architecture in order to allow multiple unverified traces while maintaining the relatively low penalty of misspeculations. Future areas for investigation also include generalising the architecture to multiple threads in order to perform sub-trace speculation during the validation of a trace that has been speculated.

APPENDIX

A. Trace-Level Reuse Theorems

In this appendix we introduce basic theorems related to trace-level reuse that may help to better understand the performance potential of this microarchitectural technique. We define 4 theorems and detail their proofs. These theorems help to explain the limits of trace-level reusability described in Section 4.2.2.

A.1. Definition of Theorems

Theorem TLR1. Let T be a trace composed of the sequence of dynamic instructions $<i_1, i_2, ..., i_n>$. If T is reusable, then i_k is reusable for every $k \in [1,n]$.

Theorem TLR2. Let T be a trace composed of the sequence of dynamic instructions $<i_1, i_2, ..., i_n>$. If i_k is reusable for every $k \in [1,n]$, then T is not necessarily reusable.

Theorem TLR3 (Generalization of Theorem TLR1). Let T be a trace composed of a sequence of traces $<t_1, t_2, ..., t_n>$. If T is reusable, then t_k is reusable for every $k \in [1,n]$.

Theorem TLR4 (Generalization of Theorem TLR2). Let T be a trace composed of the sequence of traces $<t_1, t_2, ..., t_n>$. If t_k is reusable for every $k \in [1,n]$, then T is not necessarily reusable.

A.2. Definition of Terms

Before beginning the proofs we need to define a number of terms:

- An *input* of a trace *T* is a register, condition code or memory location that is read and has not been written before in that trace. An *output* of a trace *T* is a register, condition code or memory location that has been written in that trace. Let *IL(T)* be the sequence of *input* storage locations of trace *T*. Notice that *IL(T)* is a sequence and not a set. The order of the sequence is given by the order in which the inputs are read. Let *OL(T)* be the sequence of *output* storage locations of trace *T*. The order of the sequence is given by the order in which the outputs are written. Let *IV(T)* be the sequence of *input* values of trace *T*, in the order in which they are read. Let *OV(T)* be the sequence of *output* values of trace T, in the order in which they are written.

- If A and B are two sequences, we say that $A \subseteq B$ if A is a subsequence of B. Moreover, $A \cup B$ refers to any sequence that is composed of the elements of A and B, no matter what the order of the elements is. Different dynamic instances of the same trace will be denoted using the same symbol to refer to the trace, with a superscript corresponding to the dynamic execution order. Note that different instances of the same trace will always have the same input/output registers but may have different input/output memory locations.

- If a trace T is reusable, it must happen that $IL(T^i) = IL(T^j)$ and $IV(T^i) = IV(T^j)$ for some $j < i$. This obviously implies that $OL(T^i) = OL(T^j)$ and $OV(T^i) = OV(T^j)$, i.e. if the inputs are the same and have the same value, the outputs will also be the same and will have the same values.

A.3. Proof of Theorems

In this section we present the proof of the theorems described above. In fact, we will prove the general formulation of theorem TLR1 and TLR2, in which a trace is considered as a sequence of consecutive smaller traces. Note that theorems TLR1 and TLR2 are special cases in which the size of every smaller trace is one instruction. We shall now detail the proofs of theorems TLR3 and TLR4.

Proof of Theorem TLR3 (Generalization of Theorem TLR1): If T^i is reusable, then $IL(T^i) = IL(T^j)$ and $IV(T^i) = IV(T^j)$ for some $j < i$. Notice that $IL(t_1^i) \subseteq IL(T^i) = IL(T^j)$, which implies that $IL(t_1^i) = IL(t_1^j)$ and $IV(t_1^i) = IV(t_1^j)$. Therefore, $OL(t_1^i) = OL(t_1^j)$ and $OV(t_1^i) = OV(t_1^j)$. Thus, t_1 is reusable.

