ENERGY EFFICIENT

COMPUTER ARCHITECTURE

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

IN THE FACULTY OF SCIENCE AND ENGINEERING

October 1996

By

Henrik Scheuer

Department of Computer Science

Table of Contents

List of Figures

List of Tables

Abstract

This thesis describes architectural approaches to improve the energy efficiency of RISC-style microprocessors. By breaking the convention that instructions in RISC architectures must be of fixed size, the performance and energy efficiency of a RISC microprocessor can be improved. Special instruction cache architectures are suggested to ensure an issue rate comparable with that of conventional RISC processors whilst reducing the energy consumption in the instruction cache considerably.

A high proportion of the energy consumption of a microprocessor system is consumed within the caches and external RAM. A significant proportion of memory traffic relates to allocating and de-allocating registers. Register file architectures are proposed to reduce this traffic. Of the schemes investigated, memory mapped registers held in a small separate register cache, has proved to perform well and be energy efficient.

A new branch architecture, which has the potential to eliminate or significantly reduce the miss-prediction penalty of branches through prefetching, will be examined. This scheme, which also improves the hit-rate, employs a pair of instructions. It allows the potential branch target to be prefetched into the cache and into the first stages of a shadow pipeline, before the outcome of the condition evaluation is known and thus reduce or eliminate branch penalties. The overall effect is improved performance. However due to increased cache traffic, the scheme is not energy efficient.

In conclusion, the energy efficiency of a RISC microprocessor can be improved by reducing the average instruction size. The memory traffic can be reduced and the energy efficiency consequently improved, if the allocation/de-allocation of registers can be organised such that interaction with the data cache is minimised. The examined branch architecture may improve performance but is not energy efficient. However, it shows that the Achilles' heel for performance is also the Achilles' heel for energy efficiency.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or institution of learning.

Copyright Notice

The Author

The author graduated from the Technical University of Denmark in January 1990 with a M.SC.EE degree. He joined Advance Computer Research Institute, ACRI, Lyon, France in February 1990 where he, as part of a small team, carried out a feasibility study for a supercomputer project. Later, as a part of the architecture group in ACRI, he wrote the reference model for the ACRI supercomputer architecture. Experience from this work was fed back as contributions to the architecture. The work involved working closely with both hardware and software groups. In February 1994 he joined the AMULET group at Manchester University Department of Computer Science as a research associate. He was working in the ESPRIT-funded HORN sub-group, which investigates ways to improve the energy efficiency of microprocessor architectures.

Acknowledgments

This work has been supported as part of the ESPRIT project 7249, OMI/HORN. I am grateful for this support from the CEC and from SGS-Thomson Microelectronics, the project's prime contractor. I will especially like to thank Mark B. Hill, Andy Sturges and Mark Debbage of PACT, Bristol, for their feed-back in response to presentations, support of software tools and for their openness to queries about the HORN architecture. Furthermore I would like to thank Henk Muller, PACT for his suggestions on the graphical presentation of the simulation results in Chapter 9.

Dr. J.V. Woods, my supervisor, has been a constant source of encouragement. Especially, I would like to thank him for sending me to all the HORN status-meetings. The feed-back from these meetings has been very useful to me.

Thanks are due to Dr. Rhodri M. Davies. Rhods experience with software tools and languages has been valuable and has saved me a lot of time throughout the project. Furthermore I would like to thank him for his very complete proof-reading of this thesis.

During my time in ACRI, Dr. Peter L. Bird taught me many things about computer architecture which motivated me to pursue reseach within this field. I would like to thank him for convincing me that it was a feasible career move and for his encouragement to finish the work.

Finally, I will like to thank Mr. Alasdair Rawsthorne and Dr. Alan E. Knowles for having highlighted the opportunity here at Manchester University and for their support, especially during the difficult phase of settling in a new town and country.

Chapter 1 Introduction

Microprocessors have conventionally been designed to yield maximum performance. Different design approaches have been taken and implementation technologies have improved significantly over the years. Early microprocessors such as the Z80 [Z80] had operating frequencies of approximately 1MHz. Today, in 1996, microprocessors such as the Pentium from INTEL runs at speeds of 200MHz [Child] and the Alpha [DEC21064] from Digital Equipment runs at more than 300MHz. These increases in processor speed have mainly been made possible through improvements in semiconductor technology and chip fabrication which have also allowed an increasing proportion of a computer system to be integrated onto one chip. As chips became larger it became possible to integrate, for example, larger register files and larger caches. Higher levels of integration also allowed architectural innovations such as pipelining [Patt] which overlaps phases of instruction execution and increases performance. Superscalar architectures such as the PowerPC [Gerosa] also became feasible to implement. These developments have resulted in constantly increasing performance.

In the drive for improved performance through higher integration little attention has been paid to power consumption. This thesis shows how architectural features and performance can be traded against power consumption to improve the performance-energy efficiency. The meaning of this term will be discussed at length in Chapter 4.

Improved performance-energy efficiency can be obtained by optimizing the architecture and the chip implementation. This work has examined architectural-level optimizations only, and the results presented are from a number of architectural models. Some implementation-related assumptions have been made; the sensitivity of the architectural results to these assumptions has been reduced as far as possible. The goal is to examine whether architectural features which improve the performance of an architecture also improve the performance-energy efficiency.

Based upon the observation that caches in a typical Reduced Instruction Set Computer (RISC) processor affect both performance and energy consumption, a significant proportion of this thesis is devoted to an understanding of how cache parameters affect the performance-energy efficiency of a microprocessor system with the objective of specifying a performance-energy efficient cache architecture. Register-file and branch architectures are other key-components in a RISC architecture. Different types of register file and branch architectures are examined to gain an understanding of how they affect performance-energy efficiency. Finally the effect of ignoring the dogma, that RISC architectures must have fixed-size instructions is examined. During this examination a number of instruction-fetch mechanisms are developed; the ‘eXtra-line architecture’, described in section 8.3.3, is novel and represents a way of eliminating most of the disadvantages of variable size instructions while retaining the improved cache performance and performance-energy efficiency of this instruction format.

1.1 Background

Compared to earlier ‘Complex Instruction Set Computer’, (CISC) processors such as the 8080 [Spack] and 68000 [Robin], the first RISC [Patt] architectures reduced the semantic content of instructions. Consequently, the instruction count increased. However, due to the simplification of the hardware, higher clock frequencies could be obtained and he overall effect was a *decrease* in execution time.

The quality of the compiler is an important factor in designing a efficient computer *system*. If the compiler takes account of pipeline length and register structure, the code can be scheduled to improve performance through optimized register allocation.

Computer system design is thus a two-branch discipline of providing both fast computer hardware and software tools which optimize the use of the hardware resources. The trend being that some increase in hardware complexity is accepted, if a subsequent reduction in execution time can justify it.

In order to reach the largest market, a wide software base needs to be available implying that binary compatibility must be preserved across a family of processors. The success of the Personal Computer (originally from IBM) can, in part, be explained in terms of the binary compatibility which has been retained through generations of PCs employing the Intel x86-processor family. Binary compatibility is less of an issue in the high-end workstation market, which is dominated by RISC architectures. However, some of the success of Sun's SPARC workstations is explained by the binary compatibility which exists between the different models.

As outlined above, microprocessor development has been driven almost solely by the wish to increase performance; power consumption has rarely been an issue. Improved packaging and cooling technologies, as well as improvements in the semiconductor technologies, have been sufficient to allow processor designers to ignore power consumption when specifying a microprocessor architecture.

It is only recently that computer architects have been forced to pay attention to the power consumption of microprocessors. At a chip level, the increased power consumption has implied that an increasing proportion of the chip area is used for power distribution; it has also implied an increasing number of bonding wires between the chip and the packaging. At a system level, larger and more expensive power supplies and cooling systems are required. To minimise these costs a processor architecture must now be optimized to yield a high performance within the constraints of a limited power consumption.

For example, in order to limit power consumption and thus use a relatively 'ordinary' packaging technology, Digital Equipment Corperation accesses the second-level cache in the Alpha processor in a sequential way: Two cycles to lookup and perform the tag comparison and, assuming a hit, a third cycle to read from the data storage thus reducing the power consumption of the processor by 10W (16% of the total power consumption) [Bensch]. It had become necessary to trade performance against power consumption. A performance-energy efficiency measure is useful when making such optimizations; i.e how can the performance remain high while the energy/power consumption is reduced. Chapter 4 shows that the Alpha processor is among the most performance-energy efficient processors currently available on the market (spring 1996).

There is currently a trend towards portable electronic equipment. Early portable computers, such as the first portable PC's, were portable only in that they had handles and that screen, keyboard etc. could be packed in a convenient way; they were still powered from the mains. Battery-driven lap-top computers were made possible by significant improvements in screen and battery technologies. However, battery life-time - or time between recharges - still leaves much to be desired. In recent years other portable products such as electronic personal organizers and mobile phones have also been introduced to the market.

Improvements in the performance-energy efficiency of microprocessors for the portable battery-driven market is not being driven by the high-end, highest-performance processor-market. ARM Ltd. has had considerable success with their microprocessors which have gained a reputation for delivering a 'reasonably' high performance for a relatively low power consumption. Targeting equipment such as portable telephones has brought considerable commercial success. Other markets include portable computers, electronic personal organizers and portable digital assistants (PDAs) such as the Apple

Newton [Culbert]. These products are becoming increasingly compact and do not contain devices such as cooling fans. A PDA will typically contain no mechanical device such as a hard disk, but will require significant computing power for complex tasks such as handwriting recognition. It should be able to perform tasks such as text formatting or spreadsheet calculations in parallel with the handwriting recognition task. This has to be done without increasing the power consumption significantly as such an increase implies a reduction in battery life-time and/or an increase in the weight due to the number, or size, of batteries. Despite the high performance requirement it is unlikely that the Alpha-processor will be used in portable equipment where battery life time is a very important factor, because, although performance-energy efficient, it has a high power consumption.

To improve the performance of systems such as portable computers and PDAs *and* allow normal usage of these products for at least a working day (10 hours) the *performance-energy efficiency* of the microprocessor is an important measure. The power consumption and performance-energy efficiency of the microprocessor are thus key design-parameters in the product specification along with processor performance and memory size etc.

This performance-energy efficiency measure has been developed only recently. Consequently there is little literature available on performance-energy efficiency of microprocessors. Several conferences have 'low power' sessions, but papers presented tend to examine performance-energy efficiency/power efficiency of sub-systems, especially caches rather than considering the performance and energy consumption of the entire system. This thesis examines how the performance-energy efficiency of a complete microprocessor architecture can be optimized by trading architectural features and their performance against energy consumption.

The work reported in this thesis was carried out within the OMI-HORN project (ESPRIT project 7249). The goal for the group at Manchester University was to specify ways in which the performance-energy efficiency of the HORN processor architecture [HORNV3] [HORNV5] can be improved. As a result much of the work reported centres around an already defined instruction set architecture. This has had the benefit that tools such as compilers, assemblers, functional simulators and some relatively complete libraries were available early in the project.

During the period of the project, some fundamental changes were made in the HORN architecture. The subsequent changes in the tool chain have allowed detailed comparisons between the different architectures. In addition extrapolations to other architectures have been made.

Early in the study it became clear that the power consumption in caches and I/O drivers are major factors in the total power consumption for a microprocessor chip. A detailed study of the ARM3-processor indicated that the 4Kbyte, unified, cache in an ARM3 [OMIMAP] processor consumes 46% of the total power consumption of the chip. A significant proportion of this thesis is therefore devoted to describing how cache parameters such as size, line size and associativity affect energy consumption, not only in the cache, but in the entire system.

Another important issue in performance-energy efficiency is execution time. Given that the project has centred around the HORN-processor, which is a RISC style processor, the instruction count and instruction format are important factors in the expression for execution time. Although the HORN-architecture is RISC-style, the instruction format is unusual in that instructions do not have a fixed size. The implications of this for instruction issuing and cache performance is analyzed in Chapter 10.

Furthermore, a number of register-file architectures have been analyzed. Register allocation handling has a significant influence on both performance and cache access pattern and hence the performance-energy efficiency of the entire processor system.

1.2 Overview of thesis

Chapter 2 describes the power consumption pattern in the ARM3 processor. It reports the results presented in a deliverable to the OMI-MAP project [OMIMAP] which show that the cache consumes a significant proportion of the power in a standard microprocessor. These results are used as a basis for the rest of the work reported in this thesis. In addition, section 2.3 summarizes the power consumption in commercially available RAM.

Chapter 3 describes the HORN-processor architecture, which forms the basis of this work. The instruction format and various register file architectures which have been proposed throughout the specification phase of the project are presented. The special branch structure that the HORN architecture employs is also described. Section 3.5 describes the processor *system*, which will be considered the baseline system for the experiments described in the following chapters.

Chapter 4 discusses how performance-energy efficiency should be measured. The section divides the 'architecture space' into three classes and suggests metrics for each. For microprocessor architectures such as the HORN-architecture, the metric $MIPS^2/W$ was suggested. However T. Burd, University of California, Berkeley [Burd] has suggested an even more general measure based upon the energy-delay product which consequently was adopted. This metric is termed 'energy-efficiency' to comply with terms established in the literature. To establish a basis for comparison, performance and power consumption measures have been collected for a number of processors. The results are presented in section 4.2.

Not all applications can be evaluated using this metric. Many digital signal processing (DSP) applications have a throughput requirement which cannot be traded against lower energy consumption. The decision to use a metric based on the energy-delay product throughout the thesis implies that there should be no DSP-applications in the benchmark suite. Consequently a number of suitable benchmarks have been ported to the HORN architecture. This work has partly been done by Dr. R.M. Davies of the HORN-group, Department of Computer Science, Manchester University and partly by the author.

Chapter 5 establishes how energy consumption of a cache scales with the cache parameters and derives expressions for cache energy consumption. Section 5.1.1 summarizes results from a commercially available RAM-compiler. Based on circuit capacitances extracted from the Amulet2e [Garside]; expressions for energy consumption in RAM are developed. Sections 5.2 and 5.3 derive expressions for energy consumption in direct mapped and n-way set-associative caches.

Based upon observations on redundancy in the tag storage of caches, [Seznec2][Wang] and [Burd] describe a number of cache architectures which can reduce/eliminate this redundancy. Section 5.4 quantifies the degree of redundancy and derives expressions for two of the organizations, sectored caching [Seznec2] and CAT-caching [Wang]. The results presented in these sections have been collected as a part of the author's work.

Section 5.5 describes skewed-associativity [Seznec] as a way of improving cache performance. Within a class of skewing function, it has been investigated whether an optimal set of functions exist, section 5.5.1 concludes that it is not the case. Section 5.6 quantifies the effect different replacement algorithms have on the hit rate in the cache. Section 5.7 describes how the cache parameters affect the timing of the cache. The relationships have been established using the Cacti-tool from Digital Equipment [Wilton].

Sections 5.8 - 5.11 describe techniques to reduce the activity and thereby the energy consumption within a cache.

Chapter 6 describes the effect of branch architectures on the performance and energy efficiency of a microprocessor system. Since the introduction of pipelines in processors, branch instructions have attracted much attention as they disrupt the flow of the pipeline and therefore affect the performance of the system. Many branch prediction schemes have been proposed [Patt] to limit this disruption.

Branch instructions may transfer control to locations which are not in the instruction cache hence affecting the performance negatively. An ability to prefetch the potential branch targets into the instruction cache is therefore beneficial. The HORN architecture specifies a branch structure which allows such prefetching. Chapter 6 assesses the value of the proposed branch instruction architecture both in terms of performance and in terms of energy efficiency. The section concludes that the dual-instruction branch architecture suggested in the HORN architecture is not energy efficient. All the work described and the results reported in Chapter 6 has been undertaken by the author as a part of the HORN project.

The register file architecture affects the instruction count (and thereby the performance). An 'insufficient' number of registers implies a high number of save-restore instructions, as 'old' register-variables are saved to memory to give room for new variables. The saved value may later need to be restored. The ability to allocate[1] new registers when required is therefore important. Increasing the number of registers is not always a suitable solution as more registers require more specification bits in the instructions, resulting in wider buses and lower instruction cache performance. Performance and energy efficiency must

1. 'allocate' in this context means providing a new register. De-allocating is the opposite process, freeing a register.

thus be traded against the number of registers and the mechanisms to allocate new registers. Chapter 7 analyzes the different register-file architectures which have been suggested throughout for the HORN architecture. All the work reported in Chapter 7 has been carried out by the author.

The HORN architecture specifies variable-size instructions but now the instruction format requires more than four bytes. Compared to a conventional RISC architecture with fixed-size instructions such as MIPS R2000 [Farquhar], this ensures that a higher fraction of a program can reside in a cache of a given size; this implies a higher hit rate in the instruction cache. However, variable-size instructions introduce the problem that instructions may straddle cache lines and hence require two cache accesses to be fetched and issued. Chapter 8 proposes three cache architectures aimed at reducing these instances and evaluates them for performance and performance-energy efficiency. The architecture described in section 8.3.3, a novel extension to a block buffering scheme proposed by [Su], almost eliminates the performance penalty associated with variable size instructions. Finally, Chapter 9 evaluates the optimal cache configuration for both performance and energy efficiency. All the results presented in Chapter 9 have been collected by the author.

Chapter 10 draws together the conclusions resulting from this work, assesses the results and presents suggestions for future investigations.

Chapter 2 Power consumption in an ARM3-system

To put the simulation results obtained with the HORN architecture into perspective a 'low power' and popular microprocessor family, the ARM processors are studied in this chapter.

The reasons for this choice were two-fold. Firstly, the ARM architecture has some features which resemble features of the HORN-architecture. It was therefore decided to extrapolate some of the results from this processor family onto the HORN-architecture. Secondly, the AMULET group at Manchester University has strong links with ARM Ltd. This has led to several projects amongst which the development of the asynchronous implementation of the ARM-architecture, in the AMULET-1 chip [Furber2], has attracted much attention. Furthermore, a number of ESPRIT projects have seen collaboration between the AMULET group and ARM Ltd. A deliverable to the OMI-MAP P5386 project [OMIMAP] has been particularly useful to this project and some of its main results will therefore be described here.

2.1 Results from OMI-MAP

The processor analyzed in the OMI MAP project was an ARM3 processor which is an ARM2 processor core with a 4K-byte, fully-associative, unified on-chip cache with 256 lines of 16 bytes.

Instead of building a 256-entry CAM to form the tag-store which would have resulted in very high power consumption, ARM split the CAM into four blocks making it 64-way 4-set associative. This reduces the energy consumption of the overall cache by 21%, as only one quarter of the total CAM need be activated during cache lookups.

The power estimates were based upon two types of ‘measurements’: Spice simulations and estimates based upon the total switching capacity within the design.

The system was broken up into 7 major blocks:

1. A3RAM 1K x 32 bits SRAM
2. A3CAM 4 x 64 22 bit contents addressable memories (CAMs)
3. A3PROC ARM2 CPU macrocell
4. A3CTL Main Cache Control Logic
5. A3COP Co-processor interface
6. Cdata Internal databus (32 bits) RAM/PROC/Databus pads
7. PADS Input/output pads

The power consumption of each block is shown in Table 2.1 It is clear from the table that

Table 2.1 Estimated internal ARM3 power consumption

Block	Average Power Consumption[a] [mW]	Percentage of Total Power
A3RAM	332	30.0
A3CAM	100	9.0
A3PROC	330 (240)[b]	29.8
A3CTL	91	8.2
A3COP	112 (64)	10.1
Cdata	50	4.5
PADS	91	8.2
Total	**1106**	**100**

a. (based on 1.5 µm SPICE data)

b. The numbers in parentheses indicate the power which is consumed in PLAs within the block.

the power consumption in the cache and in particular the RAM is a major factor in the total power consumption figure. The cache accounts for almost 40% of the total power consumption. Reducing the power consumption in the cache will therefore yield a significant reduction in the total power consumption of the processor.

The OMI-MAP report also comments on the use of PLA[1]s in the processor implementation. PLAs are simple to implement, and simple to correct in case of mistakes; without disturbing the chip layout, something which might be required if the combinatorial logic was implemented using ‘discrete’ gates.

The report points out that the PLAs have a static power consumption component, accounting for 70% of the power consumed, implying that the static PLA technique is not appropriate for low power designs!

Table 2.2 shows how the total power consumption drops by 19% if static PLAs are avoided in the design. It also emphasizes the importance of the power consumption of the cache blocks since the percentage for A3RAM and A3CAM has increased to 48.4% or almost half of the total power budget.

Table 2.2 Internal ARM3 power consumption (PLA structures are omitted)

Block	**Average Power Consumption[a] [mW]**	**Percentage of Total Power**
A3RAM	332	37.2
A3CAM	100	11.2
A3PROC	162	18.1
A3CTL	91	10.2
A3COP	67	7.5
Cdata	50	5.6
PADS	91	10.2
Total	**893**	**100**

a. based on 1.5 µm SPICE data

1. Programmable Logic Array

It is suggested that the static PLAs are replaced by dynamic PLAs; these use dynamic AND and OR planes to implement the PLA, with dummy terms to generate self-timing signals to indicate when a result is valid at which point the PLA is put into its pre-charge state and the output is latched [OMIMAP].