Notice that $IL(t_2^i) \subseteq IL(T^i) \cup OL(t_1^i) = IL(T^j) \cup OL(t_1^j)$. Since $IL(t_2^j) \subseteq IL(T^j) \cup OL(t_1^j)$ and $OV(t_1^i) = OV(t_1^j)$, we have that $IL(t_2^i) = IL(t_2^j)$ and $IV(t_2^i) = IV(t_2^j)$. Thus, t_2 is reusable, i.e. $OL(t_2^i) = OL(t_2^j)$ and $OV(t_2^i) = OV(t_2^j)$.

In general, we can prove that t_{k+1} is reusable provided that t_i is reusable for any $i=1..k$. Notice that $IL(t_{k+1}^i) \subseteq IL(T^i) \cup OL(t_k^i) \cup OL(t_{k-1}^i) \cup ... \cup OL(t_1^i) = IL(T^j) \cup OL(t_k^j) \cup OL(t_{k-1}^j) \cup ... \cup OL(t_1^j)$. Thus, $IL(t_{k+1}^i) = IL(t_{k+1}^j)$ and $IV(t_{k+1}^i) = IV(t_{k+1}^j)$, which means that t_{k+1} is reusable.

Proof of Theorem TLR4 (Generalization of Theorem TLR2): If $t_1^i, t_2^i, ..., t_n^i$ are reusable, then the inputs of each of them are equal to those of some previous execution. That is, $IV(t_1^i) = IV(t_1^{j1})$, $IV(t_2^i) = IV(t_2^{j2})$, ..., $IV(t_n^i) = IV(t_n^{jn})$, but $j1, j2, ..., jn$ may be different. Therefore, $IV(T^i) \subseteq IV(t_1^i) \cup IV(t_2^i) \cup ... \cup IV(t_n^i) = IV(t_1^{j1}) \cup IV(t_2^{j2}) \cup ... \cup IV(t_n^{jn})$, but $IV(T^i)$ may be different from $IV(T^j)$ for every $j < i$, so T^i may be non-reusable.

“As I’m sure you know, the beginning is hard, reading and more reading, so no papers yet”
A beginner PhD student, 1998.

References

[1] H. Abelson and G.J. Sussman, “Structure and Interpretation of Computer Programs”, McGraw Hill, New York, 1985.

[2] P. S. Ahuja, K. Skadron, M. Martonosi and D. W. Clark, “Multipath Execution: Opportunities and Limits”, *In Proceedings of the International Symposium on Supercomputing,* July 1998.

[3] H. Akkary and M. Driscoll, “A Dynamic Multithreading Processor”, I*n Proceedings of the 31st Annual International Symposium on Microarchitecture*, December 1998.

[4] C. Aliagas, C. Molina, M. García, A. González and J. Tubella, “Value Compression to Reduce Power in Data Caches”, *In Proceedings of the 9th International Conference on Parallel and Distributed Computing* (Euro-Par), August 2003.

[5] T.M. Austin, “DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design”, *In Proceedings of the 32nd Annual International Symposium on Microarchitecture*, November 1999.

[6] T.M. Austin and G.S. Sohi, “Dynamic Dependence Analysis of Ordinary Programs”, In *Proceedings of the 19th International Symposium on Computer Architecture,* May 1992.

[7] M. Azam, P. Franzon and W. Liu, “Low Power Data Processing by Elimination of Redundant Computations”, In *Proceedings of the International Symposium on Low Power Electronics and Design*, August 1997.

[8] S. Balakrishnan and G. S. Sohi, ”Exploiting Value Locality in Physical Register Files”, *In Proceedings of the 36th International Symposium on Microarchitecture*, December 2003.

[9] R. Balasubramonian, S. Dwarkadas and D. Albonesi, “Dynamically Allocating Processor Resources between Nearby and Distant ILP”, *In Proceedings of the 28th International Symposium on Computer Architecture,* June 2001.

[10] G. B. Bell, K. M. Lepak and M. H. Lipasti. “Characterization of Silent Stores”, *In Proceedings of the International Conference on Parallel Architectuers and Compilation Techniques*, October 2000.

[11] J. L. Bentley. “Writing Efficient Programs”, Prentice Hall, Englewood Cliffs, New Jersey, 1982.