[OMIMAP] also gives an example of the area- and timing implications of using a dynamic PLA in the case of the ALU control circuit, A3CTL:

Area:

Dynamic:	452λ x 684λ
Static:	410λ x 579λ

Delay:

Dynamic:	22.2ns
Static:	22.0ns

i.e. the area taken by the A3CTL-block increases by 30%; while the delay through the block increases by less than one percent when changing from the conventional static PLA design to a dynamic design.

2.2 Evaluation of results

The ARM3 processor described in the previous section might be considered obsolete and the value of the power measurements therefore questionable. However, a study of the R3000 architecture from MIPS [BurdPeters], shows that the power consumption in that processor is also dominated by the cache. The report describes power consumption estimates by measuring the amount of switching capacity. It also reports that almost 10% of the power consumed in the MIPS R3000 is consumed in the register file and that the power consumption in the *tag*-storage of the 2-Kbyte instruction and data caches each consume another 10% of the total power consumption. Comparing this value with that for the A3CAM in Table 2.2 shows that the proportion of energy/power consumed in the different blocks is similar in the R3000 and in the ARM3.

The development of the PA-RISC Microprocessor PA/50L [Okada] came to the same conclusion; that power is mainly consumed in internal memories and external signal drivers.

[Sato] reports on a tool, ESP, which is used to assess the implications of architectural changes on the power consumption. Results [Sato] confirm that caches are the dominating components, see Table 2.3.

Table 2.3 Current drawn of blocks in RISC processor [Sato]

Block	Current [mA]	Activity Rate [%]	Average current [mA]	% of total current
Instruction cache	30.0	99.6	29.8	38.9
Branch unit	9.1	99.6	9.06	11.8
Increment addr.	0.1	99.6	0.10	0.1
Register file	13.0	97.4	13.64	17.8
ALU	9.1	59.0	5.37	7.0
Data cache	32.5	47.8	15.54	20.2
Address calculator	9.1	30.1	2.74	3.6
Shifter	5.6	6.4	0.36	0.5
Multiply-Add unit	40	0.2	0.08	0.1
		Total:	**76.8**	**100**

The power consumption of the main RAM-block in the cache is said to be independent, to a first degree, of the dimensions of the cache, due to other overheads, such as I/O buffers and sense amplifiers [BurdPeters]. However the detailed study of the ARM3, described above, shows that is not necessarily the case.

2.3 Power consumption in RAM

As indicated in Tables 2.1 and 2.2, the power consumption of the cache RAM represents a significant proportion of the total power budget in the ARM3. [OMIMAP] also investigated where power is consumed within the RAM. The analysis was divided into two: a 'pre-charge read cycle' and a 'pre-charge write cycle'. This section gives a summary of the results and their implications for the HORN architecture.

Table 2.4 ARM3 RAM dissipation - Pre-charge/*Read* Cycle

Block	Average power consumption [mW]	Comments	% of total power in RAM
ARM3 storage	162.3	The main RAM array	40.6
I/O buffers	68.3	CDATA bus	17.1
A3RAMrd8	81.0	32-Sense Amps	20.3
ARM3row128	41.0	Decode+precharge	10.3
Other blocks	47.4	-	11.9
Total	**400.0**		**100**

Table 2.4 shows the power consumption in the cache RAM during read cycles. It is clear that the storage itself is the major consumer of power, but I/O buffers and sense amplifiers represent a significant 37% of the RAM power budget during read cycles. Other blocks represent only a small percentage of the on-chip RAM power budget.

Table 2.5 ARM3 RAM dissipation - Pre-charge/*Write* Cycle

Block	Average power consumption [mW]	Comments	% of total power in RAM
ARM3 storage	162.3	The main RAM array	64.9
I/O buffers	0.0	CDATA bus	0.0
A3RAMrd8	0.0	32-Sense Amps	0.0
ARM3row128	41.0	Decode+precharge	16.4
Other blocks	46.7	-	18.7
Total	**250.0**		**100**

Table 2.5 shows the equivalent results for a write cycle. The power consumed in the RAM block and in the decode/pre-charge is the same as during the read cycle, see Table 2.4. The

significant difference is that the sense amplifiers and the I/O buffers do *not* consume any power during a write cycle, resulting in a much lower total power consumption. A write cycle consumes only 63% of the energy of a read cycle. Note that the average power dissipation of 332mW quoted in Table 2.2 approximates to the average of the dissipation during read and write cycles. From [OMIMAP] it is not clear why the I/O drivers do not consume any power during write cycles. It will be assumed that the cost of driving the bit-lines is 'hidden' in 'Other blocks'.

The results in the two tables indicate that the sense amplifiers are important components with considerable impact on the power consumption of the cache and, thereby, of the entire processor.

Note that although the ARM3 cache is organised with 16-byte cache lines, the RAM block used for the storage of the cache contents is organised with only one (32-bit!) word per line within the RAM. Short lines/words in the RAM yield a lower energy consumption per request than longer lines, see Chapter 5.

2.4 Summary

This chapter has reported where power is consumed within the ARM3. These results are from the OMI-MAP-project and show that a significant proportion of the power is consumed within the on-chip cache.

The power consumption of the cache has been split into components for the tag-store which, in the case of the ARM3, is composed of four CAM-cells and for the RAM-block which stores data. Tables 2.4 and 2.5 have shown that it is the RAM block together with the sense amplifiers consume the majority of the power. This leads to the conclusion that reducing the power consumption of the cache will reduce the power consumption of the entire processor considerably. The results from this chapter are used to extrapolate the power consumption of the cache architectures proposed in later chapters.

Table 2.2 shows that there are further sub-designs which, if optimised for low power, could improve the performance-energy efficiency of a microprocessor system. This is particularly the case for the pad-drivers. Later chapters will show how Gray coding [Kohavi] can reduce the power consumption of this sub-design and thereby improve the energy-efficiency of the entire processor further.

Chapter 3 Baseline HORN architecture

The baseline architecture used throughout this dissertation is the HORN architecture developed during OMI-HORN, Esprit project 7249.

Key attributes of the HORN architecture are:

1. Modularity
2. Compatibility over a range of products covering a wide range of processing and communications performance.
3. Support for multiprocessing.
4. Provision for a standard programming model which eases the porting of operating systems, compilers and programming languages.
5. 64-bit processor supporting 32-bit operations.

Early releases of the architecture were targeted at the server market; systems with a high number of CPUs, capable of running multiple processes ‘simultaneously’. This required optimization of the thread-change-overhead; i.e. minimising the ‘state’ which needed to be saved and providing efficient ways of saving that state. Efficient inter-processor communications protocols were also required.

During the development process the target markets for the processor changed. The new target markets are *multimedia* systems such as games, video-decoders and Set-Top Boxes (STB), but the processor also targets video-servers providing ‘video on demand’. For these applications the thread changes are expected to be less frequent and the importance of fast thread changes is reduced.

This chapter describes the basic HORN architecture and summarizes the changes it has undergone. It does not describe any original work undertaken by the author but has been included to explain and justify the direction of the investigations described in later chapters.

Section 3.1 gives a general introduction to the HORN-architecture. Sections 3.2-3.4 describe the major features which differentiate the HORN architecture from a conventional RISC, while section 3.5 describes the system considered for this thesis.

3.1 Basic architecture

The HORN architecture is fundamentally a RISC [Patt] in that arithmetic and logical instructions operate on register storage only. Only load and store instructions can access data in memory.

The processor is a byte-addressed, 64-bit processor, i.e. the internal and external data buses are 64 bits wide. Few existing programs require 64-bit variables and the HORN processor's instruction set offers a wide range of operations on sub-ranges: integers of 1,2,4 or 8 bytes and 32- and 64-bit floating point values. Furthermore, to make better use of the wide data bus, the processor includes a new class of instructions: packed arithmetic. These 'packed-arithmetic' instructions operate on a 64-bit quantity as a collection of smaller data quantities. Figure 3.1 shows an example of packed arithmetic, where eight bytes are packed into a 64-bit word and added to a similarly 'composed' word. The result is eight one-byte sums packed into a 64-bit word. Note that any overflow or carry from the eight individual operations is lost.

This technique is potentially very powerful and can be used by the compiler to unroll loops and hence increase the performance and energy efficiency of the processor. It is

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7

+

Byte 8	Byte 9	Byte10	Byte 11	Byte 12	Byte 13	Byte 14	Byte 15

Byte 0 + Byte 8	Byte 1 + Byte 9	Byte 2 + Byte10	Byte 3 + Byte 11	Byte 4 + Byte 12	Byte 5 + Byte 13	Byte 6 + Byte 14	Byte 7 + Byte 15

Figure 3.1 Example of packed arithmetic

especially useful for graphics applications where the representation of pixel colours requires only a limited number of bits.

Memory referencing (load and store) instructions can access 1,2,4 and 8-byte quantities. References to these quantities need not be aligned, e.g. a reference to a 32-bit datum need not be aligned on a 4-byte boundary.

Furthermore the HORN architecture has broken away from the convention associated with the RISC concept, such as fixed-size instructions and has introduced a new type of control transfer instruction.

3.2 Local storage

During the project the form of local storage (register-file) structure has changed several times. This section briefly describes the different types of storage and their main advantages. Later chapters will describe the features of each architecture and evaluate their potential in a energy efficient implementation.

The register, or local-storage of the architecture was originally divided into three classes: Global registers, Local registers and a 4-word operand queue.

3.2.1 Global registers

Global registers are intended to contain stack pointer(s), global variables and constants. Early versions of the HORN architecture mapped a block of 16 global registers to memory through a pointer, GPTR, which could be altered during program execution. This implemented register renaming; the contents of ‘global-register N’ before an adjustment of GPTR by ‘M’ could be accessed through ‘global-register M+N’ after the adjustment. Memory coherency was only ensured after the use of a special form of the ‘adjust-global-pointer’, ajgp-instruction. Later versions of the architecture considered the global registers as a conventional register block.

3.2.2 Local registers

The architecture specifies 32 local registers. The local registers were also originally mapped to memory through a pointer, LPTR. The pointer was intended to be manipulated during execution of a program to allocate/deallocate registers at procedure entry/exit. Registers in scope did not need to be coherent with the memory location they mapped. This scheme will later be referred to as the ‘ajlp-scheme’ after the instruction which ‘adjusted’ the value of LPTR. As for the global registers, coherency was only ensured after the use of a special form of the ajlp instruction.

These schemes with memory mapped registers are very powerful in environments where thread-changes are frequent and therefore need to be fast. Saving the state of a register file (or restoring it) only requires the change of the two pointers, LPTR and GPTR, at the minimal cost of two instructions lasting only a few cycles.

As the architecture evolved and the product was targeted at multimedia applications, where there was less need for a fast context switch mechanism, the memory mapped scheme was replaced by register renaming instructions. These, in addition to renaming the registers, also spilled/filled four registers to/from the memory hierarchy thereby effectively implementing register windows [Weaver]. This scheme will later be referred to as the ‘spill/fill’-scheme after the instructions ‘spill’ and ‘fill’, which caused the actions just described. The difference between this and the more familiar SPARC register windows implementation [Weaver] is that once a register is out of scope in the HORN architecture, its contents should be visible to memory accessing instructions; this is not a requirement in the SPARC architecture.

While the spill/fill scheme might be simpler than the pointer schemes described above, it makes thread and context switches slower; there is now only one way of saving the state of the register file by spilling it to memory. The ‘state-content’ of the new thread can be installed using ‘fill’ instructions. This requires many more instructions than were required with the ‘ajlp’-scheme described above: 32 registers / 4 registers-per-spill/fill = 8 instructions; it may also be much slower, dependent on the exact implementation of the two schemes.

3.2.3 Operand queue

The HORN processor has a set of temporary operand locations. These are organized as a four-entry first-in-first-out (FIFO) queue, which is accessed implicitly. The queue can replace any register reference in any instruction. This reduces the need to use registers for temporary variables and hence the need, temporarily, to store and later re-load, variables to/from the rest of the memory hierarchy.

Table 3.1 compares the result of compiling an expression into machine instructions in a conventional RISC and in the HORN processor. The RISC processor accesses six registers - R_A, R_B, R_C, R_D, R_E, R_t - while the HORN processor will require only the five registers - R_A, R_B, R_C, R_D, R_E. The ‘lifetime’ of the variable in the temporary register ‘R_t’

Table 3.1 Comparative register requirements

Expression: $A = B * C + D * E$

RISC-code	HORN-code
$R_A=R_B*R_C$	Queue = R_B*R_C
$R_t = R_D*R_E$	Queue = R_D*R_E
$R_A=R_A+R_t$	R_A=Queue + Queue

in the RISC code is very short and it is unlikely that it will be required in future calculations; it has however increased the use of the register file and the contents of a register might have to be written to memory to release the space for the R_t value. This saved value might later need to be re-loaded from memory if it is required in later computations.

In the HORN architecture this extra register is not required as the temporary results will be stored in the FIFO-queue and as soon as they are consumed in the following instruction (R_A = Queue + Queue) they will release their storage. It will therefore not be necessary to store any ‘old’ values to memory or to re-load them later.

Note, that the operand queue is a part of the state of the processor and the contents of the queue needs to be preserved across interrupts.

3.3 Branch architecture

‘Control Transfer Instruction’ (CTI) is a generic term for any instruction which can alter the execution flow of a program, such as a branch, jump or call. The actions of this class of instructions can be split into:

1. Compute the potential target

2. Evaluate a condition - conditional branches only

3. Continue execution from the computed target

Some of the actions are orthogonal in that the potential target can be computed independently of the evaluation of the condition.

In modern, pipelined implementations of RISC architectures (see [Farquhar], [Weaver] and [DEC21064]) these three actions are usually combined in one ‘branch’, ‘jump’ or ‘call’ instruction. Due to pipelining it is often not possible to compute the succeeding instruction address fast enough to issue it correctly in the following cycle. Branch delay slot(s) were introduced [Patt] as a way of reducing or eliminating this penalty. Statistics showed that between 40 and 60% of delay slots following conditional branches could be filled and 90% following unconditional branches [Katevenis]. Recent architectures such as the PowerPC architecture [IBM] specify two versions of CTI’s: ‘Branch and Execute’ which executed the instruction following the branch and conventional non-delay-slot branches rather than filling delay-slots with NOP[1]-instructions. The HP-Precision Architecture [Mahon] left it as a part of the instruction to specify whether the following instruction was a delay slot instruction.

The HORN architecture takes a different approach to branching in that it separates the actions into two classes of instructions, the ‘go’ and the ‘leap’ class. The computation of the target is split from the evaluation of the condition, thus requiring two instructions per CTI.

The ‘go’ class of instructions sets up the potential target for the branch; there are a variety of formats covering PC relative offsets, absolute addressing and register relative offsets. Note that a ‘go-class’ instruction overrides the effect of a previous ‘go-class’ instruction.

Once the potential target has been set up, the condition is evaluated using the ‘leap’ class of instructions. Leap-instructions evaluate the contents of a register for a number of

1. NOP = No OPeration

```
             go label
label:       start of loop body
             ....
             ... ...
             ... ...
             leapZ reg, N       } N-bytes
             ... ...
             end of loop body
             ... ...
```

Figure 3.2 Example of a go-instruction outside a loop body

conditions ranging from un-conditional to conditions such as ‘zero’, ‘negative’, ‘positive or zero’. The value to be evaluated must be held in the local storage, see section 3.2. Furthermore ‘leap’ instructions specify *when*, relative to its position, the execution route should be altered if the condition evaluation is positive. This is implemented by specifying a *variable leap shadow* - a number of bytes, potentially covering several instructions, between the leap instruction and the branch location.

This scheme allows the set up of a target instruction stream in advance and allows prefetching of instructions into the cache and/or into a shadow pipeline. The value of this technique will be explored in Chapter 6. Furthermore, in the case of a simple loop the compiler can migrate the go-class instruction outside the loop body and hence reduce the number of instructions issued inside a loop, see Figure 3.2.

This scheme is more flexible than that employed in many commercially available RISC machines such as the SPARC [Weaver]. In these architectures, the size of the branch delay slot is fixed at one instruction, and there is often a significant branch penalty associated with misprediction. With the HORN architecture this mispredicted branch penalty might be eliminated if it is possible for the compiler to migrate the leap-instruction far enough back in the instruction stream.

Figure 3.3 illustrates how the leap instruction can migrate to the very top of the loop body. As the figure illustrates there is now plenty of time to evaluate the outcome of the

```
            go label
label:      leapZ,Q,N
            i = i + 1
            Q = a_i * b_i          } N-bytes branch Shadow

            s = s + Q              <--- Branch or Not
            ... ...
            ... ...
```

Figure 3.3 Optimal migration of the ‘go’ and ‘leap’ instructions

branch. In either case (taken or not taken) no instructions need be fetched speculatively and eventually discarded. The example in Figure 3.3 is very optimistic; it will not always be possible to specify a loop-shadow sufficient to avoid disruption in the pipeline flow. However it is believed that the scheme will perform at least as well as the conventional scheme used in SPARC and MIPS.

A very efficient branch prediction scheme has been proposed [Bird] based upon the sign bit of the displacement for the branch instruction. The scheme yielded a hit-rate of more than 80% by predicting all backward going branches ‘taken’ and all forward going ‘non-taken’. A similar scheme would be difficult to implement, given this two instruction control transfer structure.

It is important to remember that the HORN branch-architecture increases the number of instructions to be executed, as each branch requires two instructions. However, as shown above, the ‘go’-instruction might be migrated outside a loop body by compiler optimizations reducing the overhead. Chapter 6 will evaluate the value of this two part CTI-scheme for performance and for energy efficiency.

3.4 Instruction format

Variable-size instruction formats are not commonly used in RISC architectures. The fetching and decoding of variable-size instructions have been considered too complicated and incompatible with the RISC concept. However, while retaining the other characteristics of a RISC approach, the HORN architecture does exploit variable-size instructions. Instructions can be 1, 2, 3 or 4 bytes in length, the shorter instructions implicitly addressing the queue as mentioned in section 3.2.

The instruction formats used in the HORN architecture are shown on Figures 3.4 and 3.5.

Byte 3		Byte 2	Byte 1		Byte 0	
Arg3:6	ExtOPC:2or4	Arg2:6	Arg1:6	Type:2	OPC:6	Length:2

Figure 3.4 Instruction format

An instruction specifying one, or more, operands from the operand queue releases corresponding register reference bits (Arg1 - Arg3 in Figure 3.4) in the instruction format. The usage of the queue is specified in the first bytes, (Length and Type fields, see Figure 3.5). Table 3.2 shows examples of instructions and their corresponding sizes where a '*' denotes that the corresponding operand is to be taken-from/written-to the operand queue.

Table 3.2 Operand Queue usage and the corresponding instruction sizes

Instruction	Size [bytes]
add R1,R2,R3	4
add *,R1,R2	3
add *,R1,*	2
add R1,*,*	2
add *,*,*	1

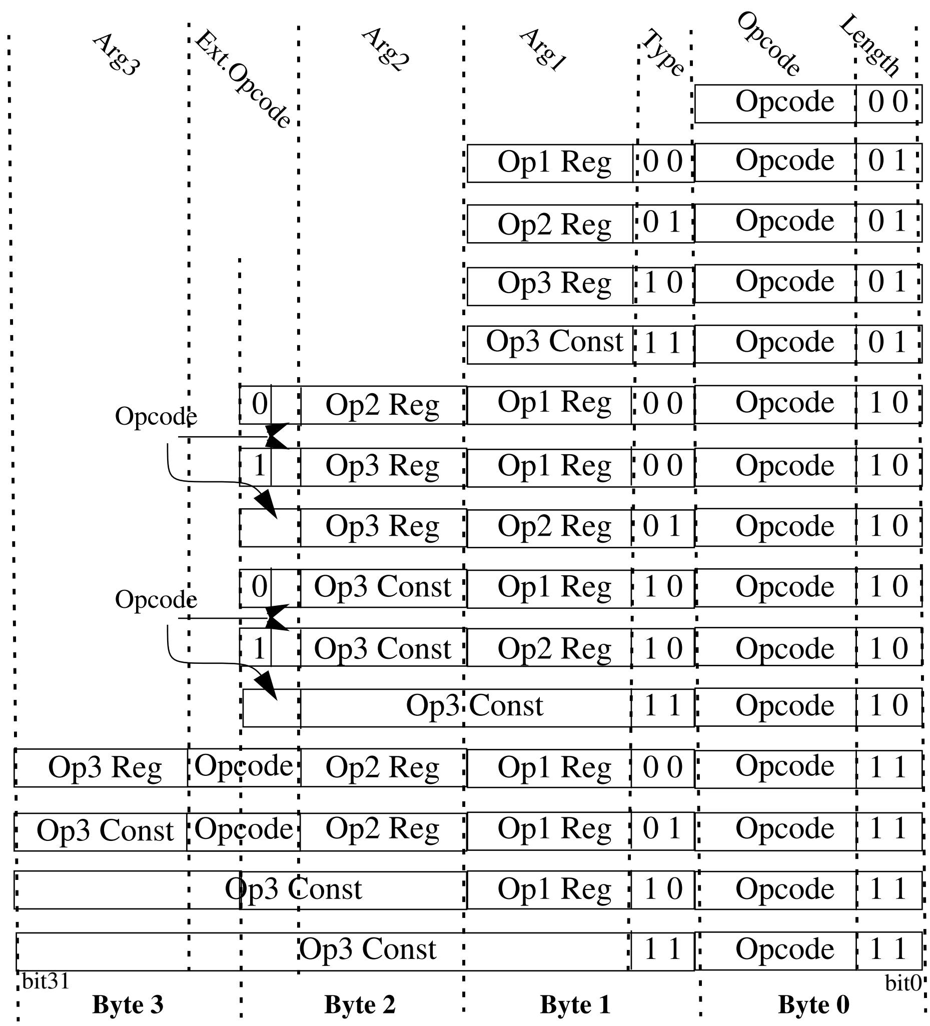


Figure 3.5 Instruction encoding

Using this technique, the average instruction size can, with the current compiler technology, be reduced to 3.12 bytes/instruction, a 23% reduction compared to conventional 4-byte RISC instructions, see Chapter 8. Consequently, a cache line of 32 bytes can, on average, contain more than 10 instructions instead of 8 conventional 4-byte instructions. This reduces the pressure on both the cache and memory system as the cache can contain a larger proportion of the program. Consequently, the instruction cache[1] hit-rate increases on average by 1.1% (see Table 8.3) corresponding to an average reduction

1. Cache parameters: 8Kbyte, 32 bytes per line, 2-way set-associative, Random replacement

in I/O traffic[1] of 12%; this implies a significant increase in performance and decrease in power consumption. The exact effect on performance depends on the cache configuration:

- Does the cache block the processor on misses until the entire line has been fetched?
- Is the requested word forwarded to the execution pipeline straight away?
- Is the requested word fetched first?

Note that this technique of reducing instruction sizes does not change the semantic content of the instructions. The number of instructions required for a given program is the same as for the conventional 4-byte instruction format. In fact, a program can be turned into a program of 4-byte instructions simply by referencing the queue through its register alias, register 63.

Other techniques which reduce the program size are described in [Bunda][Fleet] where all instructions are 16 bits. This gives a smaller code size but increases the number of instructions as the semantic content of each instruction is reduced. The Thumb-format in some ARM processors (see Chapter 2 and [Furber]), allows instructions to be encoded into a 16-bit instruction format which is ‘decompressed’ during execution. A program can be composed of a number of code fragments, some written in normal ARM code, some in Thumb code. Thumb code is entered using a special instruction and there is another to ‘return’ to ARM-code. A Thumb-code fragment is guaranteed to contain an even number of instructions to ensure proper alignment of succeeding conventional instructions. As is the case in [Bunda] and [Fleet] the semantic content of these compressed instructions is *not* the same as the conventional 4-byte instructions as only a subset of the registers and

1. Traffic between the instruction cache and the external memory

of the opcodes is available. The binary of a program compiled into Thumb format will therefore typically contain 40% [Furber] more instructions than the corresponding conventional 4-byte instructions format. Consequently the size of a Thumb binary is approximately 30% smaller that the equivalent 4-byte-per-instruction binary. With the instruction formats discussed in [Bunda] and [Furber] individual instructions cannot straddle cache lines.

Variable-size instructions do introduce the problem of instructions, which may straddle cache line boundaries; these require two cache look-ups implying a potential performance degradation. Chapter 8 will propose instruction cache architectures which almost eliminate this problem.

3.5 Summary

This chapter has described the HORN architecture which forms the basis for much of the work reported in this thesis. The chapter has highlighted the areas where the HORN architecture differs from most RISC architectures: The register file, the branch architecture and the instruction format. Furthermore, the concept of packed-arithmetic has been described.

The work described in this thesis has made extensive use of the tools developed for the HORN processor. Many of the techniques described in later chapters are closely linked to this architecture. On two points though, the tools do not explore the options that the architecture gives:

1. The HORN processor is a 64-bit processor. However, the compiler developed for the prototype system does not exploit this feature. No instruction operates on 64-bit quantities nor does any register contain values which cannot be contained in a 32-bit register. Furthermore

the architecture specifies a 64-bit address space, but again, no address is ever accessed which could not be contained in a 32-bit integer. The study carried out in the rest of this thesis will therefore describe the HORN architecture/processor as if it was a 32-bit processor.

2. Section 3.1 stated that data references need not be aligned on corresponding byte boundaries. No instances of this have been encountered. The rest of this thesis will therefore assume that data references are aligned on proper byte boundaries. The same assumption is, of course, not made for the variable length instructions.

3. Although the architecture manuals [HORNV3] and [HORNV5] do not specify it, this work will assume a Harvard architecture [Patt]; i.e. separate non-overlapping instruction and data memory segments.

Chapter 4 Metrics and benchmarks

When evaluating a computer architecture some of the questions that arise are: When is an architecture or a processor implementation performance-energy efficient? What does performance-energy efficiency mean? In order to answer these questions and make trade-offs and comparisons a suitable metric is required. [Burd] states that the metric for performance-energy efficiency differs, dependent on the class of application. There should therefore be different metrics for different classes of products. All the metrics considered are based upon other metrics such as performance, measured in MIPS, and power consumption, measured in Watts. Other measures of performance such as Specmarks [SPEC91] or Dhrystone [Weicker], would be equally suitable as a measure of performance and have the advantage of being instruction set insensitive.

The following sections describes how applications can be classified for ‘low power’. The classification is made by T. Burd from University of California, Berkeley [Burd]. The metrics proposed are based upon the throughput of a processor, which do not allow comparisons to be made across different instruction set architectures.

[Burd] uses the term ‘energy-efficiency’ generically to describe the ratio between performance and energy consumption, i.e. the performance-energy efficiency. Although the ‘energy-efficiency’ term is not as precise as ‘performance-energy efficiency’ it was felt that consensus was needed in this field of research. Consequently, this thesis adopts the term ‘energy-efficiency’ to describe the relation between performance and energy consumption. Energy-efficiency will be abbreviated EE throughout the rest of this thesis.

To illustrate the importance of choosing the right metric, performance and power consumption measures have been collected for a range of processors from a number of sources into a table (see Table 4.1 on page 56).

Finally, seven benchmarks were ported to the HORN architecture, partly by the author, partly by his colleague on the HORN-project Dr. Rhodri M. Davies.

4.1 Background on metrics

Performance is generally measured in Million of Instructions Per Second (MIPS) and energy consumed is measured in Joules. Energy per unit time is power consumption measured in Watts.

[Burd] divides applications into three classes:

1. Digital Signal Processing (DSP) - class applications, which require a fixed level of performance, and do not benefit from any further increase in performance. The challenge in such a system is to deliver the required performance while consuming as little energy as possible. The energy per operation should be minimized. This is equivalent to 'power divided by throughput', which is inversely proportional to 'MIPS/W' or 'SPEC/W', a figure often quoted in the literature [Zivkov].

2. Server applications, which are characterised by the processor constantly being busy. For this class of applications the energy-efficiency metric should be based upon two factors: The execution time and the energy consumption; (an energy-delay product) as performance can be traded against energy consumption and vice versa. As it will be explained below this metric is proportional to Energy/Throughput and Power/Throughput^2, which is inversely proportional to MIPS^2/W or SPEC^2/W. [Burd] denotes this measure Energy Throughput Ratio, ETR.

3. PC/workstation applications, where the processor is typically busy for some time T_1 followed by some idle time T_2. Products such as Personal Digital Assistants (PDAs), Personal Organisers (POs) also come into this category. Dependent on the ratio of these times, it may become beneficial to 'put to sleep' parts of, or even the

entire, processor in order to obtain high overall energy efficiency. This is called 'burst mode' in [Burd] which defines a metric Microprocessor Energy Throughput Ratio (METR). Once again this metric is based upon the energy-delay product and measured in Joules/MIPS inversely proportional to $MIPS^2/W$ or $SPEC^2/W$. It is important to note that the MIPS in this case are the 'useful' MIPS, i.e. also the performance of the operating system when no specific application is running. If $T_1 >> T_2$ this is equivalent to ETR. [Burd] argues that if $T_1 << T_2$, METR turns into a 'Power/Throughput' metric inversely proportional to MIPS/W.

The HORN Architecture Manual - 5th Edition specifies:

"... first phase concentrating on high volume consumer computing products with low system cost. Promising areas for initial HORN products are:

a. Multimedia Systems...... integrating DSP, image processing and communications functions into the general purpose processor.

b. Games....

..... In the second phase, new products will build on this, addressing general purpose computing areas such as:

c. Multimedia servers...."

Comparing the list of applications with the three classes of systems there is significant analogy:

Multimedia Systems are typical DSP applications, Games are a typical PC-style application, which might stay idle for some time, waiting for user input, before becoming busy again. *Multimedia Servers* are specific forms of server applications where there are limits to the trade-offs one can make in the performance. Given that it is a server, however, an assumption can be made that the system can handle more than one stream of data at the requested rate, the trade-off that can be made is therefore trading the number of streams versus power consumption.

It might not be obvious why energy efficiency is a relevant issue for servers. Servers will typically not be battery driven but cooling and noise might be important issues. If the processor in the server is energy efficient, cheaper and less noisy cooling technology may be used resulting in a cheaper and more environmentally friendly product.

Note that there is, in general, a performance requirement associated with applications of type 1, while this is not necessarily the case for classes 2 and 3.

Let β denote the ratio P_{IDLE}/P_{BUSY} i.e. the power consumption while idle divided by the power consumption while busy. By having T denote the throughput for a given application and T_{AVE} denote the average throughput over time [Burd] established the following relation between ETR and METR:

$$METR = ETR\left[1 + \beta\left(\frac{T}{T_{AVE}} - 1\right)\right] \qquad T \geq T_{AVE} \tag{EQ 4.1}$$

which is applicable for classes 2 and 3.

Early microprocessors typically had β-values of 1.0, while an asynchronous implementation will yield a much smaller β-value approaching 0. Modern processors such as the INTEL Pentium [Child] and the PowerPC 603 from Motorola [Suessmith] have several power down modes, yielding a range of β-values. Note that if $T=T_{AVE}$, i.e the processor is busy all the time, METR = ETR.

β is a constant given by the implementation while the architecture 'determines' ETR. T and T_{AVE} are given by the user/application. Thus optimising the $MIPS^2/W$ ratio at the architecture level will minimize the ETR and minimizing the power consumption during idle periods through various implementation techniques will decrease β and therefore also be a gain. The two factors can thus be considered independently and can be optimized for separately. Applications of type 2. and 3. can be used to optimize the (micro) architecture, but that applications of type 1. may not lead to the same results.

This analysis has two consequences for the investigations carried out as a part of this work. Firstly, it is very easy to find applications/benchmarks of type 2 and 3, which can help in optimizing the architecture and later the implementation for energy efficiency. Essentially all the benchmarks normally used for processor performance assessments can be used as benchmarks. Secondly, it was decided to consider the HORN processor as a 'normal' microprocessor where performance improvements obtained through optimizing the architecture are 'passed-on' to the application/user.

This could lead to the conclusion that an ETR or a $MIPS^2/W$ metric is the correct metric for optimizing an architecture for energy efficiency. However, the metrics are only suitable if the programs/benchmarks used contains the same number of instructions for all the architectural options explored or, more precisely, the same instruction set architecture (ISA) [Burd]. As shown in Chapter 6 this work has also explored different instruction set architectures where 'MIPS' is not a suitable metric for performance as the number of instructions required for a given program compiled for two different architectures may not be constant. An example of this is the comparison a program compiled for a RISC and a CISC architecture. The CISC binary will typically contain fewer instructions than the RISC binary. To measure the effectiveness of two different architectures it is the execution time of the program which is interesting to the user, not the MIPS-ratio for the two architectures. Other measures of performance which are instruction set insensitive such as the Dhrystone [Weicker] or the SPEC[SPEC91] measure could have been chosen. The SPEC benchmark suite comprises integer and floating point benchmarks denoted

‘SPECInt’ and ‘SPECfp’ and is updated regularly, the year of the release is often specified. SPECInt92, consequently refers to the integer benchmarks in the SPEC benchmarks suite from 1992. However, given the difficulties encountered porting benchmarks (due to incomplete libraries) it was considered infeasible to use the SPEC-measures. Furthermore, due to the danger of focusing too much on one benchmark, the Dhrystone performance measure was discarded as well. Simplicity was needed:

An energy efficiency metric should only consider the time taken for a task or program and the energy consumed during the execution of the task. This leads to a energy-delay or J·sec metric. Note, that this definition *does* take account of architectural parallelism as the delay is the time taken to execute a given task, rather than the cycle time of the processor. Using the critical path delay as a measure for time, fails to include the effects of architectural parallelism [Burd].

To allow positive selection the inverse metric: $\frac{1}{J \cdot \sec}$ will be used in the comparisons. It will be called Energy Efficiency, EE. Such a metric can be used to evaluate different architectures and configurations for energy efficiency but it cannot be used to compare different benchmarks, even in the same architecture. Note that as the definition of EE is based upon the energy-delay product it is not suitable for DSP-class applications where the delay cannot be traded.

It is important to realize that this ‘new’ metric is inversely proportional to ETR and proportional to MIPS^2/W as long as the same ISA is used in the evaluations:

$$ETR = \frac{J}{MIPS} = \frac{\frac{J}{\sec}}{\frac{\mathrm{Inst}/10^6}{\sec \times \sec}} \propto \left.\frac{W}{MIPS^2}\right|_{\text{for constant no. inst}} \qquad \text{(EQ 4.2)}$$

$$EE = \frac{1}{J \cdot \sec} = \frac{\frac{1}{\sec^2}}{\frac{J}{\sec}} \propto \frac{MIPS^2}{W}\bigg|_{\text{for constant no. inst}} \qquad \text{(EQ 4.3)}$$

Furthermore, the metric has the advantage of being independent of the supply voltage: The power consumption scales with the square of the supply voltage, V_{dd} [Weste], while the operating frequency scales linearly with the supply voltage [Weste][RYork2]. Consequently the EE metric is independent of the supply voltage.

4.2 Evaluation of metrics and discussion

It might be considered controversial to choose a metric such as energy efficiency when the science generally [Zivkov], [Williams], [Lev], [Bensch] quotes measures such as MIPS/W or SPECInt92/W. However, as explained above, if the goal is to increase the throughput per unit of energy (the EE), $MIPS^2$/W and $SPECInt92^2$/W are more suitable metrics.

To illustrate the sensitivity of which metric to choose Table 4.1 shows the SPECInt92 measures and power consumption measures from a number of processors [MRP1092], [Zivkov], [Williams], [Lev], [Bensch], [RYork], the SPECInt92/W and $(SPECInt92)^2$/W.

Note that all ARM measures are based upon the Dhrystone benchmarks! The SPECInt benchmarks are designed to assess the *system* performance including I/O operations rather than just the processor performance. The Dhrystone measure is based upon compute power and ARM Ltd. considers it a more appropriate measure for the performance of their processors [RYork][1]. A conversion factor between the Dhrystone and the SPECInt performance measures has been derived for the MIPS R4200 from

1. More information on the performance and power consumption of the ARM610 and ARM710 is available on WWW: http://www.arm.com

Table 4.1Performance and power consumption for existing processors

Processor	Frequency [MHz]	SPECInt92	Power [Watts]	$\frac{SPECInt92}{W}$	$\frac{(SPECInt92)^2}{W}$
Alpha-Quad[a]	300	341	50	6.8	2,326
ARM3	20	6.4[b]	1.1	5.8	38
ARM610[c]	25	12.0	0.53	22.7	272
ARM710a[d]	50	20.1	0.32	63.1	1,266
Hobbit	20	11.0	0.4	27.5	302
i486 DX/2	66(int)/33(ext)[e]	32.2	7	4.6	148
MicroSPARC	50(int)	22.8	4	5.7	130
Pentium	66	64.5	16	4.0	258
PowerPC	66	60.0	9	6.7	402
PowerPC[f]	80	75.0	2.2	34.1	2,557
R4000SC	100(int)/50(ext)	61.7	12	5.1	317
R4200	80(int)/40(ext)	55.0	1.5	36.7	2,018
R4400SC	150(int)/75(ext)	94.0	15	6.3	592
SPARC V9 MCM[g]	143	230	50	4.6	1,058
SPARC V9[h]	167	270	28	9.6	2,592
68040[i]	33	17.7	1	17.7	313

a. Quad-issue full-custom VLSI implementation of the Alpha-architecture

b. SPECInt92 performance for ARM3 estimated by comparing R4200 performance of 137K Dhrystone [Zivkov] with 16K Dhrystone for ARM3[OMIMAP].

c. The power consumption are estimates based upon [MRP1092]. The ARM610 core is expected to consume 525mW to deliver the performance of a Hobbit processor [MRP1092]. The SPECInt92 measure is derived from running the Dhrystone2.1 benchmark on an ARM610 - 25MHz within the department, yielding 31 KDhrystone2.1.

d. Personal mail exchange with Mr. R. York, ARM Ltd., 4th of April 1996. ARM710a has an 8K-byte unified cache. Performance: 52KDhrystone2.1,

e. int: internal clock frequency, ext: externally supplied clock frequency

f. see [Gerosa]

g. A HaL implementation of the SPARC V9 64-bit architecture, integrating a CPU-chip, a memory management unit and four 64K byte cache chips into a ceramic multi-chip module

h. 64-bit superscalar, 4 instructions per cycle

i. see [Biggs]

[Zivkov]: 1 SPECint92 = 2.5 KDhrystone2.1. This conversion factor has been used to assess the SPECInt performance of the ARM processors

The processors in Table 4.1 can be divided into three bands determined by their power consumption. Some processors, the Hobbit, the R4200 and the ARM processors, consume little power, less than 2.5W. A number of processors consume between 4W and 16W, while three processors consume more than 20W.

Using the $\frac{SPECInt92}{W}$ metric (i.e. the power-delay product) the first group of processors (excluding the older ARM3) stand out, yielding measures a factor 3 to 6 better than any of the other processors. However it should be observed that the 'SPARC V9' processor performs very well with this metric despite its 28W power consumption. Using the same measure, it can be seen that the 'Hobbit' obtains its high measure through a very low power consumption despite a low performance (SPECInt92); whereas the 'R4200' architecture gains its position though effective performance, which is comparable to early 486-microprocessors and low power consumption.

Using the energy efficiency metric, $\frac{(SPECInt92)^2}{W}$ (i.e. the energy-delay product), the ranking changes completely. Now, the high-end processors, together with the R4200 and the PowerPC[1] form a class of their own yielding measures a factor 3 to 10 times better than any of the other processors. These measures indicate that an 'Alpha' processor delivers eight times more performance per Joule than a 'Hobbit' processor or nine times more than a 'Pentium' processor. Although the 'Alpha' and other high-end processors are very power consuming, they are very energy efficient in their computation. Thus an 'Alpha' or a 'SPARC V9' processor could be the optimal choice if one was to build a server (defined as above), independently of whether the goal was energy efficiency or throughput. However, the 'R4200' or the 2.2W PowerPC would, despite their lower performance, also be very energy efficient choices.

The results do *not* imply that the optimal processor for *hand-held* equipment is an Alpha or a 'SPARC V9'. The degree of utilization might be very low, as is the case for a PDA.

1. The 2.2W version described in [Gerosa]

It is therefore important, as shown in section 4.1, that a processor has 'power-down' modes entered during idle periods. The power consumption during this time should be as low as possible, see Equation 4.1, to extend battery life time. The 'Alpha' does not have such a feature and will therefore consume 50W independently of its utilization. In contrast the 2.2W PowerPC does [Suessmith] use dynamic power management where different parts of the chip can be shut down. Dependent on how much of the chip has been disabled, stand-by power consumption of a PowerPC-processor is between 2mW and 350mW.

As illustrated in [Culbert], PDA-style products are designed with great attention paid to weight and size. The power consumption and supply voltage of the processor is important. For the 'Apple Newton' PDA [Culbert], the designers chose to use the ARM610 processor due to its low power consumption, which Apple estimated would give one week of 'normal' use with 4 AAA NICAD battery cells. As mentioned above, the ability to 'power-down' the processor is also important. In stand-by mode, the Apple Newton PDA consumes only 50mW compared with 2W when operating [Culbert], i.e a β-value of 0.025 in Equation 4.1. A part of this reduction is due to 'powering-down' the processor.

If the requirement for battery lifetime was reduced to a working day a '486DX2-66' processor might have been the optimal choice. It has approximately the same energy efficiency, but provides a performance which is three times higher than that of the 'ARM610', see Table 4.1.

Choosing the right processor for a portable product involves more than choosing the most energy efficient processor. Factors such as battery lifetime and required performance level are important factors as well.