[12] B. Black, B. Rychlik, and J. Shen, “The Block-Based Trace Cache”. *In Proceedings of the 26th International Symposium on Computer Architecture*, May 1999.

[13] D. Brooks and M. Martonosi, “Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance”, *In Proceedings of the 5th* I*nternational Symposium on High-Performance Computer Architecture,* January 1999.

[14] D. Brooks, V. Tiwari and M. Martonosi, “Watch: A Framework for Architectural-Level Power Analysis and Optimization”. *In Proceedings of the 27th International Symposium on Computer Architecture*, May 2000.

[15] D. Burger, T. M. Austin and S. Bennet, “Evaluating Future Microprocessors: The SimpleScalar Tool Set”. Technical Report CS-TR-96-1308. University of Wisconsin, July 1996.

[16] D. Burger, J. R. Goodman, and A. Kägi, “Quantifying Memory Bandwidth Limitations of Current and Future Microprocessors”, *In Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996.

[17] D. Burger, A. Kägi and J. R. Goodman, “The Declining Effectiveness of Dynamic Caching for General Purpose Microprocessors“, Technical Report 1261, Computer Sciences Departament, University of Wisconsin, Madison, WI, January 1995.

[18] J. A. Butts and G.S. Sohi, “Dynamic Dead-Instruction Detection and Elimination”, *In Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, October 2002.*

[19] R. Canal, A. González and J. E. Smith, “Very Low Power Pipelines using Significance Compression”, *In Proceedings of the* 33rd International Symposium on Microarchitecture, December 2000.

[20] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt, “Simultaneous Subordinate Microthreading (SSMT)”, *In Proceedings of the 26th International Symposium on Computer Architecture,* May 1999.

[21] T. F. Chen and J. L. Baer, “A Performance Study of Software and Hardware Data Prefetching Schemes”, *In Proceedings of the 21st International Symposium on Computer Architecture,* April 1994.

[22] Y. Choi, J. J. Yi, J. Huang, and D. J. Lilja, “Improving Value Prediction by Exploiting Both Operand and Output Value Locality”, Laboratory for Advanced Research in Computing Technology and Compilers Technical Report No. ARCTiC 00-09, July, 2000.

[23] Y. Chou, J. Fung, J. Shen, “Reducing Branch Misprediction Penalties Via Dynamic Control Independence Detection”, *In Proceedings of the International Conference on Supercomputing*, June 1999.

[24] D. Citron, D. G. Feitelson and Larry Rudolph, “Accelerating Multi-Media Processing by Implementing Memoing in Multiplication and Division Units”, In *Proceedings of the 7th International Conference on Architecture Support for Programming Languages and Operating Systems*, October 1998.

[25] D. Citron and D. G. Feitelson, “Look it Up or Do the Math: An Energy, Area, and Timing Analysis of Instruction Reuse and Memoization”. *In Proceedings of the Workshop on Power-Aware Computer Systems*, December 2003.

[26] D. Citron and D. G. Feitelson, “Revisiting Instruction Level Reuse”. *In Proceedings of the Workshop on Duplicating, Deconstructing, and Debunking*, May 2002.

[27] J. Collins, S. Sair, B. Calder and D.M. Tullsen, “Pointer Cache Assisted Prefetching”, *In Proceedings of the 35th Annual International Symposium on Microarchitecture*, November 2002.

[28] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery and J. Shen, “Speculative Precomputation: Long-range Prefetching of Delinquent Loads”, *In Proceedings of the 28th International Symposium on Computer Architecture*, July 2001.

[29] D. A. Connors and W. W. Hwu, “Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results“, *In Proceedings of the 32nd International Symposium on Microarchitecture*, November 1999.

[30] D. A. Connors, H. C. Hunter, B. C. Cheng, and W. W. Hwu, “Hardware Support for Dynamic Activation of Compiler-Directed Computation Reuse”, *In Proceedings of the 9th International Conference on Architecture Support for Programming Languages and Operating Systems*, November 2000.

[31] A. T. Costa, F. M. G. Franca and E. M. Chaves Filho, “The Dynamic Trace Memoization Reuse Technique", *In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, October 2000.

[32] Digital Equipment Corporation, “Alpha 21164 Microprocessor Hardware Reference Manual”, 1995.

[33] O. Ergin, D. Balkan, K. Ghose, D. Ponomarev, “Register Packing: Exploiting Narrow Width Operands for Reducing Register File Pressure”, *In Proceedings of the 37th International Symposium on Microarchitecture,* December 2004.

[34] F. Gabbay and A. Mendelson, “Can Program Profiling support Value Prediction”, *In proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997.

[35] F. Gabbay and A. Mendelson. “Speculative Execution based on Value Prediction”. Technical Report EE Department TR 1080, Technion - Israel Institute of Technology, November 1996.

[36] F. Gabbay and A. Mendelson. “Using Value Prediction to Increase the Power of Speculative Execution Hardware”, *ACM Transaction on Computer Systems (TOCS)*, August 1998.

[37] A. González, J. Tubella and C. Molina, “The Performance Potential of Data Value Reuse”, *Technical Report UPC-DAC-1998-23, Universitat Politècnica de Catalunya*, September 1998.

[38] A. González, J. Tubella and C. Molina, “Trace Level Reuse”, *In Proceedings of the International Conference on Parallel Processign*, September 1999.

[39] J. González and A. González, “Speculative Execution via Address Prediction and Data Prefetching”, *In proceedings of the 11th International Conference on Supercomputing,* July 1997.

[40] R. Gonzalez, A. Cristal, A. Veidenbaum, and M. Valero, "A Content Aware Register File Organization”, *In Proceedings of the 31th International Symposium on Computer Architecture*, June 2004.

[41] J. R. Goodman and W. C. Hsu,“ On the Use of Registers vs. Cache to Minimize Memory Traffic”, *In Proceedings of the 13th International Symposium on Computer Architecture,* June 1986.

[42] J. R. Goodman, “Using Cache Memory to Reduce Processor Memory Traffic” *In Proceedings of the 10th International Symposium on Computer Architecture,* June 1983.

[43] M. Gowan, L. Biro and D. Jackson, “Power Considerations in the Design of the Alpha 21264 Microprocessor”. *In Proceedings of the 35th Annual Conference on Design Automation*, June 1998.

[44] L. Gwennap, “Digital 21264 Sets New Standard”, *Microprocessor Report*, vol. 10, no. 14, Oct. 1996.

[45] E. G. Hallnor and S. K. Reinhardt, “A Compressed Memory Hierarchy using an Indirect Index Cache”, *In Proceedings of the* 3rd Workshop on Memory Performance Issues, June 2004.

[46] S. Harbison, "An Architectural Alternative to Optimizing Compilers", *In Proceedings of the 1st International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 57--65, March 1982.

[47] J. L. Hennessy and D. A. Patterson, “Computer Architecture: A Quantitave Approach”, Third Edition. Morgan Kaufmann Publishers, San Francisco 2002.

[48] J. Huang and D. J. Lilja, “Extending Value Reuse to Basic Blocks with Compiler Support”, IEEE Transactions on Computers 2000.

[49] J. Huang and D. J. Lilja, "Exploiting Basic Block Value Locality with Block Reuse", In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, January 1999.

[50] J. Huang and D. J. Lilja, "Exploring Sub-Block Value Reuse for Superscalar Processors", In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*, October, 2000.

[51] J. Huang and D. J. Lilja, "Improving Instruction-Level Parallelism by Exploiting Global Value Locality", High-Performance Parallel Computing Research Group Technical Report No. HPPC-98-12, October 1998.

[52] J. Huh, D.C. Burger, and S.W. Keckler, "Exploring the Design Space of Future CMPs". *In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[53] http://www.sandpile.org/, "The World's Leading Source for Pure Technical x86 Processor Information".

[54] http://www.spec.org/, "The Standar Performance Evaluation Corporation".

[55] K. Z. Ibrahim, G. T. Byrd, and E. Rotenberg, "Slipstream Execution Mode for CMP-Based Multiprocessors", *In Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, February 2003.