Figure 4.1 presents the results from Table 4.1 in a graphical form. The graph can be used to aid the selection of the optimal processor for a given product. Given a power-budget

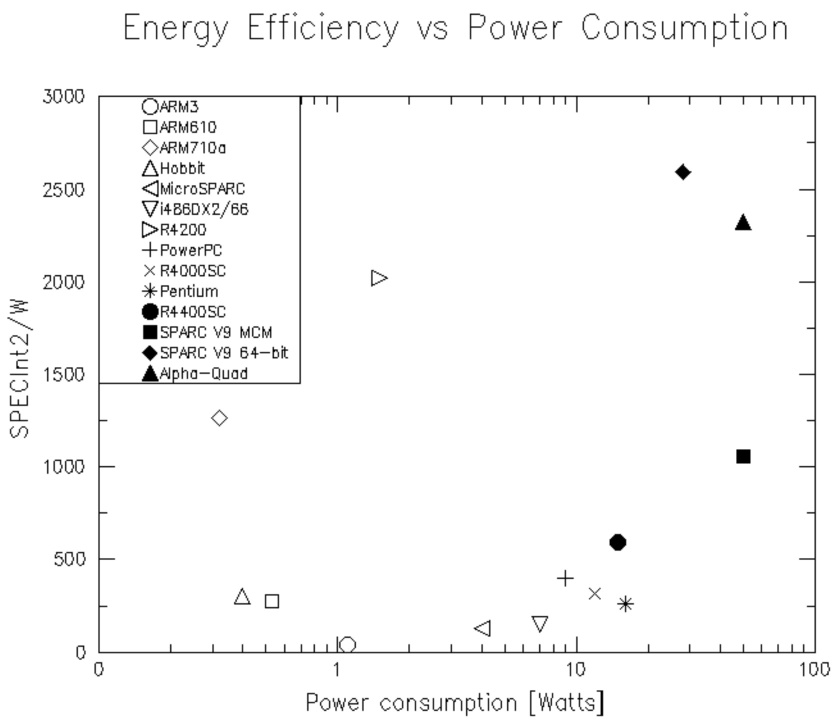


Figure 4.1 Energy Efficiency vs. power consumption for existing processors

the graph presents a simple way of choosing the most energy efficient processor. The scale on the 'Power Consumption' axis is logarithmic. This helps to differentiate the processors in the low end of the range. However it makes it more difficult to separate the 'high-end' processors.

4.3 Selection criteria for benchmarks

The sections above have highlighted that optimizing a computer architecture or an implementation of an architecture for energy efficiency has different meanings dependent on the type of target application.

In section 4.1 it is stated that the goals which are set up for the HORN architecture [HORNV5] are contradictory when optimizing the architecture for 'low-power'; as the architecture aims to address both DSP and microprocessor/server applications. The section concluded that the work in this dissertation should focus on the improvement of server and microprocessor applications and use the related metrics such as Energy Efficiency, EE.

The decision to use this metric excludes DSP-class applications from the benchmark suite, so MPEG and related programs can not be used as benchmarks even though they are listed as one of the prime targets for the HORN architecture. The benchmarks suite has therefore been chosen to contain typical benchmarks for the workstation/PC domain.

The HORN architecture has undergone significant changes during this project. The instruction set architecture and consequently the compiler and linker have been changed several times. Due to the emphasis on optimizing the architecture for performance the development of libraries was deferred. This had consequences for this project in that it was not possible to port a significant number of benchmarks such as the SPEC [SPEC91] and SLASH [Singh] suites to the HORN architecture.

4.4 Benchmark Suite

This section gives a brief description of each benchmark used in this dissertation. There will be a short description of the functionality of each benchmark and the characteristics which justified its inclusion in the benchmark suite for this work.

4.4.1 Hello

Hello is an extended version of the minimal program printing the classical phrase. The program has been extended to contain a loop which iterates 20 times over a print statement printing the value of the loop-variable. The benchmark has been included in the suite to represent the class of small programs or tools, such as the UNIX utility 'grep' which is heavily used to scan text-files.

The executable program is characterized by being built almost entirely from library routines and start-up code. It is therefore useful for illustrating the effect of these code-fragments as they will be found in all the benchmarks mentioned hereafter.

The executable comprises 52.000 instructions and 21.000 data references

The benchmark contains 16 lines of C-code.

4.4.2 Espresso

Espresso [SPEC89] is a program which transforms a boolean truth-table into a form suitable for a given implementation technology with emphasis on criteria such as speed and area. The program takes a textual input file and produces an output file in a variety of formats. The default output format was used.

The executable program comprises a significant number of instructions, 4.6 million, of which 1.4 million are memory referencing instructions. The code contains a number of short loops which is illustrated by the fact that 12% of taken branches branched less than 32 bytes[1] backwards. The code does contain some I/O procedures, but the majority of the run-time is spent in the reduction algorithm.

Espresso is the largest binary in the benchmark suite having the largest memory requirement it therefore has the highest number of compulsory misses [Patt] in the instruction cache.

The benchmark contains 17,000 lines of C-code.

1. Equivalent to approximately 10 instructions

4.4.3 Flex

Flex is a 'fast lexical analyzer generator' which is available as a UNIX tool. The program contains 10.7 million instructions of which 4.3 million are data referencing instructions. Compared with Espresso, see section 4.4.2, a high percentage of taken branches had very small offsets. 95% of all taken branches branch to the same cache line (32 bytes). This, together with the fact that the size of the binary is only 50% of espresso, indicates a high degree of spatial locality and that high hit-rates can be obtained in instruction caches even with small cache sizes.

Flex has been included to represent tools commonly available (and used) on workstations.

The benchmark contains 12,000 lines of C-code.

4.4.4 Cacti

Cacti [Wilton] is a cache timing analyzer program developed by Digital Equipment Coperation; it calculates various timing parameters for a cache specified by the user using information from a technology file. The user can specify total cache size, cache-line size and degree of associativity. The program provides information about cycle and access time to the cache and sub-divides these 'times' up into various components in a cache such as latency through the sense amplifiers in both the data and tag areas in the cache.

The size of the binary is very small. However, due to heavily nested loops, the dynamic instruction count is 18.9 million instructions of which 2.1 million involved memory references. Consequently, it is not as memory-intensive program as is for example espresso, see section 4.4.2.

The spatial locality is not as high as for espresso and flex, only 6% of taken branches are to the same cache line (32 bytes). The hit-rate in the instruction cache is therefore expected to be lower than for espresso and flex. Furthermore the average basic block size in this benchmark is more than twice that of the other benchmarks: 54.3 bytes versus an average of 27.1 bytes for the rest of the benchmarks.

Cacti has been included in the benchmark suite to represent simulators and tools used in R&D environments. As the instruction count is very high, the benchmark can be considered as one used for testing high-end systems.

The benchmark contains 1,500 lines of C-code.

4.4.5 Fft

The Fast-Fourier-Transformation (FFT) benchmark Fourier-transforms 1,024 numbers, generating 1,024 complex numbers; these complex numbers are then fed to an inverse-fft process which regenerates the 1,024 original numbers.

The program prints a number of useful time statistics such as the over-all execution time, the transpose time (i.e the time it takes to perform the Fourier-transformation) and the initialization time.

As a benchmark, the program is characterized by being sensitive to organization of the data cache due to the non-linear access of data. For large caches the cache organization is less important due to the relatively small data-set. It is also characteristic that the array-elements are accessed very few times. The data cache misses are therefore dominated by compulsory misses.

The performance of the data cache is such that the hit-rate is high (>95%) even for small caches as long as the degree of associativity is higher than one.

The binary of this benchmark is very small, less than twice the size of hello and the performance of the instruction cache is therefore expected to be high (>98%) even for small configurations. Furthermore, 17% of all taken branches are to the same, current, instruction cache line. However, the spatial locality is much higher than this number indicates as a significant proportion of the branches are taken during the initialization of the code. Once in the core in of the program the spatial locality is much higher.

‘Fft’ has been included in the benchmark suite as signal processing is one of the targets for the HORN architecture. However, it should be noted that there is no timing requirement associated with this benchmark as is typical for DSP class applications, although fft is often a significant part in DSP applications.

The benchmark contains 1,000 lines of C-code and executes 1,104,931 instructions including 225,796 memory referencing instructions, i.e. a 5:1 ratio.

4.4.6 Dhrystone

Dhrystone2.1 [Weicker] is another small benchmark, performing a precise number of tasks. The program contains a main loop. This loop is executed N times where ‘N’ is a parameter given to the program when launched. For all the runs of this benchmark, N was set arbitrarily to 500. This allows the program to measure the time taken to compute the precise number of arithmetic operations and on this basis various performance characteristics were calculated for the machine used to run the program. Computing these measurements involves timing the calculations which used timing calls not included in the libraries supplied with the tools for the HORN architecture. The calls to these functions have therefore been “commented-out” of the source code and the statistics which involved timing have been omitted.

The binary of the benchmark is small, only 50% larger than hello, and as the data set is very small, a data cache size of 2K bytes is sufficient to ensure a hit rate in the data cache of more than 99%.

Dhrystone is a classical synthetic benchmark quoted in [Weicker] and has been included in the benchmark suite for that reason.

The benchmark contains 1.000 lines of C-code, 688,173 instructions including 221,819 memory referencing instructions; i.e. a ratio of 3:1. The program is characterized by short basic blocks, typically six instructions.

4.4.7 Stcompiler

Stcompiler is a publicly available C-compiler [Ruegg] used to compile the 'hello.c' program for measurement purposes. The program has, together with flex, see section 4.4.3, the largest binary in the benchmark suite and due to relatively low spatial locality, a large cache ($\geq$8K bytes) is required to obtain a high hit rate, >98%, in the instruction cache. Equally, the data cache needs to be large (> 4K bytes) to ensure a hit rate there of >95%.

The reason for the low hit-rates, particularly in the data cache, is a very high number of compulsory misses [Patt] due to a very large binary and data set. The benchmark has been included in the suite to represent the compilers which are common in a development environment. It contains 13,000 lines of C-code, executes 1,865,924 instructions including 720,468 memory referencing instructions, i.e. a ratio of 2.6 : 1; it is a very memory intensive benchmark. The program is characterized by short basic blocks: typically 4.6 instructions

4.5 Summary

This section has shown that, provided DSP-like applications are not targeted, EE is a suitable metric for evaluating energy-efficient architectural trade-offs. Even though the HORN architecture specifies DSP-like applications as one of its targets, this work will *not* optimize for this class of application. As stated above, the metrics, when optimizing the architecture and the implementation for the two classes of applications, are completely different due to the differences in the nature of the applications. DSP applications require a constant performance, which due to the nature of the application cannot be traded against a lower energy consumption. A spreadsheet or a word processor will still work correctly if some of the performance is traded for a reduction in energy consumption.

To allow comparisons across a range of - not necessarily binary compatible - architectures, it has been decided to base the architectural investigations in this dissertation on the $\frac{1}{J \cdot \sec}$ metric, even though this implies that it is not possible to optimize for all the classes of applications that the HORN processor architecture specifies.

This has had implications for the benchmark suite to be used for the rest of the investigation described hereafter. The benchmark suite should contain microprocessor applications such as compilers, filters and games, but it should *not* include applications such as MPEG (video compression/decompression) as it belong to a completely different class of application which, by choosing the $\frac{1}{J \cdot \sec}$ as 'base metric', *may* not be able to execute correctly if the performance requirement is not met.

This does not mean that DSP-applications and products cannot or should not be optimized for energy efficiency, only that the metric used for such optimizations is not the same as the one used for optimizing microprocessors and servers. Techniques which can be used for optimizing a microprocessor might not be the same as for a DSP-processor. This work has considered the HORN-processor only as a microprocessor and the remaining of this thesis will therefore investigate only the energy efficiency of microprocessors. DSP-processors and applications will not be mentioned again.

Consequently a benchmark suite of eight microprocessor and server benchmarks has been chosen. Each benchmark has been described briefly, and a motivation for including it in the benchmark suite has been given.

Chapter 5 Energy consumption in caches

As shown in Chapter 2, the cache consumes a significant proportion of the total power of a typical microprocessor. Understanding the effect that changing cache parameters and architecture has on cache power consumption is therefore essential when designing an energy efficient microprocessor system. The cache clearly affects not only the performance of the processor; it also reduces external memory traffic and thereby the power consumption of the entire system.

Based on capacitances derived under the OMI-DE-project [Garside2], this project has derived expressions for energy consumption in a number of cache architectures and analysed their suitability for an energy efficient processor architecture. Section 5.1.1 presents results collected from commercially available tools and a low power sense amplifier design designed by T. Burd [Burd2].

Based on results extracted from [Garside2], section 5.1.2 derives expressions for energy consumption in RAM. Sections 5.2-5.4 apply these results to a number of cache architectures described in the literature, and evaluate their potential for an 'energy efficient processor architecture'. The expressions used have been derived as a part of the author's work.

Section 5.5 discusses associative caches and evaluates the value of skewed associativity [Seznec] while section 5.6 evaluates the value of different replacement algorithms. Section 5.7 presents results of an analysis of cycle times of different cache organizations.

Sections 5.8-5.11 present techniques to reduce cache activity and thereby energy consumption. The evaluation has been carried out as part of this work.

Section 5.12 summarizes the results.

5.1 Energy cost

In this section expressions for the energy consumption of conventional direct-mapped and set-associative cache configurations will be derived to establish an understanding of the effect of various cache parameters on power consumption. Other cache architectures, such as sectored caching [Seznec2] and CAT-caching [Wang], will also be investigated. An expression for energy consumption of CAT-caches will be derived. Cache organizations, such as sub-caching [Su], which break up cachelines into sub-lines and hence reduce the size of the RAM block accessed[1], will not be investigated; they are not considered useful in architectures where cache references are not aligned on fixed byte boundaries. Throughout this section a 32-bit address- and data-bus is assumed.

5.1.1 RAM-compiler

Conventional cache designs may use static RAM-blocks such as those generated by a RAM-compiler [VLSI] employing a conventional sense amplifier design. The design of the sense amplifiers in this technology is such that they have a large static power dissipation. For RAM-blocks generated by the RAM compiler, the dissipation is therefore dominated by the sense amplifiers, see [VLSI]. As Table 5.1 shows, the line size, '*ls*' in

Table 5.1 Dynamic energy consumption in RAM [VLSI]

	Line size		
Lines	**32bits/4bytes [nJ/cycle]**	**64bits/8bytes [nJ/cycle]**	**128bits/16bytes[a] [nJ/cycle]**
128	4.78	9.55	19.09
256	4.82	9.59	19.17
512	4.86	9.73	19.45
1024	4.86	9.73	19.45

a. Data for longer lines were not available.

1. And thereby the energy consumption per request.

bits, is the dominant factor in the expression of energy consumption, and the dynamic energy consumption per access to the RAM-block can therefore be approximated to:

$$E_{RAM} \approx K \times ls \qquad \text{(EQ 5.1)}$$

where K is a proportionality factor equal to:

$$K = \frac{19.45 - 4.86}{128 - 32} \frac{nJ}{bit} = 0.15 \frac{nJ}{bit} \qquad \text{(EQ 5.2)}$$

This is due, in part, to the power consumption of the sense amplifiers, which have been designed to drive relatively large capacitances. The figures quoted in Table 5.1 assume an input capacitance, $C_{I/O,}$ per bit of ~1pF and an output load of 1pF. Such driving capacities are not necessary in a cache design where the sense amplifiers have to drive only the input of a multiplexer. Using sense amplifiers with lower driving capabilities will reduce the energy consumption proportionally, but it is not clear whether the static power consumption will scale, see below.

Due to the leakage in the sense amplifiers there is also a static dissipation, $P_{RAM,Static}$ of ~2.5mW per bit which cannot be neglected. For a RAM block with a 32-bit wide data bus and 1024 lines, cycled at 33MHz, the total power consumption, static and dynamic, can be calculated [VLSI]:

$$P_{RAM,(Dynamic)} = 4.86 mW/MHz \times 33 MHz = 160 mW \qquad \text{(EQ 5.3)}$$

$$P_{RAM(Static)} = 2.5 mW/bit \times 32 bit = 80 mW \qquad \text{(EQ 5.4)}$$

i.e. a 2:1 ratio. There might be several sense amplifiers per bit dependent on the internal organization of the RAM, although Equation 5.4 assumes only one sense amplifier per bit.

The static dissipation associated with the sense amplifiers therefore represents a significant proportion of the total power consumption of the RAM-block. If the sense amplifiers employed a dynamic circuit which is activated, and hence energy consuming, only when the RAM is accessed, other parameters in the RAM block such as the total size and line size may dominate the expression for energy consumption.

Figure 5.1 presents a design of a dynamic sense amplifier which does not have any static power dissipation. Building the sense amplifier and precharge circuit shown in Figure 5.1 and described in detail in [Burd2] eliminates the static power consumption in the sense amplifier almost[1] completely. The numbers in parenthesis indicate transistor dimensions:

M1, M3 and M4 form the precharge circuit. When 'Clk' is 'low' M1 will charge 'Node X' to V_{DD}, and the line '$\overline{bitline}$' to V_{DD}-V_T through M4. When 'Clk' goes 'high' M1 and M4 will be cut off, 'Node X' will discharge towards the value of $\overline{bitline}$ and 'Output' will switch to the value of the bit in the storage. Note that the threshold voltage of the inverter should be relatively high for the inverter output to switch to 'high' as quickly as possible. This is achieved by scaling the transistors in the inverter appropriately. Transistor M2 forms a weak feedback to the 'Node X' and maintains the value on the Output-node. Note that, unlike the circuit described in [VLSI], this is a dynamic circuit which requires a system clock or a similarly derived signal to function.

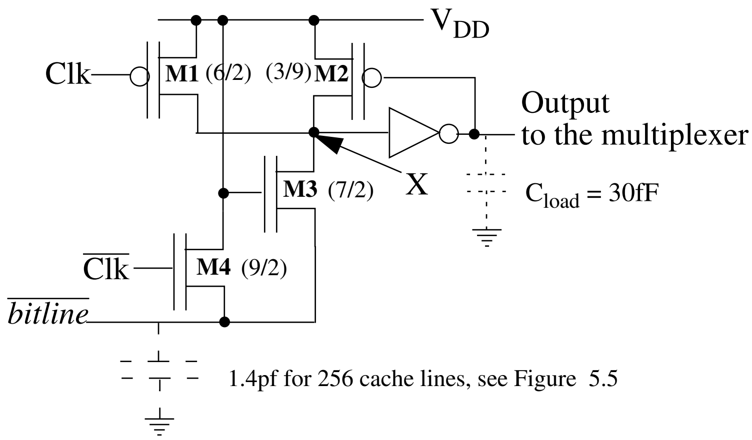


Figure 5.1 Sense amplifier without static power dissipation

1. A small static power consumption will remain due to leakage current through the transistors in the inverter and in the storage element (not shown); however it is considered negligible. [Burd2]

Transistors M1, M2, M3 and M4, in the circuit are small and the circuit capacitances are also very small. The capacitance on the $\overline{bitline}$ will be dominated by the capacitance of the storage block. Only the output from the inverter is expected to be energy consuming in that it will be the only node with a full voltage swing. The output of the sense amplifier will drive the input to a multiplexer and hence have a very small load. Current technology [Garside] specifies input capacitances of simple circuits such as multiplexers to be C_{in}~20fF. Adding some capacitance for routing, C_{load} is estimated at 30fF. Given an architecture where the sense amplifiers are placed *before* the output multiplexer, '*ls*' sense amplifiers are required. They will consume:

$$E_{Sense} = ls \times 0.03\frac{pF}{bit} \times (3V)^2 = ls \times 0.27\frac{pJ}{bit} \qquad \text{(EQ 5.5)}$$

per request.

Note also that the sense amplifiers should consume energy only during read cycles and as the normal reference pattern is two read requests per write request, the importance of the energy consumption of the sense amplifiers is reduced.

However, mixing results extrapolated from widely different technologies, such as [VLSI] and [Burd2], may lead to wrong conclusions. The rest of this chapter will therefore seek an understanding of how the cache parameters affect the energy consumption. The results will be used to extrapolate the results from OMI-MAP to a RAM of any dimension. The extrapolations are based on the cache implementation in the ARM3 [OMIMAP] and in Amulet2e [Garside][Garside2].

5.1.2 Fundamental relations

Details on technology issues such as bit-line and word-line capacitances is commercially sensitive information which can rarely be extracted from data sheets. To understand how the energy consumption of a RAM block scales with conventional cache parameters such as line-size and number of lines it was therefore decided to build an expression for energy consumption based on capacitances in the cache circuit extracted [Garside] for the Amulet2e project [Garside2] which uses a 0.6μ, three layer metal, CMOS process.

Figure 5.2 shows a basic static RAM memory cell as used in caches; the line capacitances shown correspond to a 256 lines x 256 bits (8K bytes) configuration. In addition to the storage circuit itself, a pre-charge circuit, a sense amplifier and an output multiplexer are shown. The capacitances shown in Figure 5.2 and those mentioned later in this chapter, have been extracted from the Amulet2e design [Garside].

The general expression for energy consumption, E, in a CMOS circuit with N nodes is [Mead]:

$$E = \frac{1}{2}\sum_{i=1}^{N} T_i \times C_i \times (\Delta V_i)^2 \qquad \text{(EQ 5.6)}$$

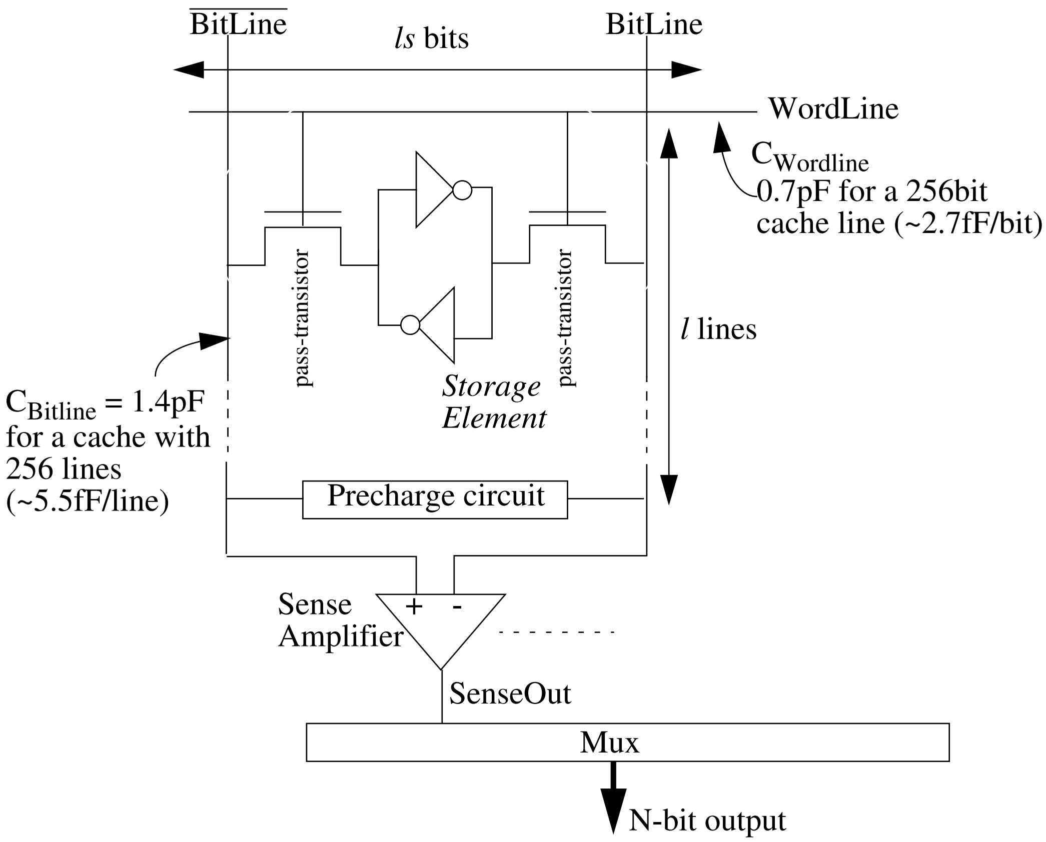


Figure 5.2 Extract from RAM circuit

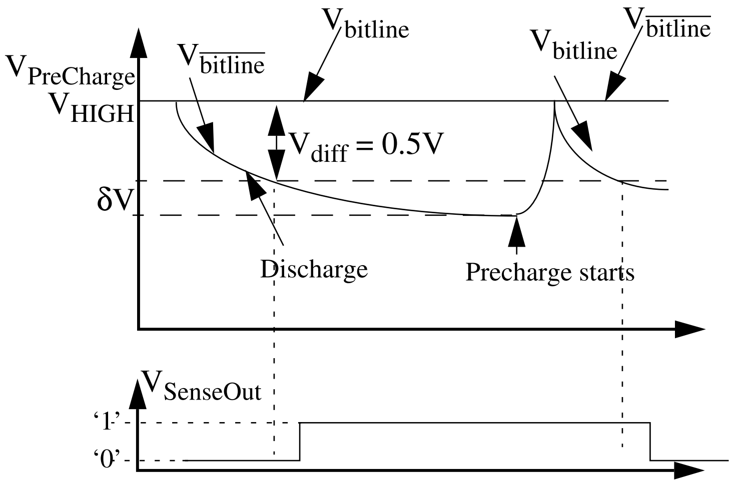


Figure 5.3 Voltage swing when discharging and precharging bit lines

where T_i denotes the number of transitions in the i'th node. C_i denotes the capacitance and ΔV_i the voltage swing of the i'th node. The voltage swings on the word- and bit-lines differ due to the different driving sources. The word-line is driven by a decode circuit, i.e. a gate output with good driving capabilities and a full logical voltage swing of 3V is expected. BitLine and $\overline{\text{BitLine}}$ are driven by the storage element, through the pass-transistors. This part of the circuit will be designed with very small transistors to optimize silicon area and will therefore have relatively low driving capabilities. Bit-line capacitances are twice those on the word-line (Figure 5.2) implying that voltage change on the bit-lines will be slower. The sense amplifier is designed to 'sense' the value of the storage element before the full logical voltage swing has been encountered. A bit-line voltage difference of $V_{diff} = 0.5V$ is normally sufficient for the sense amplifier circuit to detect the value of the cell [Weste] [Burd2]. However it is important to note that the voltage on the discharging bit-line will keep falling after the logical value of the cell has been detected, see Figure 5.3. The overshoot, δ, can be adjusted by scaling the transistor sizes in the storage cells [Weste].

The energy consumed during a cycle (discharge and precharge) is:

$$E_{Bitline} = C_{Bitline} \times (V_{diff} + \delta)^2 \qquad \text{(EQ 5.7)}$$

However, if the bit lines are not precharged to high, but to an intermediate voltage, $V_{PreCharge}$, see Figure 5.4, a significant amount of energy can be saved [Weste]. The access time might suffer, depending on how fast the storage cell can charge/discharge the bit lines. Discharging one bit line as shown in Figure 5.3, sufficiently for a sense amplifier to detect the value of the cell may be faster than the scheme presented in Figure 5.4. However careful design should minimize this penalty.

The energy consumption throughout the discharge and precharge in such a circuit is:

$$E_{Bitlines} = C_{Bitline} \times (\Delta V_{Bitline})^2 + C_{\overline{Bitline}} \times (\Delta V_{\overline{Bitline}})^2 \qquad \text{(EQ 5.8)}$$

If symmetry is assumed, i.e $V_{Precharge} = \frac{V_{High} - V_{Low}}{2}$, the expression for $E_{Bitlines}$ reduces to:

$$E_{Bitlines} = 2 \times C_{Bitline} \times \left(\frac{V_{diff}}{2} + \delta\right)^2 \qquad \text{(EQ 5.9)}$$

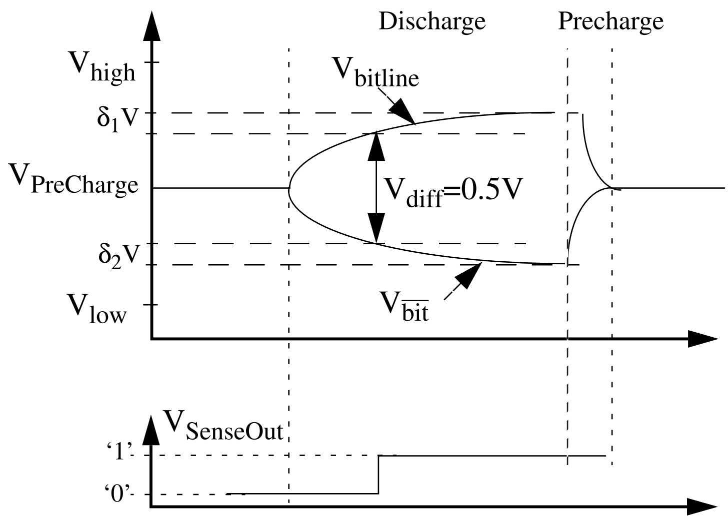


Figure 5.4 $V_{Bitline}$ when precharging to an intermediate voltage

The energy consumption of the storage part of the RAM can be approximated to:

$$E_{Storage} = E_{Wordline} + E_{Bitlines} \tag{EQ 5.10}$$

By substitution this can be expressed as:

$$E_{Storage,Read} = C_{Wordline} \times (\Delta V_{Wordline})^2 + 2 \times C_{Bitline} \times (\Delta V_{Bitine})^2 \times l \tag{EQ 5.11}$$

To simplify the calculations in the rest of this thesis it will be assumed that:

$$2 \times \left(\frac{V_{diff}}{2} + \delta\right)^2 = \frac{(V_{diff})^2}{2} \tag{EQ 5.12}$$

in which case Equation 5.11 reduces to:

$$E_{Storage,Read} = C_{Wordline} \times (\Delta V_{Wordline})^2 + C_{Bitlines} \times (V_{diff})^2 \times l \tag{EQ 5.13}$$

By scaling the capacitances in Figure 5.2, the energy consumption of the upper (storage part) of the circuit during a read cycle is:

$$E_{Storage,Read} = 2.7\frac{fF}{bit} \times ls \times (3V)^2 + 5.5\frac{fF}{bit} \times ls \times (0.5V)^2 \times l \tag{EQ 5.14}$$

Where '*ls*' denotes number of *bits* per cache line and '*l*' denotes the number of lines in the storage block, see Figure 5.2.

Scaling this expression shows how the energy consumption changes with *l* and *ls*:

$$E_{Storage,Read} \propto K_1 \times ls + ls \times l \tag{EQ 5.15}$$

where

$$K_1 = \frac{C_{Wordline} \times (\Delta V_{Wordline})^2}{C_{Bitline/wline} \times (\Delta V_{diff})^2} = \frac{2.7fF/bit \times (3V)^2}{5.5fF/bit \times (0.5V)^2} = 18 \tag{EQ 5.16}$$

This expression says that the energy consumption in RAM is more sensitive to changes in line size, *ls*, than to changes in number of lines, *l*, at least for small values of *l*. Increasing the RAM size by increasing the number of lines therefore appears more attractive than increasing the line size or any combination of the two.

During a read cycle the bit-lines do not need to discharge completely before the sense amplifiers can detect the voltage difference and determine the value of the memory cell. The bit-line will therefore only consume energy corresponding to V_{diff}=0.5V as shown in Equation 5.14. During a write cycle, the bit-line is driven by an external source and a total voltage swing of 3V can be anticipated. However, it will only be the bits in the word on the line which get overwritten which will experience this magnitude of voltage swing.

The remaining bits will discharge even though they are not accessed but will (dis-)charge as during a read-cycle. The energy consumption during a write is therefore approximated to:

$$E_{Storage,write} = C_{Wordline} \times \left(\Delta V_{wordline}\right)^2 + C_{Bitline/wline} \times \left(\Delta V_{Bitline,Write}\right)^2 \times l + C_{BitlineS} \times \left(\Delta V_{diff,Idle}\right)^2 \times l \qquad \text{(EQ 5.17)}$$

Substituting the capacitance and voltage values from the Amulet2e project gives the following expressions:

$$E_{Storage,write} = 2.7\frac{fF}{bit} \times (3V)^2 \times ls + 5.5\frac{fF}{bit} \times w \times (3V)^2 \times l + 5.5\frac{fF}{bit} \times (0.5V)^2 \times (ls - w) \times l \qquad \text{(EQ 5.18)}$$

where '*w*' signifies the size of the word (in bits) which is written to the storage.

Scaling as before yields:

$$E_{Storage,write} \propto K_1 \times ls + K_2 \times w \times l + 1 \times (ls - w) \times l \qquad \text{(EQ 5.19)}$$

$$E_{Storage,write} \propto K_1 \times ls + l \times (ls + w \times (K_2 - 1)) \qquad \text{(EQ 5.20)}$$

where K_1 is defined above and K_2 is:

$$K_2 = \frac{C_{Bitline/wordline} \times (\Delta V_{Bitline,Write})^2}{C_{Bitline/wordline} \times (\Delta V_{diff,Idle})^2} = \frac{5.5fF \times (3V)^2}{5.5fF \times (0.5V)^2} = 36 \qquad \text{(EQ 5.21)}$$

Inserting w= 32 bits in Equation 5.20 gives:

$$E_{Storage,write} \propto K_1 \times ls + 32 \times K_2 \times l + 1 \times (ls - 32) \times l \qquad \text{(EQ 5.22)}$$

and thus:

$$E_{Storage,write} \propto K_1 \times ls + l \times (K_2 + ls - 32) \qquad \text{(EQ 5.23)}$$

Tables 5.2-5.3, show how the energy consumption during a read and a write cycle differs widely due to the activating of the sense amplifiers. In contrast to the sense amplifiers used in [VLSI], there is no static power dissipation in the sense amplifiers used by ARM. The tables show that the sense amplifiers do not consume any power during write cycles.

Table 5.2 ARM3 RAM dissipation - Pre-charge/*Read* Cycle

Block	Average power consumption [mW]	% of total power in RAM
The main RAM array	162.3	40.6 ($X_{RAM,R}$)
I/O buffers	68.3	17.1 ($X_{I/O,R}$)
32-Sense Amps	81.0	20.3 (X_{Sense})
Precharge	29.6	7.4 ($X_{Pre,R}$)
Other blocks	58.8	14.6 ($X_{Other,R}$)
Total	**400.0**	**100**

Table 5.3 ARM3 RAM dissipation - Pre-charge/*Write* Cycle

Block	Average power consumption [mW]	% of total power in RAM
The main RAM array	162.3	64.9 ($X_{RAM,W}$)
I/O buffers	0	0.0 ($X_{I/O,W}$)
32-Sense Amps	0	0.0 (X_{Sense})
Precharge	24.9+4.7	11.8 ($X_{Pre,W}$)
Other blocks	58.1	23.2 ($X_{Other,W}$)
Total	**250.0**	**100**

It is therefore clear that an expression for energy consumption in a cache is a sum of two products: One product for read accesses and one for write accesses:

$$E_{Cache} = \#\text{reads} \times E_{read} + \#\text{writes} \times E_{write} \qquad \text{(EQ 5.24)}$$

General expressions for E_{read} and E_{write} are derived through extrapolations from the numbers in Tables 5.2 - 5.3.

The energy consumption in RAM scales as shown in Equation 5.15, while the energy consumption of I/O buffers and sense amplifiers scale linearly with the word-size i.e the number of bits to be read. The energy consumption of the precharge circuit scales with the length and the number of bitlines i.e. with size of the RAM. As will be shown in Chapter 8, an entire cacheline will be read every time the cache is accessed. It is therefore necessary to have sense amplifiers on each bit-line-pair in the memory. Due to the nature of ‘other blocks’ it will be assumed that their energy consumption is not affected - or only affected in a sub-linear way - by the cache size and line size. This is therefore an overhead which is carried with every RAM-block.

The numbers quoted are for a 4K-byte cache organized in lines of 4 bytes; a general expression for $E_{RAM,Read}$ is therefore:

$$\frac{E_{RAM,Read}}{E_{RAM,ARM3}} = X_{RAM,R} \times \frac{E_{Storage}}{E_{4K,4bytes}} + \left(X_{I/O,R} + X_{Sense}\right) \times \frac{E_{Sense+Buffer}}{E_{Sense+Buffer,ARM3}} + X_{Pre,R} \times \frac{E_{Pre}}{E_{Pre,ARM3}} + X_{Other,R} \qquad \text{(EQ 5.25)}$$

where X_y are the percentages shown in table 5.2. The energy consumption in storage during read-cycles scales as shown in Equation 5.15 and the energy consumption in the sense amplifiers scales with their number. Equally the energy consumption in the precharge circuit scales with the number of bit lines to be precharged and with the length of the bit lines, i.e. it will scale with the size of the storage block. Given that the RAM in the ARM3 cache is a 1Kx32bit RAM block, the relations in Equation 5.25 gives the following expression for $E_{RAM,Read}$:

$$\frac{E_{RAM,Read}}{E_{RAM,ARM3}} = X_{RAM,R} \times \frac{K_1 \times ls + ls \times l}{K_1 \times 32 + 1024 \times 32} + (X_{I/O,R} + X_{Sense}) \times \frac{ls}{32} + X_{Pre,R} \times \frac{ls \times l}{32 \times 1024} + X_{Other,R} \quad \text{(EQ 5.26)}$$

Equally the expression for $E_{RAM,Write}$ is:

$$\frac{E_{RAM,Write}}{E_{RAM,Write,ARM3}} = X_{RAM,W} \times \frac{E_{Storage}}{E_{Storage4K,4bytes}} + X_{Pre,W} \times \frac{E_{Precharge}}{E_{Prehage,ARM}} + X_{Other,W} \quad \text{(EQ 5.27)}$$

which by inserting Equation 5.23 for $E_{storage}$ is:

$$\frac{E_{RAM,Write}}{E_{RAM,Write,ARM3}} = 0.649 \times \frac{K_1 \times ls + l \times (K_2 + ls - w)}{K_1 \times 32 + 1024 \times (32 + 32 \times (K_2 - 1))} + 0.118 \times \frac{ls \times l}{1024 \times 32} + 0.232 \quad \text{(EQ 5.28)}$$

Chapter 2 showed that the power consumption of the in the ARM3-cache RAM-block is 400mW during read cycles, while it is 250mW during write cycles. Equations 5.26 and 5.28 show how these consumptions will scale when the dimension of the cache RAM changes.

5.1.3 Multi-ported RAM

An example of a multi-ported RAM is shown in Figure 5.5 [Weste]. The voltage swings on the lines are the same as for the single ported RAM. The capacitances on the bit- and word-lines are clearly higher than those shown for a single ported RAM, see Figure 5.2, due to the larger cell area[1]. However, if this increase in line capacitances is ignored, an expression for energy consumption in the multi-ported RAM with N read ports and M write ports can be approximated to:

$$E_{Multiport} = N \times E_{Read,SinglePort} + M \times E_{Write,SinglePort} \quad \text{(EQ 5.29)}$$

1. Extra pass-transistors make the cell wider and extra word-lines makes it higher

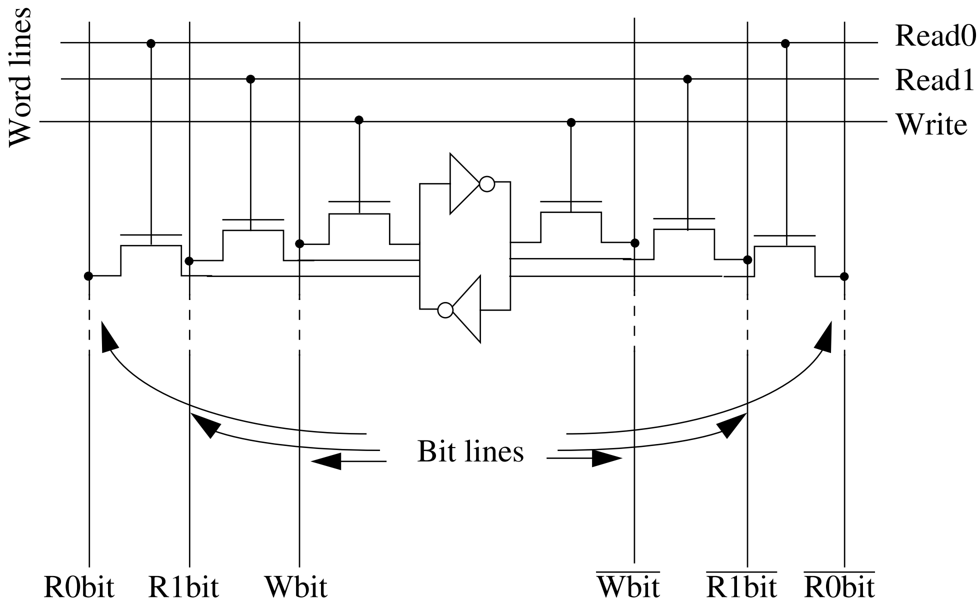


Figure 5.5 Extract of a bit cell from a multi-ported RAM circuit

The expressions for $E_{Read,Singleport}$ and $E_{Write,SinglePort}$ are those derived in the previous section. The approximation made above - ignoring the increase in line capacitances due to the increase in cell-dimensions - becomes less accurate as 'N' and 'M' increase.

Notice that M and N are the number of active ports, i.e. a non-active port should be disabled and hence not consume any energy [VLSI][Garside][Yeung].

5.2 Direct mapped cache

Figure 5.6 shows a block diagram of a M-byte[1] direct mapped cache with *l* lines of *w* words[2], in a *S*-bit address space[3]. The design is different from conventional cache designs in that every bit in the line which is read out of the Data Storage is 'sense amplified'. The reason for this will be explained in Chapter 8.

There are essentially three different types of accesses to a cache:

1. $b = \log_2(M)$
2. Each 32 bits
3. The sense amplifiers on the output of the tag storage are not shown

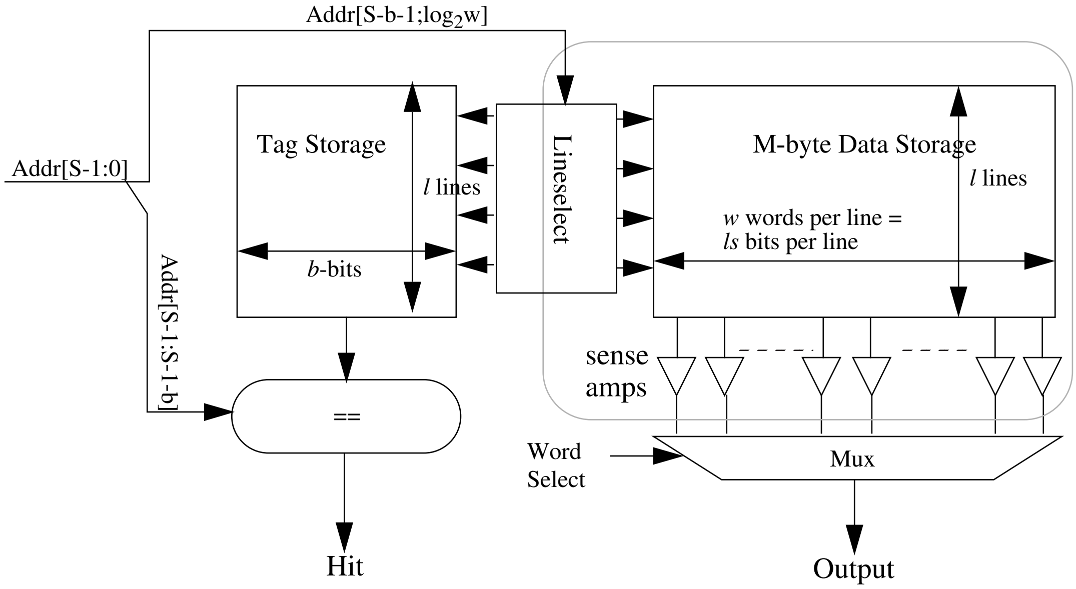


Figure 5.6 Block diagram for a direct mapped cache

1. A read access from both the Data Storage and the Tag Storage. Given the measurements from Chapter 2, this is the most power consuming type of access.
2. A write access to both the Data Storage and to the Tag Storage. This happens when the first word is written to the line following a miss.
3. A read in the Tag Storage and a write to the Data Storage. This is the case when a store-instruction hits in a Data cache.