[56] L. K. John, "More on Finding a Single Number to Indicate Overall Performance of a Benchmark Suite", *ACM Computer Architecture News, Vol. 32, No. 1*, March 2004.

[57] N. P. Jouppi, *"*Cache Write Policies and Performance", *In Proceedings of the 20th International Symposium on Computer Architecture*, May 1993.

[58] N. P. Jouppi, *"*Improving Direct-Mapped Cache Performance by the Addition of Small Fully Associative Cache and Prefetch Buffers", *In Proceedings of the 17th International Symposium on Microarchitecture*, May 1990.

[59] S. Jourdan, R. Ronen, M. Bekerman, B. Shormar and A. Yoaz, “A Novel Renaming Scheme to Exploit Value Temporal Locality through Physical Register Reuse and Unification”. *In Procceedings of 31st Annual International Symposium on Microarchitecture*, November 1998.

[60] I. Kim and M. Lipasti, “Implementing Optimizations at Decode Time”, *In Proceedings of the 29th International Symposium on Computer Architecture*, 2002.

[61] J. J. Koppanalil and E. Rotenberg, “A Simple Mechanism for Detecting Ineffectual Instructions in Slipstream Processors”, *IEEE Transactions on Computers,* April 2004.

[62] M. S. Lam and R. P. Wilson, “Limits of Control Flow on Parallelism”, *In Proceedings of the 19th Annual International Symposium on Computer Architecture*, May, 1992.

[63] S. Y. Larin, “Exploiting Program Redundancy to Improve Performance, Cost and Power Consumption in Embedded Systems”, Ph.D. Thesis, ECE Department, North Carolina State University, Raleigh, North Carolina, August 2000.

[64] J. S. Lee, W. K. Hong, and S. D. Kim, "An On-Chip Cache Compression Technique to Reduce Decompression Overhead and Design Complexity," *Journal of Systems Architecture, vol. 46,* December 2000.

[65] K. M. Lepak and M. H. Lipasti, “On the Value Locality of Store Instructions”, *In Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

[66] K. M. Lepak and M. H. Lipasti. “Silent Stores for Free”, *In proceedings of the 33rd Annual International Symposium on Microarchitecture*, December 2000.

[67] K. M. Lepak and M. H. Lipasti, “Temporally Silent Stores”, *In Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[68] C. Liao and J. Shieh, “Exploiting Speculative Reuse Using Value Prediction”, *In Proceedings of the 7th Asia-Pacific Conference on Computer Systems Architecture*, Jannuary 2002.

[69] J. Lin, T. Chen, W. C. Hsu, P. C. Yew, R. D. C. Ju, T. F. Ngai, S. Chan, “A Compiler Framework for Speculative Analysis and Optimizations”, *In Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2003.

[70] M. H. Lipasti, “Value Locality and Speculative Execution”, Ph.D. Dissertation, Department of Electrical and Computer Engineering, Carnegie Mellon University, April 1997.

[71] M. H. Lipasti and J. P. Shen. “Exceeding the Dataflow Limit Via Value Prediction”, *In Proceedings of 29th International Symposium on Microarchitecture*, December 1996.

[72] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. “Value Locality and Load Value Prediction”, *In Proceedings of 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, September 1996.

[73] G. N. B. Manoharan and J. S. Narayanan, “Dynamic Exploitation of Redundancy in Programs Using Value Prediction and Instruction Reuse”, *In Proceedings of the 10th International Conference on High Performance Computing*, December 2003.

[74] P. Marcuello, J. Tubella and A. González, “Value Prediction for Speculative Multithreaded Architectures”, *In Proceedings of the 32nd International Symposium on Microarchitecture*, November 1999.

[75] D. Michie, “Memo Functions and Machine Learning”, *Nature, Vol 218*, April 1968.

[76] C. Molina, A. González and J. Tubella, “Compiler Analysis for Trace-Level Speculative Multithreaded Architectures”, *In Proceedings of the 9th Annual Workshop on Interaction between Compilers and Computer Architectures*, San Francisco, United States, February 2005.