The energy consumption of the three types of accesses will be denoted: E_{RR}, E_{WW} and E_{RW}.

Following a cache miss, the new tag will be written with the first word of the new line, i.e. a WW-access. The writing of the remaining words in the new line will be considered as RW-accesses. Given long ($\geq$ 4 words) cachelines and a high hit-rate, the number of cycles where there is a write to both the tag- and data storage and hence the frequency of E_{WW} type accesses is very low. In the expressions for energy consumption of the different cache architectures to be explored in the following sections, each instance of a WW-type operation will - for energy consumption purposes - be replaced by a RW-type access. This can be seen as a conservative replacement as the RW-type access is more energy-consuming, see section 5.1.

The rest of this chapter will derive expressions for E_{RR} and E_{RW} for a number of different cache architectures.

The energy consumption of the cache, in case of a hit, can be expressed as:

$$E_{Request} = E_{RAM,TAG} + E_{Compare} + E_{RAM,Storage} + E_{Mux} \quad \text{(EQ 5.30)}$$

As the complexity, and hence the energy consumption of the Mux and the compare circuit only scales with the log(line size) and log(cache size), the energy consumption of these circuits is approximated to be constant across the cache configurations considered:

$$E_{Request} = E_{RAM,TAG} + E_{RAM,Storage} + \text{Const} \quad \text{(EQ 5.31)}$$

Each access to RAM has a fixed energy cost proportional to the length of the wires charged or discharged [Mead], thus the *power* consumption in RAM is largely proportional to the frequency and type of requests. As with all CMOS designs there is a leakage current and hence some static power consumption, but it is small enough to be neglected [Mead].

The results from section 5.1.2 indicate that the energy consumption per access to the RAM blocks, Tag and Storage is a function of the number of cache lines, *l*, and especially of their total bit-width, *b* and *ls,* see Equations 5.26 - 5.28.

Assume an ‘M’-byte cache i.e.:

$$8M = ls \times l \quad \text{(EQ 5.32)}$$

By simple substitution in Equation 5.26 (*ls* replaced by *ls* + *b*) and normalization, the expression for E_{RR} is proportional to:

$$E_{Cache,RR} \propto 824 \times (ls + b) + l \times (ls + b) + 10144 \qquad \text{(EQ 5.33)}$$

The expression for E_{RW} is more complex as it includes both a read operation from the Tag storage and a write operation to the data storage:

$$E_{Cache,\,RW} \propto 824 \times b + 1 \times l \times b + 24.3 \times ls + 1.6 \times ls \times l + 26250 \qquad \text{(EQ 5.34)}$$

where $b = S - \log_2 M$.

Substituting l with $8M/ls$ gives:

$$E_{Cache,RR} \propto \left(103 + \frac{M}{ls}\right) \times (ls + b) + 1268 \qquad \text{(EQ 5.35)}$$

and

$$E_{Cache,\,RW} \propto ls + (498 + 3.4 \times ls) \times \frac{M}{ls} + 1202 \times b + 11.6 \times b \times \frac{M}{ls} + 38181 \qquad \text{(EQ 5.36)}$$

$$E_{Cache,\,RW} \propto (ls + 1202 \times b) + (498 + 3.4 \times ls + 11.6 \times b) \times \frac{M}{ls} + 38181 \qquad \text{(EQ 5.37)}$$

Analysing these expressions shows that $E_{RAM,RR}$ increases almost linearly with the size of the cache, *M*, and with the line size, *ls*, except for very small values of *ls,* Figure 5.7.

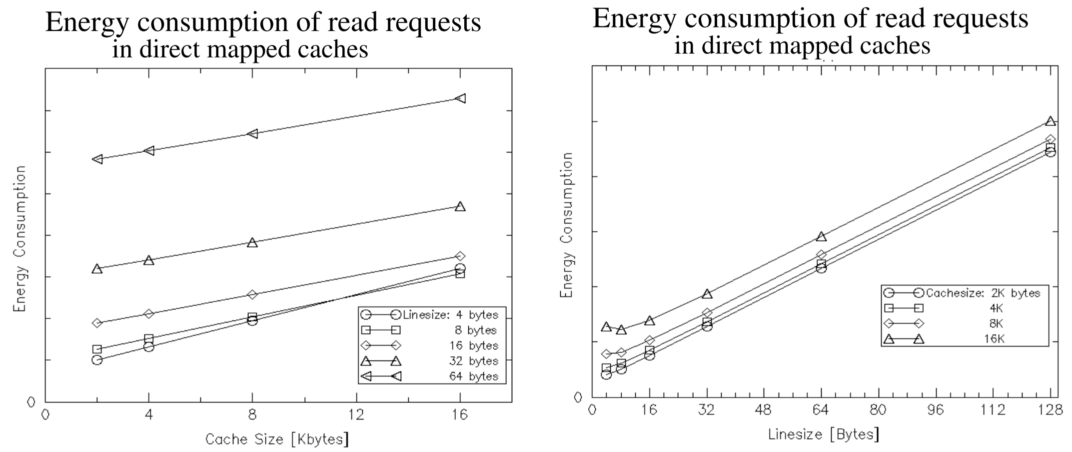


Figure 5.7 $E_{Cache,RR}$ vs. cache size and $E_{Cache,RR}$ vs. line size

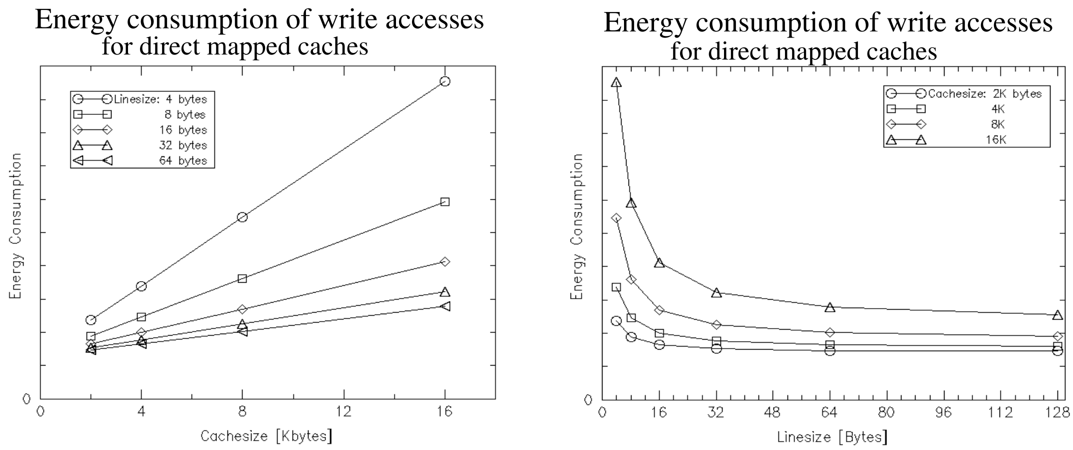


Figure 5.8 $E_{Cache,RW}$ vs. cache size and $E_{Cache,RW}$ vs. line size

$E_{RAM,RW}$ increases linearly with the cache size, M, but *decreases* inversely proportionally with the line size, *ls*, see Figure 5.8.

Given the general expressions for energy consumption in a cache, as described in Equation 5.33 and 5.36, the power consumption of the cache is proportional to the frequency of access:

(EQ 5.38)

$$P_{Cache,Total} \propto \frac{\left(Req_{Read} + Writebacks \times w\right) \times E_{Cache,RR} + \left(Req_{Write} + Miss \times w\right) \times E_{Cache,RW}}{Cycletime \times cycles}$$

This has assumed a conventional direct-mapped cache architecture. More sophisticated cache architectures/technologies such as CAT-caching [Wang] or sectored caching [Seznec2] will change these equations, see section 5.4.

5.3 N-way set-associative caches

Figure 5.9 shows an N-way set-associative cache comprising *N* directly addressed sub-caches, each 1/*N*th of the total cache size[1]. Requesting a word implies accessing all N sets in parallel and if there is a hit the requested word is read from the set with matching tag.

1. The sense amplifiers on the tag storage blocks are not shown

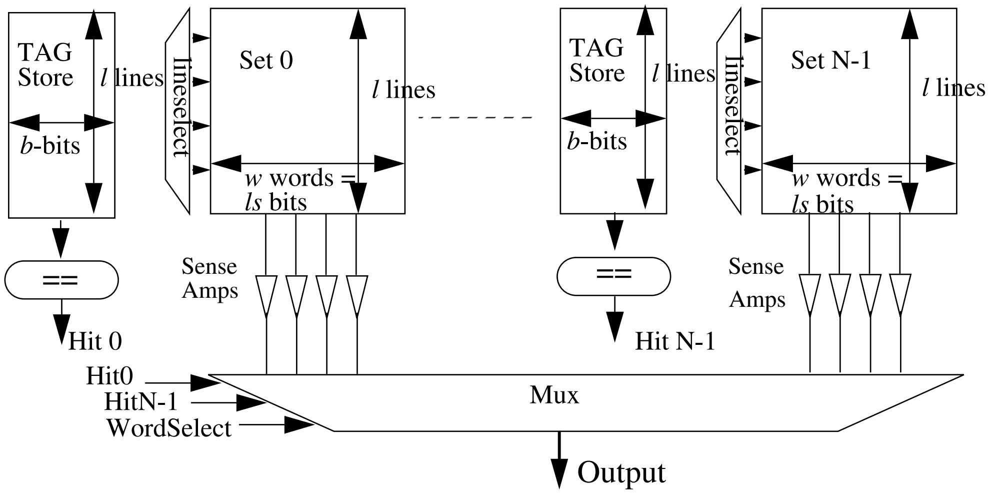


Figure 5.9 N-way set-associative cache

For a total cache size of M-bytes, each of N-sets will contain M/N bytes of storage; for l lines in each set, b is given by:

$$b = S - \log_2\left(\frac{M}{N}\right) \qquad \text{(EQ 5.39)}$$

and

$$M = \frac{N \times ls \times l}{8\,\text{bits/byte}} \qquad \text{(EQ 5.40)}$$

The expressions for $E_{Cache,RR}$ and $E_{Cache,RW}$ thus become:

$$E_{Cache,RR} \propto N \times \left(\left(\frac{M}{N \times ls} + 103\right) \times (ls + b) + 1268\right) \qquad \text{(EQ 5.41)}$$

and

$$E_{Cache,\,RW} \propto N \times \left(ls + 1202 \times b + (498 + 3.4 \times ls + 11.6 \times b) \times \frac{M}{N \times ls} + 38181\right) \qquad \text{(EQ 5.42)}$$

As was the case for the direct mapped cache in section 5.2, the expressions are more sensitive to changing *ls* than to changing 'M'. Figure 5.10 shows how the energy consumption increases for increasing 'N' and line size, '*ls*'. The graph also shows that the expression for energy consumption is much more sensitive to 'N' than to '*ls*'. A similar graph (and similar conclusion) can be drawn for $E_{Cache,RW}$.

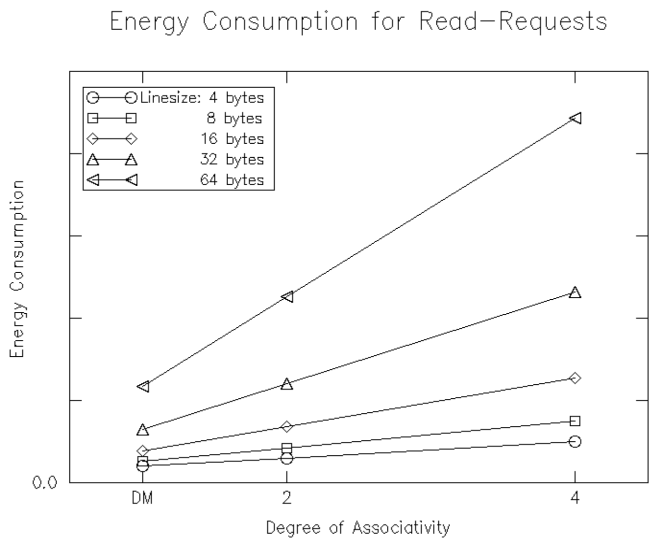


Figure 5.10 $E_{Cache,RR}$ vs. degree of associativity for a 8K-byte cache

This result, with that from section 5.2, suggests that as the energy expressions are more sensitive to increased line size and to increased associativity than to increased size of the cache, a large, direct-mapped cache with short cache lines might be more desirable than a smaller cache with some form of associativity.

5.4 Other cache organizations

Cache designs, such as the ones described above, associate a set of tag bits with each cache line as shown in Figures 5.6 and 5.9. For an 8K-byte cache with 32-byte lines the tags bits occupy approximately 7% of the total chip area taken by the cache, dependent on the degree of associativity given a 32-bit address space:

$$TotalLinesize = Linesize + Tagsize \tag{EQ 5.43}$$

The overhead decreases linearly with increasing line lengths. According to Equations 5.33 and 5.36, this overhead translates into energy consumption. As the number of bits in the address space increases so does the number of bits in the tag store. For a 32-bit address space, the tag storage represents ‘only’ 7% of the area/energy

$$TagOverhead_{8M,32\frac{Bytes}{Line},DM} = \frac{Tagsize}{TotalLinesize} = \frac{32 - \log_2 8192}{32 - \log_2 8192 + 32 \times 8} = 6.9\%$$

consumptions of the cache. However, if a 64-bit address space is considered the tag store would occupy 20% of the area of the cache and consume a significant proportion of total energy consumed, see for example Equation 5.33.

Several techniques to reduce this overhead have been studied. Sectored caching [Seznec] and Cache Address Tag (CAT-)caching [Wang] have proved to be most promising.

Both techniques exploit the spatial locality in data and instructions further than the architectures described in sections 5.2 - 5.3, where many of the tags stored will be identical. Statistics collected with the HORN-architecture tools have shown that the

Table 5.4 Tag distribution - 8K byte unified cache, Direct mapped, 256 lines

	Number of different tags present in Cache at any time[a]									
Benchmark	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
cacti	0.1	29.9	0.0	0.2	69.5	0.2	0.0	-	-	-
dhry	2.0	1.2	0.5	12.5	82.9	0.7	0.2	-	-	-
espresso	0.7	0.0	57.9	28.2	0.7	0.2	0.7	2.0	9.6	-
fft	0.0	9.7	2.4	82.9	2.4	2.4	0.1	-	-	-
flex	0.0	3.9	0.9	5.7	0.9	84.2	0.2	4.8	0.0	-
hello	1.3	0.8	93.2	0.6	4.1	-	-	-	-	-
stcompiler	1.1	0.9	13.0	23.5	20.5	18.2	22.5	0.3	-	-

a. Format: ‘x’ in column ‘y’ indicates that there were only ‘y’ *different* tags in the cache in ‘x’ percent of the cycles. 0.0 indicates that a there were ‘y’ different tags present less than 0.05% of the cycles while ‘-’ means that there were never ‘y’ different tags.

number of different tags present in a unified cache at any time during the execution of a program is very low, see Table 5.4. No benchmark had, at any time, more than 9 different tags present in the cache and only in very few cases here there more than 8 different tags present. There is thus great redundancy in the Tag storage; this can be exploited to reduce the size, as well as the energy consumption, of the cache.

5.4.1 Sectored caching

The principle of sectored caching as described in [Seznec] is shown in Figure 5.11. Instead of associating a tag with each line a tag is associated with a sector comprising a number of lines, in this case 8. The larger the sectors the fewer tags and hence a smaller tag-overhead. The results given in Table 5.4 show that the unified cache only ever contains 9 different tags. A cache with 16 tags and hence 16 sectors will therefore be sufficient to supply the need for tag store. In a cache with 256 lines it means that the tag store can be reduced by 15/16 (93%) resulting in an overall reduction in storage of 6.5% in a 32-bit address space. This reduction in storage will also imply a reduced energy consumption per cache access. The saving increases with the number of bits in the address space.

Although several sectors may hold the same tag, the scheme is not very flexible since the number of lines per sector is defined at design time. This can lead to sectors which are not fully used and therefore an under utilization of the cache. There is clearly a trade-off to be made between fewer tags - and hence lower energy consumption per request - and more tags, better utilization and higher hit-rate.

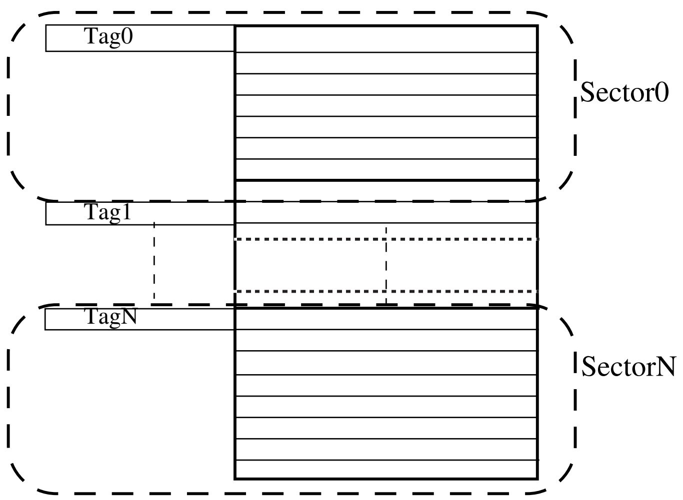


Figure 5.11 Sectored cache

The architecture might also have another application in a low-power environment: Section 5.1 showed how the line size is a significant factor in the expression for energy consumption of a cache. Keeping a short line size implies lower energy consumption per access. However, shorter lines do normally imply a significant tag-overhead.

[Uhlig] described a technique where the N lines succeeding a line which missed are fetched into the instruction cache following a miss. One of the results of the paper is repeated in Table 5.5. The table shows how reducing the line size and increasing the number of lines prefetched increases the performance of the system: For example, a cache with 16-byte lines which prefetches one line performs better than a cache with 32-byte lines without any prefetch.

If a sector is considered as one long cacheline, many of the advantages of a cache with long lines is maintained and the energy consumption of each request is reduced. The results in Table 5.5 suggest that fetching a sector, for example 4 short lines, would perform as well or better than fetching one longer line. The tag overhead associated with this is small (2 bits per line, 11% for a 8K direct mapped cache) but would involve a lower energy consumption (energy consumption scales with the line size).

Table 5.5 ΔCPI versus line size and prefetch distance[Uhlig]

Number of lines prefetched (N)	**Line size[a] [bytes] (M)**		
	16	**32**	**64**
0	0.439	*0.335*	**0.297**
1	*0.305*	**0.271**	-[b]
2	0.270	-	-
3	**0.260**	-	-

a. 8K direct mapped instruction cache

b. "-" denote points which are either not reasonable, or that shows an increase in CPI

5.4.2 Cache Address Tag-caching

To decouple the storage of the full tag field from its associated data items Cache Address Tag (CAT-)caching [Wang] has been proposed. The principle of CAT-caching is shown in Figure 5.12.

This avoids some of the limitations of the sectored cache architecture described above. In the CAT-cache there is no fixed allocation of tag-bits to individual cachelines. A cacheline links itself to a tag-value in the CAT-cache with a pointer, Ptr in Figure 5.12. There can be a variable number of cachelines associated with each tag in the CAT-cache.