[77] C. Molina, A. González and J. Tubella, “Dynamic Removal of Redundant Computations”, *In Proceedings of the International Conference on Supercomputing*, June 1999.

[78] C. Molina, C. Aliagas, M. García, J. Tubella and A. González, “Non Redundant Data Cache”, *In Proceedings of the International Symposium on Low Power Electronics and Design*, August 2003.

[79] C. Molina, J. Tubella and A. González, “Reducing Misspeculation Penalty in Trace-Level Speculative Multithreaded Architectures”, *In Proceedings of the 6th International Symposium on High Performance Computing*, September 2005.

[80] C. Molina, A. González and J. Tubella, “Reducing Memory Traffic Via Redundant Store Instructions”, *In Proceedings of the International Conference on High Performance Computing and Networking*, April 1999.

[81] C. Molina, A. González and J. Tubella, “Trace-Level Speculative Multithreaded Architecture”, *In Proceedings of the International Conference on Computer Design,* September 2002.

[82] R. Muth, S. Watterson and S. Debray, “Code Specialization Based on Value Profiles”, *In Proceedings of the 7th. International Static Analysis Symposium*, June 2000.

[83] S. F. Oberman and M. J. Flynn, “Reducing Division Latency with Reciprocal Caches”, *In Reliable Computing*, vol. 2, no. 2, pp. 147-153, April 1996.

[84] S. Onder and R. Gupta, “Load and Store Reuse Using Register File Contents”, *In Proceedings of the 15th International Conference on Supercomputing,* June 2001.

[85] J. Oplinger, D. Heine, M. S. Lam, “In Search of Speculative Thread-Level Parallelism”, *In Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques,* October 1999.

[86] V. Petric, A. Bracy and A. Roth, “Three Extensions to Register Integration”*, In Proceedings of the 35th International Symposium on Microarchitecture*, November 2002.

[87] M. L. Pilla, P. O. A. Navaux, F. M. G. Franca, A. T. da Costa, B. R. Childers, M. L. Soffa, “Reuse Through Speculation on Traces in Deeply Pipelined Superscalar Processors”, *In Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing*, October 2004.

[88] M. L. Pilla, P. O. A. Navaux, F. M. G. Franca, A. T. da Costa, B. R. Childers, M. L. Soffa, “The Limits of Speculative Trace Reuse on Deeply Pipelined Processors”, *In Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing,* November 2003.

[89] M. Postiff, D. Greene, G. Tyson, and T. Mudge “The Limits of Instructions Level Parallelism in SPEC95 Applications. *In Proceedings of the 3rd Workshop on Interaction Between Compilers and Computer Architecture*, October 1998.

[90] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. "A Study of Slipstream Processors". *In Proceedings of the 33rd International Symposium on Microarchitecture*, December 2000.

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

[92] S. E. Richardson, “Exploiting Trivial and Redundant Computations”, *In Procedings of International Symposium on Computer Arithmetic*, pp. 220-227, 1993.

[93] E. Rotenberg, “AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors”, *In Proceedings of the 29th Fault-Tolerance Computing Symposium,* June 1999.

[94] E. Rotenberg, “Exploiting Large Ineffectual Instruction Sequences”, *Technical Report, North Carolina State University*, November 1999.

[95] A. Roth, R. Ronen and A. Mendelson, “Dynamic Techniques for Load and Load-Use Scheduling“, *In Proceedings of the IEEE, VOL.89, NO 11,* Novemeber 2001.

[96] A. Roth and G. S. Sohi, “Register Integration: A Simple and Efficient Implementation of Squash Reuse”, *In Proceedings of the 33rd International Symposium on Microarchitecture*, December 2000.

[97] A. Roth and G. S. Sohi, “Speculative Data-Driven Multithreading”, *In Proceedings of the 7th International Symposium on High Performance Computer Architecture*, Jannuary 2001.

[98] S. S. Sastry, R. Bodik, and J. E. Smith, “Characterizing Coarse-Grained Reuse of Computation”, *In Proceedings of the 3rd ACM Workshop on Feedback Directed and Dynamic Optimization*, December 2000.

[99] T. Sato, "Exploiting Instruction Redundancy for Transient Fault Tolerance", *In Proceedings of the 18th International Symposium on Defect and Fault Tolerance in VLSI Systems*, November 2003.

[100] T. Sato and I. Arita, "Comprehensive Evaluation of an Instruction Reissue Mechanism," *In Proceedings of the 5th International Symposium on Parallel Architectures, Algorithms and Networks*, December 2000.

[101] T. Sato and I. Arita, "Execution Latency Reduction via Variable Latency Pipeline and Instruction Reuse," *In Proceedings of the 7th International Conference on Parallel and Distributed Computing* (Euro-Par), August 2001.

[102] T. Sato and I. Arita, "Table Size Reduction for Data Value Predictors by Exploiting Narrow Width Values," *In Proceedings of the 14th International Conference on Supercomputing*, May 2000.

[103] A. Saulsbury, F. Pong and A. Nowatzyk, “Missing the Memory Wall: The Case for Processor/ Memory Integration”, *In Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996.

[104] Y. Sazeides, “Instruction–Isomorphism in Program Execution”, *In Proceedings of the Journal of Instruction-Level Parallelism, Vol.5, 2003.*

[105] Y. Sazeides and J. E. Smith, “The Predictability of Data Values”, *In Proceedings of 30th Annual International Symposium on Microarchitecture*, December 1997.

[106] J. P. Shen and M. H. Lipasti, “Modern Processor Design: Fundamentals of Superscalar Processors”, *McGraw-Hill Higher Education*, 2003.

[107] P. Shivakumar, N. P. Jouppi, “CACTI 3.0: An Integrated Cache Timing, Power and Area Model”. *Western Research Lab (WRL), Research Report 2001/2,* August 2001.

[108] K. Skadron and D. Clark, “Design Issues and Tradeoffs for Write Buffers”, *In Proceedings of the 3rd International Symposium on High Performance Computer Architecture,* February 1997.

[109] A. J. Smith, “Cache Memories,”, *Computing Surveys,* Vol.14, No. 3, September 1982.

[110] A. Sodani, “Dynamic Instruction Reuse”, Ph.D. Dissertation, Department of Computer Science, University of Wisconsin-Madison, , April 2000.

[111] A. Sodani and G. S. Sohi, “An Empirical Analysis of Instruction Repetition”, *In Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[112] A. Sodani and G. S. Sohi, “Dynamic Instruction Reuse”, *In Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.

[113] A. Sodani and G. S. Sohi, “Understanding the Differences Between Value Prediction and Instruction Reuse “, *In Proceedings of the 31st International Symposium on Microarchitecture*, December 1998.

[114] G. S. Sohi, “Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers”, *IEEE Transactions on Computers*, 39(3):349-359, 1990.

[115] G. S. Sohi and M. Franklin, “High-Performance Data Memory Systems for Superscalar Processors”, *In Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[116] S. P. Song, M. Denman and J. Chang, “The PowerPC 604 RISC Microprocessor”, *IEEE Micro*, 14(5):8-17, October 1994.

[117] A. Srivastava and A. Eustace, “ATOM: A System for Building Customized Program Analysis Tools”, *In Proceedings of the International Conference on Programming Languages Design and Implementation,* June 1994.

[118] C. L. Su and A. M. Despain, “Cache Design Tradeoffs for Power and Performance Optimization: A Case Study”, *In Proceedings of the International Symposium on Low Power Electronics and Design*, April 1995.

[119] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. "Slipstream Processors: Improving both Performance and Fault Tolerance". *In Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[120] G. Surendra, S. Banerjee, S. K. Nandy, “Enhancing Speedup in Network Processing Applications by Exploiting Instruction Reuse with Flow Aggregation”, *In Proceedings of the International Conference of Design, Automation and Test in Europe, March 2003.*

[121] G. Surendra, S. Banerjee, and S. K. Nandy, “On the Effectiveness of Prefetching and Reuse in Reducing L1 Data Cache Traffic: A Case Study of Snort”, *In Proceedings of the* 3rd Workshop on Memory Performance Issues, June 2004.