There is an overhead associated with the CAT-cache in that storage is required for the pointer, Ptr. However, as the results in section 5.4 showed, the number of tags which need to be stored is very low and consequently l_2 should be small and the size of the pointer correspondingly small. The total amount of storage and thereby the energy consumption of the CAT-cache is therefore smaller than in any of the architectures explored above. This advantage increases with the number of bits in the address space, *S*. The amount of storage in a set-associative cache increases significantly when S increases from 32 bits to,

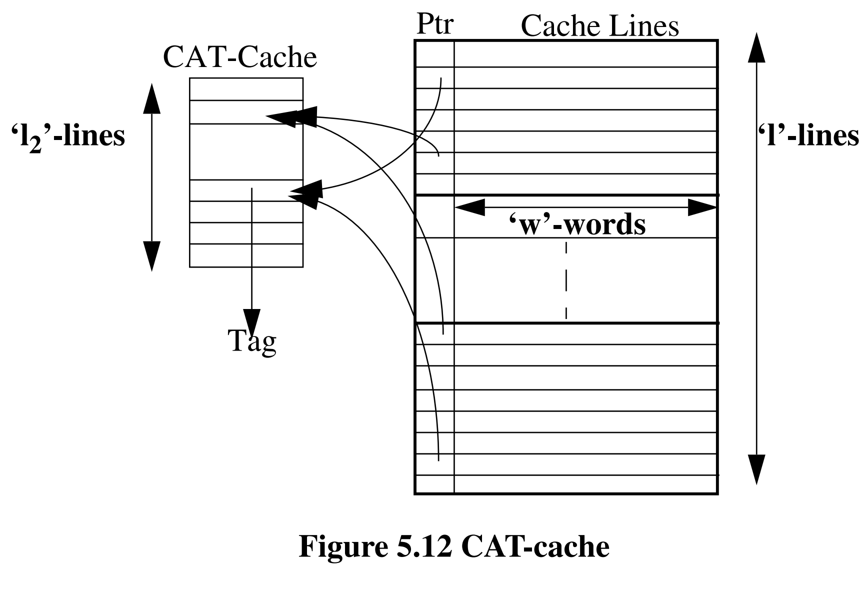


Figure 5.12 CAT-cache

for example, 64 bits. In a CAT-cache this need not be the case. If l_2 is as low as indicated then the size of the CAT-cache is only a fraction of the size of the Data Storage:

Consider an 8K-byte, direct mapped, unified cache, with 32-byte cache lines. The simulation results from Table 5.4 indicated that a nine-line CAT-cache would be sufficient (l_2=9). In a 32-bit system, the size of the CAT-cache would be ~0.3% of the size of the data storage; add to this a 4-bit pointer on every cacheline and the total overhead, $\frac{TotalSize - SizeOfData}{TotalSize}$, is 1.8%. In a 64-bit system the overhead would be 2.3%. These figures should be compared against the storage overhead in a conventional direct mapped cache: 7.4% in the 32-bit environment and 20% in a 64-bit environment. These are significant reductions, which clearly will affect the energy consumption of the cache.

The advantage improves with the number of lines in the cache. Shorter lines will therefore gain most from CAT-caching. Equation 5.15 on page 76 shows that energy consumption in RAM is very sensitive to the line size. The CAT-cache is therefore a powerful and energy efficient architecture.

The architecture might appear to provide a slow cycle time due to the sequential nature of the look-up. Firstly a direct-addressed lookup in the data area where a pointer is fetched and secondly a lookup in the CAT-cache from which the tag is extracted.

If the CAT-cache is implemented as a Content Addressable Memory (CAM), the lookup in the CAM can be done in parallel with the access to the data area, and the encoded position of the matched tag from the CAM storage can be compared with the pointer from the data area. However, a CAM cell consumes considerably more power that an ordinary RAM cell used in the set-associative cache, see Chapter 2.

The value of the CAT cache therefore depends on the ratio between the energy consumption in a CAM cell and that in a normal RAM-cell and the relative size of tag-storage in the two systems.

The figures quoted in Chapter 2, indicate that a block of 64 x 22 CAM cells consume 100mW while a 1024 x 32 RAM block consume 332mW; i.e. an energy consumption ratio per cell of 5:1. Given that the required number of entries in the CAT-cache is much smaller than the number of lines in the cache, the CAT-cache architecture may be an energy efficient alternative to a conventional cache, depending on the number of cachelines.

Maintaining a CAT-cache is complicated, especially if the there is an insufficient number of entries in the CAT-cache and multiple dirty lines have to be identified and written back to the main memory. The performance implications of a miss in a CAT-cache have therefore not been investigated here. Further assessment of the CAT-cache is recommended as a fruitful area for future research.

5.5 Skewed-associativity

Cache performance is normally optimized by adjusting parameters such as size, line size and degree of associativity. The size and the line size are normally chosen relatively freely, within the constraints of the total chip area available, while the degree of associativity is often limited by other constraints: the designer may choose a direct-mapped or a 2-way set-associative cache configuration because it is fast and not very power consuming, or a fully associative cache because it will yield the best hit-rate. Unfortunately, a fully associative cache is significantly slower than a 2-way set-associative cache and typically will be more power consuming. This makes it desirable to use a lower degree of associativity and to find other ways of improving the hit rate.

Figure 5.13 shows the principles of set- and skewed-associativity [Seznec], [Bodin], [Hilditch]. In the set-associative cache, a given address will be checked against the same line in each set, while in the skewed associative approach the two skewing functions, *Phi0* and *Phi1*, skew the line numbers so that, for a given address, different lines are accessed in each set. As a consequence of the skewing, two addresses that map to the same line in 'Set 0' may not both map to the same line in 'Set 1'.

Skewed associativity distributes the usage of the cache lines in a set-associative cache using different line mapping functions. As section 5.3 showed that energy consumption increases significantly with the number of sets in the cache, the degree of associativity should be kept low. Consequently, only 2-way skewed-associativity is discussed here although, in general, a N-way skewed-associative cache can be built.

Consider a cache referencing address, A. When accessing a conventional direct-mapped or set-associative cache, the bits in A are divided up into three fields, A1(MSB), A2 and A3 (LSB), where A2 is used to select the lines in the sets, A1 is the tag, and A3 is the byte offset within the line.

In a skewed associative cache A1 is split into two parts, $A1_1$ and $A1_2$, where $A1_2$ contains the same number of bits as the A2 part, used to select lines. The skewing functions, *Phi0*

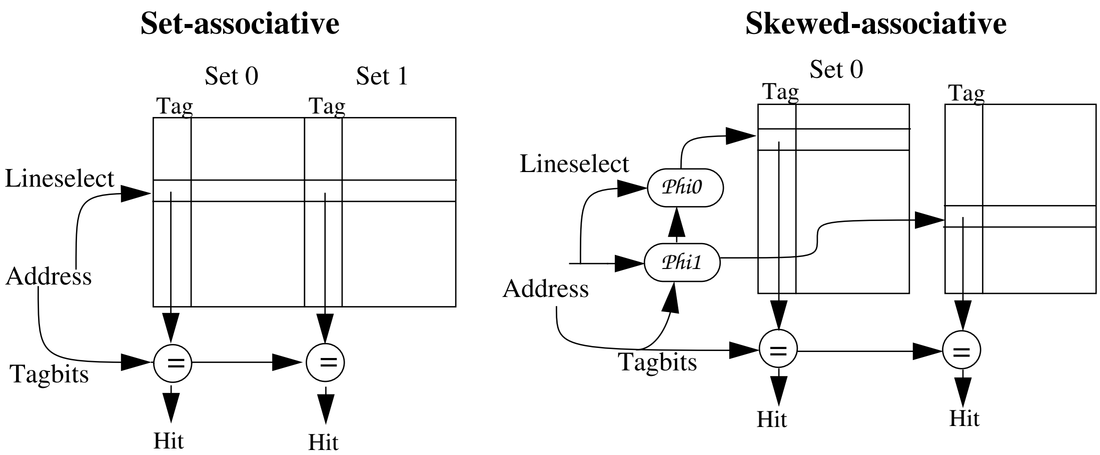


Figure 5.13 2-way set- and skewed-associative caches

and *Phi1*, are applied to, $A1_2$ and A2, to form new, different, line numbers in the different sets. Figure 5.14. shows how the address bits are divided and how the skewing functions can be implemented using xor-gates.

A simple class of skewing functions, which has been investigated with the aim of minimizing the delay overhead can be employed if the following criterion is met [Seznec]:

$$Phi0\,(address) \bullet Phi1\,(address) = 0 \qquad \text{(EQ 5.44)}$$

where • signifies bit-wise ‘and’. This criterion will ensure that bit ‘n’ in A2 and $A1_2$ will only be loaded with the input of *one* xor-gate each implying minimum effect on the cycle time of the structure

If the skewing functions employed have inverse functions it is possible to regenerate the original, physical address when the cache line is written back to memory; in this case the tag incorporated in the cache line is the same as in a conventional set-associative cache. If the original address cannot be regenerated, the tag field needs to be extended to contain both the A1 and A2 parts of the referencing address as for a fully associative cache.

To minimize the delay and the power consumption of the cache it is desirable to keep the number of tag-bits as low as possible, hence skewing functions which can be reversed should be chosen.

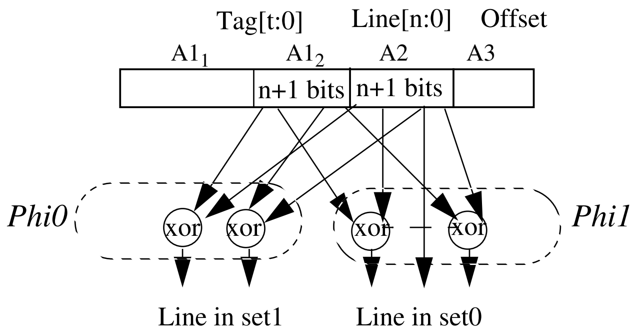


Figure 5.14 Different mapping functions in different sets

Skewed-associativity has an effect equivalent to doubling the ‘conventional’ degree of associativity, so that a 2-way skewed-associative cache performs as well as a 4-way set-associative cache [Seznec], for a small overhead in terms of timing. If only simple skewing functions are considered, the energy consumption of an N-way skewed-associative cache is estimated to be the same as the corresponding N-way set-associative configuration. However, as the hit rate is expected to be higher in a skewed-associative than in a set-associative cache, it is considered more energy efficient.

5.5.1 Choosing a set of skewing functions

The number of possible skewing functions for an N-way set-associative cache is high. Here the investigations will be limited to the use of skewing functions built using xor-gates as illustrated in Figure 5.14. This class of skewing functions is simple to handle as it is monadic, and it is therefore simple both to compute the line numbers in the sets in the skewed-associative cache and to regenerate the memory address if the cache line needs to be written back to memory.

Furthermore, to limit the load on each bit in the address paths, each bit in the line field (A2 in the description above) should be loaded with one xor-gate only; this will minimize the timing overhead. Work has shown that although some improvement in cache performance can be obtained by tuning the skewing function for a specific program, the improvement obtained from skewing is largely independent of the skewing function, for the class of skewing functions considered, over the range of benchmarks described in Chapter 4.

5.6 Replacement algorithms

A number of replacement algorithms exist for set-/fully-associative cache configurations. The most accepted ones are the Random and LRU[1] [Patt] algorithms, where the LRU algorithm in general produces the best results. A ‘random’ replacement policy is easy to implement and requires a minimum of extra hardware whereas a LRU algorithm requires information for each line regarding the least recently accessed set. This state is very small, one bit, when targeting a 2-way set-associative cache. For 4-way set-associative configurations the state can be incorporated into 4 bits [Thakker]. For higher degrees of associativity the complexity of the LRU algorithm increases rapidly and it is not feasible to use LRU for higher degrees of associativity. In the case of the 2-way set-associative cache, the timing overhead for manipulating this ‘state’-information is minimal.

For a skewed-associative cache the relation between the lines and the sets is not as simple as for the set-associative cache. One address might map to line ‘L_1’ in set 0 and to ‘L_2’ in set 1 while another address also maps to line ‘L_1’ in set 0 but maps to line ‘L_3’ in set 1. To implement a LRU-replacement algorithm it is therefore not enough to compare the access-pattern between ‘number of sets’-lines. Choosing between the lines selected by a given address is effectively as ‘bad’ as choosing randomly. For the two-way skewed-associative caches, however, a replacement policy which has many of the properties of the LRU-replacement algorithm has been proposed [Seznec]:

> “An extra bit is associated with each line in set 0. This extra bit is asserted when the requested word is in set 0 and de-asserted when the data is in set 1.”
>
> “On a miss, the extra bit of the line selected in set 0 is read: when this tag is 1, the missing line is written in set 1, otherwise the missing line is written in set 0.”

This replacement algorithm will be referred to as Pseudo-LRU. Note it has the same hardware requirement as the LRU algorithm in a set-associative configuration. As

1. Least Recently Used

Table 5.6 shows, the pseudo-LRU replacement algorithm yields a performance better than the Random replacement algorithm; but not as good as that given by true LRU.

However, the table shows how a 2-way *skewed*-associative cache with the pseudo-LRU replacement algorithm performs better than a 2-way *set*-associative with the true LRU replacement algorithm.

Table 5.6 Performance of replacement algorithms

Replacement Algorithm	Hit Rate in 4K-byte unified, 2-way *skewed*-associative cache with 32-byte cache lines						
	cacti	**dhry**	**espresso**	**fft**	**flex**	**hello**	**stcompiler**
Random	97.6	96.7	97.4	98.8	97.3	93.7	94.7
Pseudo-LRU	97.9	98.1	97.6	99.0	97.3	93.6	95.0
LRU[a]	98.1	98.3	97.8	99.1	97.5	94.4	95.3
	Hit Rate in 4K-byte unified, 2-way *set*-associative cache with 32-byte cache line						
LRU	97.6	96.7	97.2	98.6	96.1	92.6	94.4

a. Simulated by attaching a 32-bit timestamp to each line.

5.7 Cache timing

The effect of the cache configuration on the access- and cycle-time of a cache was investigated using the ‘cacti’ [Wilton]; a cache evaluation package developed by Digital Equipment Coporation[1]. This section will describe how the different cache parameters such as size, line size and associativity affect the timing of the cache.

Figure 5.15 shows how the cache cycle time *decreases* for increasing line size and how the cycle time of a cache increases for increasing cache size and increasing degree of associativity. This is in line with the relationship explained in section 5.1. The bit lines in the cache have relatively poor driving characteristics compared to the word lines (see

1. Cacti has also been ported to the HORN-architecture and is used in the investigations as an application benchmark.

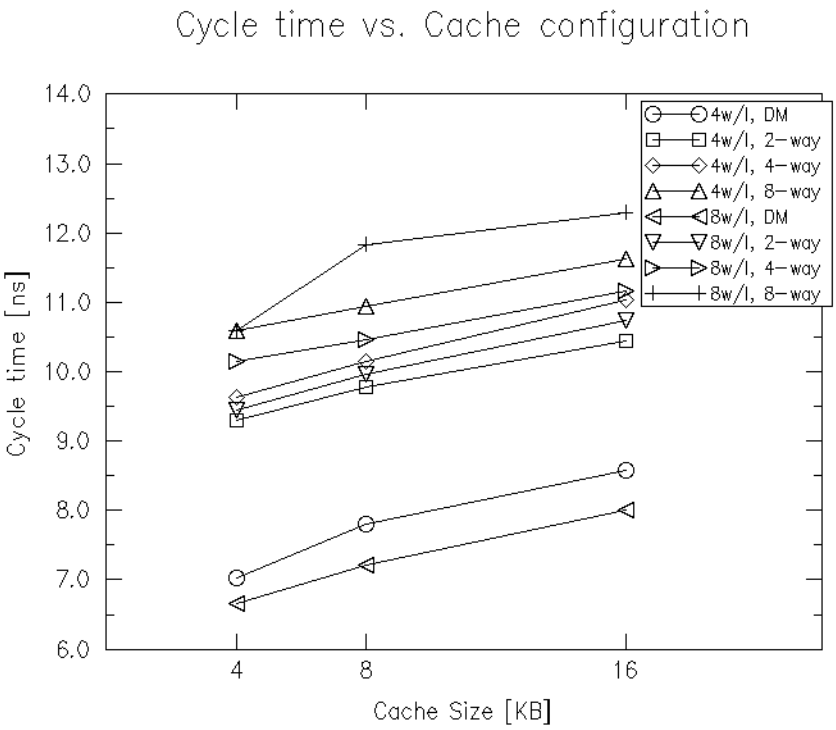


Figure 5.15 Cache cycle time versus cache size and organization

Figure 5.5 on page 81) and a reduction in the number of cache lines does therefore improve the cycle time of the cache.

Increasing the associativity is expected to increase the cycle time although the accesses to the individual sets becomes faster as the size of each set is reduced. Figure 5.16 shows how the cycle time of the cache increases for increasing associativity:

In a direct-mapped cache, the lookup in the data memory and the setup of the output multiplexer can be done in parallel with the tag comparisons. The requested word can therefore be at the output of the cache at the same time as the hit/miss-signal. For a set-associative cache this is not possible as the hit/miss signal from the tag comparisons is required before the output multiplexers can be set up correctly. This explains the very steep increase in cycle time going from a direct mapped cache to a 2-way set-associative configuration, see Figure 5.16. The increase in cycle time observed for higher degree of

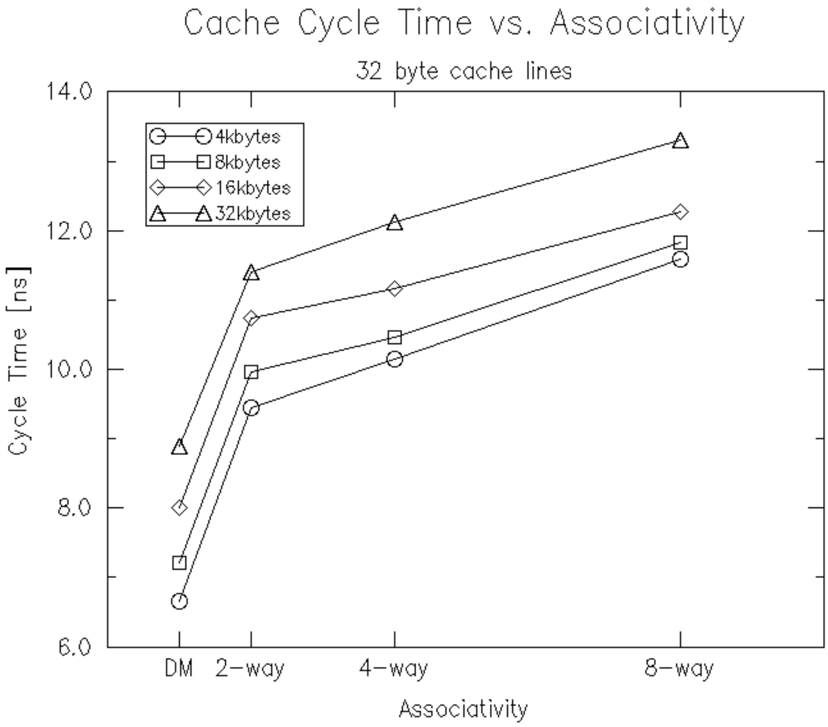


Figure 5.16 Cache cycle time versus associativity

associativity is partly due to extra internal routing and partly due to the increase in the complexity of the output multiplexer.

From Figure 5.15 and Figure 5.16, it can be seen that long cache lines have little effect on the cycle time for a constant cache size. In general, a small cache with low degree of associativity, will yield the shortest cycle time.

5.8 Block buffering

It has been suggested, [Hill], [Su], [Bunda], [Okada] that the introduction of a buffer on the output of the cache, as shown in Figure 5.17, will reduce the number of accesses to the energy consuming memory blocks. The requesting address will be checked against a 'Tag Buffer' holding the tag for the data in the Data buffer and hence determine whether the requested word is in the 'Data Buffer'. If the contents of the Tag Buffer matches the requesting address, the word will be fetched from the buffer and the rest of the cache will not be activated.

Effectively the Data Buffer forms a small fully-associative level-0 cache, but as the energy consumption of the cache scales with the dimensions of the cache there is a considerable energy-saving associated with the introduction of a small level-0 cache as the cost of fetching a word from the buffer is smaller than that associated with fetching a word from the Data Memory. It follows - within limits [Bunda] - that the longer the cacheline, and hence the longer the data buffer, the bigger the saving.

The R4300i architecture has a small, two-instruction, block buffer on the instruction cache in order to reduce the number of references to the cache itself thus reducing the energy consumption of the cache [R4300i].

The block buffer 'cache' should be of type 'write through' to the main cache to avoid the necessity to implement a coherency protocol, which might be complex and could decrease the performance. It is important that the Tag Buffer contains all information that would normally be in the tag of a fully associative cache, i.e all the bits of the address except the offset-bits, see Figure 5.17.

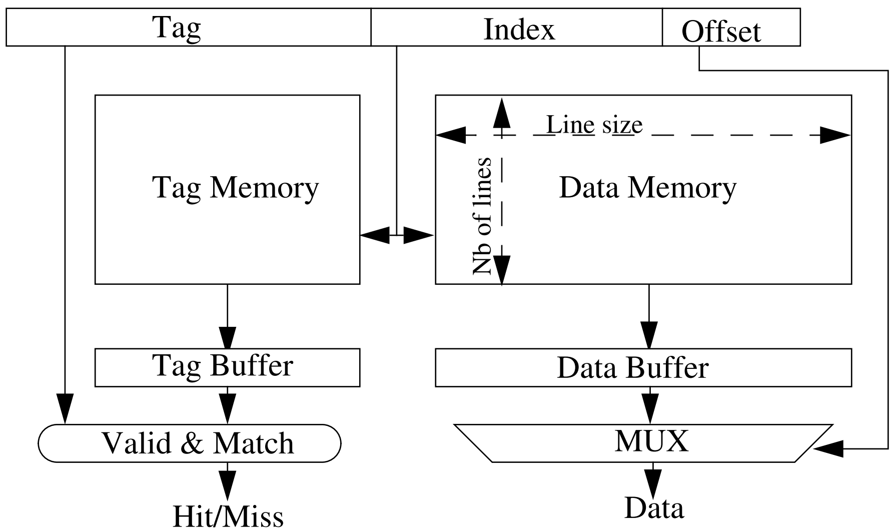


Figure 5.17 Block Buffering

Let ‘N1’ signify the number of accesses served by the memory blocks and ‘N2’ the number of accesses served by the Buffers. $E_{BufAccess}/E_{MemAccess}$ is the ratio of energy consumption for an access to the Buffers and an access to the Memory Blocks of the cache. The reduction in energy consumption can therefore be expressed as:

$$Reduction = 1.00 - \frac{N1 + \frac{E_{BufAccess}}{E_{MemAccess}} \times N2}{(N1 + N2)} \qquad \text{(EQ 5.45)}$$

Note that N1+N2 is greater than the total number of accesses as write operations, which ‘hit’ in the block buffer, will both count towards N1 and N2 due to the write through approach. The system is not in the same way a ‘read-through’ system in that a read from the cache only counts towards N1.