[122] D. M. Tullsen, S. J. Eggers and H. M. Levy, “Simultaneous Multithreading: Maximizing On-Chip Parallelism”. *In Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[123] G. S. Tyson and T. M. Austin, “Improving the Accuracy and Performance of Memory Communication Through Renaming,” *In Proceedings of the 30th Annual Symposium on Microarchitecture,* December 1997.

[124] L. Villa, M. Zhang, and K. Asanovic, "Dynamic Zero Compression for Cache Energy Reduction", *In Proceedings of the* 33rd International Symposium on Microarchitecture, December 2000.

[125] D. W. Wall, “Limits of Instruction-Level Parallelism”, *In Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[126] S. Wallace, B. Calder and D. Tullsen, “Threaded Multiple Path Execution”, *In Proceedings of the 25th Annual International Symposium on Computer Architecture,* June 1998.

[127] N. Weinberg and D. Nagle, “Dynamic Elimination of Pointer-Expressions”, *In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, October 1998.

[128] M. Wilkes, “Slave memories and dynamic storage allocation”, *IEEE Transactions on Electronic Computers*, April 1965.

[129] Y. Wu, D. Chen and J. Fang, “Better Exploration of Region-Level Value Locality with Integrated Computation Reuse and Value Prediction”, *In Proceedings of the 28th International Symposium on Computer Architecture*, July 2004.

[130] J. Yang and R. Gupta, “Energy Efficient Frequent Value Data Cache Design”, *In Proceedings of the 35th International Symposium on Microarchitecture,* November 2002.

[131] J. Yang and R. Gupta, ”Energy Efficient Load and Store Reuse," *In Proceddings of the ACM/IEEE International Symposium on Low Power Electronics and Design,* August 2001.

[132] J. Yang, Y. Zhang, and R. Gupta, “Frequent Value Compression in Data Caches”, *In Proceedings of the* 33rd International Symposium on Microarchitecture, December 2000.

[133] J. Yang and R. Gupta, ”Load Redundancy Removal through Instruction Reuse," *In Proceedings of the International Conference on Parallel Processing,* pages 61-68, August 2000.

[134] T. Yeh and Y. N. Patt, “Alternative Implementations of Two-Level Adaptive Branch Prediction”, *In Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.

[135] J. J. Yi and D. J. Lilja, “An Analysis of the Amount of Global Level Redundant Computation in the SPEC95 and SPEC2000 Benchmarks", *In Proceedings of the Workshop on Workload Characterization*, December 2001.

[136] J. J. Yi and D. J. Lilja, “Improving Processor Performance by Simplifying and Bypassing Trivial Computations”, *In Proceedings of the International Conference on Computer Design*, September, 2002.

[137] J. J. Yi, R. Sendag, and D. J. Lilja, “Increasing Instruction-Level Parallelism with Instruction Precomputation”, *In Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par)*, August 2002.

[138] Y. Zhang and R. Gupta, “Enabling Partial Cache Line Prefetching Through Data Compression”, *In Proceedings of the International Conference on Parallel* , October 2003.

[139] Y. Zhang, J. Yang, and R. Gupta, “Frequent Value Locality and Value-Centric Data Cache Design”, *In Proceedings of the 33rd International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[140] C. Zilles and G. S. Sohi, “Execution-based Prediction Using Speculative Slices”, *In Proceedings 28th International Symposium on Computer Architecture*, July 2001.

[141] C. Zilles and G. S. Sohi, “Master/Slave Speculative Parallelization”, *In Proceedings of the 35th International Symposium on Microarchitecture,* November 2002.

LIST OF FIGURES

CHAPTER 1. INTRODUCTION 15

CHAPTER 2. VALUE REPETITION IN DATA CACHES 31

CHAPTER 3. INSTRUCTION LEVEL REUSE 63

CHAPTER 4. TRACE LEVEL REUSE 91

CHAPTER 5. TRACE-LEVEL SPECULATION 107

CHAPTER 6. CONCLUSIONS AND FUTURE WORK 147

No Figures

LIST OF TABLES