The precise value of $\frac{E_{BufAccess}}{E_{MemAccess}}$ is difficult to determine without implementing the architecture. However it should be clear that the ratio is much less than 1.0. A first order approximation is that the ratio scales with the number of lines in the caches. This is an optimistic assumption as the line size is an important factor in the expression for energy consumption, especially for caches with few lines, see Equation 5.15. However, if the block buffer is simply implemented as a latch-register there is almost no energy consumption associated with fetching a word for the block buffer. The rest of this thesis approximates the ratio with:

$$\frac{E_{BufAccess}}{E_{MemAccess}} = \frac{1}{NbOfSets \times NbOfLinesPerSet} \qquad \text{(EQ 5.46)}$$

Table 5.7 shows the reduction in accesses to the Data and Tag memory blocks and the corresponding reduction in energy consumption if block buffers are introduced in both instruction and data caches.

The table shows how the reduction in cache accesses and hence in energy consumption increase with the line size in the caches. Both the instruction and the data caches were 4K byte, 2-way set-associative. There is a significant reduction when increasing the line size from 4 words/16 bytes to 8 words/32 bytes while the reduction is smaller when the line size is increased further to 16 words/64 bytes. It should be observed that the reduction in traffic is greater in the instruction cache (IC) than in the data cache (DC). There are two results for memory access reduction for the data caches. The principal result indicates the percentage of read requests served by the buffer, while the result in parentheses indicates the percentage of all accesses served (read or write) which could be served only by the buffer. The results indicate that there is a high degree of spatial locality in the data as well as in the instructions, consequently there is a significant reduction in energy consumption if block buffers are added to a cache design.

Table 5.7 Effect of Block Buffering on cache traffic and energy consumption

		Line size: 16 bytes		**Line size: 32 bytes**		**Line size: 64 bytes**	
		Reduction in Data- and Tag Memory accesses [%]	**Reduction in energy consumpt. [%]**	**Reduction in Data- and Tag Memory accesses [%]**	**Reduction in energy consumpt. [%]**	**Reduction in Data- and Tag Memory accesses [%]**	**Reduction in energy consumpt. [%]**
cacti	DC	22.6 (40.6)	22.5	27.4 (50.4)	27.0	32.4 (57.3)	31.5
	IC	59.0	58.8	66.5	66.0	67.1	66.1
dhry	DC	14.7 (28.2)	14.6	19.3 (35.8)	19.0	22.3 (38.4)	21.7
	IC	64.1	63.9	73.6	73.0	77.0	75.8
espresso	DC	13.5 (23.7)	13.4	16.7 (28.5)	16.4	14.1 (23.4)	13.7
	IC	61.6	61.4	70.2	69.6	71.8	70.7
hello	DC	14.6 (29.4)	14.4	19.8 (40.4)	19.5	21.4 (42.2)	20.7
	IC	48.6	48.4	50.1	49.7	47.7	46.9
Average	DC	16.4	16.2	20.8	20.5	22.6	21.9
	IC	58.3	58.1	65.1	64.6	65.9	64.9

If an instruction-only block buffer was built into the (unified) cache in an ARM processor, (16-bytes, 4-words, cache lines,) mentioned in Chapter 2 the power consumption could be reduced considerably as follows:

Consider an average benchmark where the ratio between accesses to the data and instruction cache is 1:4 the reduction in power consumption is:

$$Reduction = 1 - \frac{\left(\frac{1}{5} + \frac{4}{5} \times (1 - AccessReduc_{Icache})\right) \times P_{UniCache} + P_{Proc}}{P_{Unicache} + P_{Proc}} \qquad \text{(EQ 5.47)}$$

$$Reduction = 1 - \frac{\left(\frac{1}{5} + \frac{4}{5} \times (1 - 0.583)\right) \times 432mW + 693mW}{432mW + 693mW} = 17.8\% \qquad \text{(EQ 5.48)}$$

Block buffering can also improve processor performance considerably [Su]. The cache(s) will often be on the critical path in the implementation of a pipelined architecture. The block buffer provides faster access to instructions and data than if the cache itself needs to be accessed. The rest of the pipeline can therefore be designed to match or exploit the cycle time of the block buffers and take a small penalty when the request needs to access the Data and Tag Memories.

As an integrated part of the cache structure the block buffer is expected to have a minimal effect on the cache cycle time: The cache cycle time increases proportionally with the number of lines in the cache [Wilton] and introducing the block buffer[1] might therefore increase the cycle time by as little as than 0.4% for a cache with 256 lines.

5.9 Fetch and Write Back buffers

The results presented in Table 5.7 showed that the number of references to the caches could be reduced considerably by the introduction of a block buffer. A significant proportion of the remaining cache accesses are related to fetching - and in the data cache writing back - cache lines. Lines are normally fetched and written back word-by-word i.e. words are written into the cache storage as they arrive from memory. Alternatively the

1. This effectively increases the number of lines in the cache by one. However it depends strongly on the implementation strategy

words arriving from memory can be collected in a ‘Fetch Buffer’ and only when all the words for a cache line have arrived will the contents of the buffer be written into the cache memory. Similarly, if a cache line is to be written back to memory, the cache line is fetched into a ‘Write Back Buffer’ from where the individual words are written back to memory. This reduces the number of accesses to the energy consuming cache further.

Figure 5.18 shows how these two buffers can be integrated into a cache with a block buffer. The numbers in parentheses explain the sequence of operations following a cache miss:

(1) Following a cache miss the victim line in the cache is - if dirty - copied into the Write Back Buffer.

(2) The words for the new cache line arrive from external memory at a rate of one word per cycle. They are temporarily stored in the Fetch Buffer.

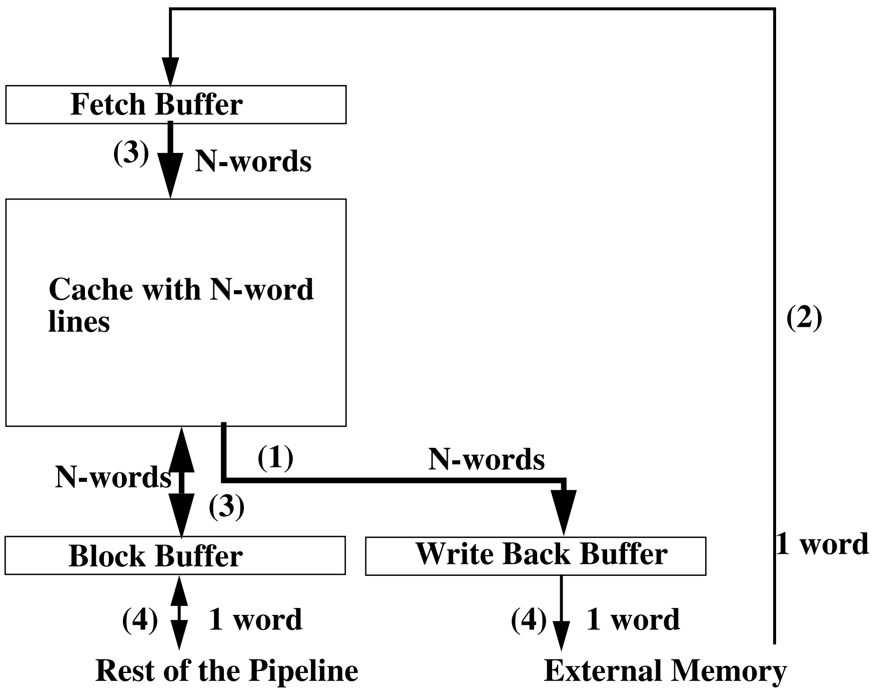


Figure 5.18 Cache with three block buffers

(3) When all N words have arrived from the external memory, the entire line is written into the cache and into the block buffer.

(4) The execution pipeline is served from the block buffer and the data Write Back Buffer is written back to memory simultaneously.

The Write Back Buffer should be only one ‘element’ deep and should block the rest of the pipeline if a cache miss occurs before it has written all the N-words it contains back to memory. Thus read-after-write hazards are avoided, and no detection circuit is required.

5.10 Gray-coding fetches/writebacks

Gray-coding [Kohavi] is a set of monadic encoding functions which map N numbers in such a way that Gray-code representation of ‘X’ and ‘(X+1) `modulo` N’ differs by exactly one bit. Table 5.8 shows an example of a Gray-encoding of the numbers from 0 to 7. Note also the single bit-transition between the representations of ‘7’ and ‘0’.

Table 5.8 Gray-coding

Decimal Representation	Binary Representation	Gray code Representation
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

For energy efficient designs this encoding can be used to minimize transitions on the address and data buses [Su]. It would, however, require compiler knowledge of the use of Gray coding within the processor. In this thesis only schemes which are invisible to the program execution, i.e. schemes which do not require re-compilation, are considered.

Gray-coding has therefore been employed only when fetching/writing-back data between the cache(s) and the main memory and only on the address bus. Words in the cache line will then be fetched/written back in a Gray code order rather than in the conventional sequence. This reduces the number of bit-transitions on the address bus by up to 43% for an 8-word/32-byte cache line. Note that the number of transitions on the address bus will be independent of choice of word to be fetched first; this is not the case if words are fetched sequentially. The reduction in bit-transitions on the address bus increases with the line size, even though the increase is very small for cache lines longer than 8 words per line, see Figure 5.19.

By counting the number of bit transitions on the address and data buses, a precise measure can be obtained of the value of fetching/writing back words from cache lines in a Gray-code order. The bit-transitions have been counted on a HORN-processor system with an 8K byte direct mapped, unified cache with 32-byte (8-word) cache lines. Table 5.9 shows the saving in bit-transitions due to fetching and writing back cache lines in Gray-code order instead of a conventional binary sequential order. Note that for all benchmarks there is a considerable reduction in the number of transitions on the address bus and that the

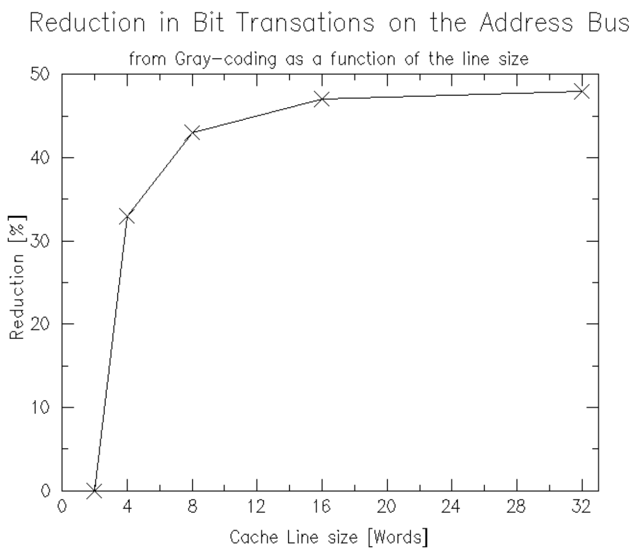


Figure 5.19 Reduction in bit-transitions on the address bus from Gray-coding

Table 5.9 Effect of Gray-coding in a 8K byte unified cache with 32-byte lines

Benchmark	No of line fetches and writebacks	Bit trans. on address bus		Bit trans. on data bus		Reduction [%]
		Sequential	Gray code	Sequential	Gray code	
cacti	301,974	5,804,045	3,922,835	28,530,786	28,578,966	5.6
dhry	17,970	347,041	239,239	1,652,856	1,601,969	7.9
espresso	199,134	4,525,583	3,190,439	15,558,612	14,793,959	10.5
fft	30,005	633,333	453,279	2,875,630	2,370,673	19.5
flex	580,473	15,278,341	10,863,805	44,159,598	42,696,315	9.9
hello	5,261	139,460	99,362	470,477	463,459	7.7
stcompiler	131,658	3,076,063	2,158,687	11,154,112	10,833,464	9.5
				Average Reduction:		10.1

number of transitions on the data bus is affected minimally by the Gray-coding of the address bus. The large number of transitions on the data bus masks the reductions in transitions on the address bus, thus reducing the overall effect of the Gray-coding. Comparing these results to those published elsewhere [Su], it can be observed that while the reduction obtained on the address bus is comparable, ~33%, the reduction on the data bus, < 3% see Table 5.9, is much less than the 12% reduction quoted in [Su]. The overall saving is therefore ‘only’ 10.1%. The difference is difficult to explain, but the high savings reported in [Su] might be a result of careful opcode and register allocation and data-layout.

Applying this result to the measurements[1] of the ARM processor (Chapter 2) for which 10.2% of the power consumption of the processor is consumed in the I/O-drivers and pads, suggests that the reduction in bit-transitions is equivalent to 1% of the power budget of an ARM3. This is not a significant reduction but it should be noted that this reduction in switching activity on the I/O interface is likely to migrate to the external memory system. It is not feasible to quantify this saving, however, due to lack of detailed power information on memory chips. To quantify the saving in the memory a break-down of the

1. PLA structures omitted

total power consumption is required indicating the current drawn by the I/O drivers. This information is not available in the memory chips studied [Hitachi].

5.11 Selective writeback

The write back of dirty cache lines represents a significant proportion of the I/O traffic as shown in Table 5.10. However, not all words in a cache line which are written back to memory are altered or ‘dirty’. Therefore, if a ‘dirty’ bit is allocated for each word in a cache line instead of one for the whole line, the number of memory accesses can be reduced. Table 5.11 shows the distribution of ‘dirty’ words in the cache lines before they

Table 5.10 Writeback proportion of total I/O[a]

Benchmark	$\frac{Writeback}{TotalIO} \times 100\%$
cacti	3.68
dhry	4.16
espresso	10.5
fft	28.6
flex	21.1
hello	21.2
stcompiler	13.9
Average	**14.7**

a. Cache parameters: 8Kb Unified, 32-byte/line, Direct Mapped

were written back to memory following a miss. It shows that 94% of all lines to be written back to memory contain 7 or 8 dirty words, indicating that most words in ‘dirty’ lines need to be written back.

Let ‘n’ denote the number of dirty words in a cache line. The total reduction in writeback due to selective writeback can be calculated as:

$$Reduction = \sum_{n=1}^{8} (8-n) \times P(n) \qquad \text{(EQ 5.49)}$$

Table 5.11 Frequency of 'dirty' words per cache line

Benchmark	Number of dirty words per line[a]							
	1	2	3	4	5	6	7	8
cacti	0.4	0.2	0.2	13.4	0.1	0.1	0.1	85.4
dhry	6.2	4.9	2.5	7.4	1.5	1.2	1.9	74.4
espresso	0.2	0.1	0.1	0.6	0.2	0.2	0.8	97.9
fft	0.5	0.4	0.1	0.6	0.1	0.1	16.1	82.2
flex	0.1	0.1	0.1	0.4	0.1	0.1	0.2	98.9
hello	5.9	1.8	1.8	5.9	0.5	0.9	1.4	82.0
stcompiler	0.3	0.2	0.3	0.4	0.1	0.2	0.3	98.1
Average	1.9	1.1	0.7	4.1	0.4	0.4	5.9	88.4

a. In a 8K byte, direct mapped data cache with 32-byte (8-words) cache lines

i.e only 1.7%, based on the 'average' numbers in Table 5.11.

Adding the extra dirty bits, increases the size of the lines in the cache by:

$$Increase = \frac{WordsPerLine - 1}{32 \times WordsPerLine} \times 100\% = \left(\frac{1}{32} - \frac{1}{ls}\right) \times 100\% \qquad \text{(EQ 5.50)}$$

or 2.7% for a 32-byte cache line (8 words). Given that the energy consumption of the cache is dominated by the line size as shown in section 5.1, the energy consumption of a cache request increases. Given that the cache consumes approximately 50% of the power in the processor, see Chapter 2, and the I/O only 11%, the reduction in I/O traffic is not sufficient for the scheme to be energy efficient. The energy efficiency of a conventional 'one-dirty-bit-per-line' policy is better than the selective writeback scheme proposed above.

5.12 Summary

This chapter has shown where, within a RAM block, energy is consumed. The energy consumption in conventional RAM blocks, such as those designed by a RAM-compiler, is dominated by the sense amplifiers. The sense amplifiers not only consume significant energy when active, they also have a static power dissipation which cannot be neglected.

A sense amplifier circuit, proposed by [Burd2] has been introduced and analysed, it has no static power dissipation and a small dynamic energy consumption. The rest of the simulations therefore assume that the static power consumption could be eliminated or reduced to a level which could be ignored. Section 5.1.2 described how the energy consumption of both the precharge circuit, RAM-storage and the sense amplifiers could be reduced considerably by pre-charging the bit lines in the storage to an intermediate voltage only.

Based on these observations a number of cache organizations have been investigated and expressions for the energy consumption of each has been derived. For all organizations examined, the line size in the cache data memory is the major factor in the energy expressions. However, as the address space increases, so does the size and importance of the energy consumption of the tag-storage and hence the significance of the cache organization. Sections 5.4.1 and 5.4.2 presented two organizations, sectored caching and CAT-caching, whose energy consumptions are less sensitive to the number of bits in the address spaces.

Timing simulations were presented in section 5.7. These showed that the cache cycle time increases with size and associativity. The conclusion is therefore to incorporate a cache which is as small as possible to yield the performance required with a low degree of associativity. A way of improving the performance of a set-associative cache without affecting the energy consumption and the timing, skewed-associativity, was presented in section 5.5.

Gray-coding external memory accesses and the use of selective writeback were discussed in sections 5.10 and 5.11 as ways of reducing bit-transitions on the external buses. Gray-coding, was found to have a small beneficial effect on the number of transitions on external buses; selective writeback reduces the amount of I/O traffic slightly but increases the size of the Data Memory, and hence its power consumption, due to the extra dirty bits.

The most efficient way of reducing the cache energy consumption was by the introduction of block buffering, as described in section 5.8. This proposed the fetching of the 'current' cache line into a buffer, effectively a level-0 cache, thus reducing the number of accesses to the Tag and Data memories themselves, (see Figure 5.17 on page 101). This architecture is expected to reduce the power consumption of the caches by between 16% and 65% dependent on the exact cache configurations; this is equivalent to between 7% and 19% of the total power dissipation if incorporated into an ARM-processor [OMIMAP]. This buffer architecture will be explored further in Chapter 8 and Chapter 9.

Chapter 6 Dual instruction branch

The HORN architecture specifies a two-instruction control-transfer structure, see Chapter 3. It comprises a go-class instruction, specifying the target for the control transfer, and a leap-class instruction, possibly specifying a condition and a leap-shadow; the leap shadow indicates the place for the control transfer to take place, see Figure 6.1.

```
                 .....
                 go target      Condition
                 .....
                 .....
                 leapZ Rx,shadow
                 .....
leap shadow {    .....          PC will be overwritten after
                                this instruction if condition
                                is met
         target: ......
```

Figure 6.1 Go-leap structure

This chapter will examine how such a control transfer instruction (CTI) architecture affects the execution time for a benchmark and the energy efficiency, EE, of the processor.

All the investigations and the results described in this chapter form part of the author's research.

The compilers, which have been available throughout the project, have never generated code which specified any leap-shadow. This work has consequently not investigated the impact of a variable size leap-shadow on performance and energy efficiency.

6.1 Improving hit-rate through dual instruction branches

The total number of instructions in a program with this type of CTI-structure is greater than in conventional CTI architectures, such as the ones found in the MIPS and SPARC architectures [Farquhar][Weaver]. This increase in 'number of instructions to be executed' needs to yield a corresponding reduction in CPI for the structure to be performance efficient.

For example, the dhrystone benchmark executes 686,646 instructions including 98,763 two-instruction CTIs. However, if a conventional branch architecture had been employed there would only have been

$$686,646 - 98,763 = 587,883$$

instructions to execute. In other words; the two-instruction CTI architecture increases the number of instructions to be executed by 14%.

For this to be performance efficient, the execution time needs to be reduced by 14% through improvements in instruction cache hit-rate due to prefetching of the branch targets into the instruction cache, and reduced or eliminated branch penalties.

Due to the increased cache and I/O traffic and hence increased energy consumption in both the cache and the I/O systems, the reduction in execution time needs to be even more significant for the architecture to be energy efficient.

6.1.1 Effect on effective hit-rate

One of the underlying ideas behind the two-part CTI is that a go-instruction will have ensured that the target for a branch is already in the instruction cache when the branch is taken after the leap shadow. Compared with a conventional, single-instruction branch, the go-instruction ensures that the instruction stream achieves a higher hit-rate or, in the worst case, a reduced cache-miss penalty after branches. Table 6.1 shows the effective hit-rate as seen by the instruction stream for a number of benchmarks with a number of cache configurations. The first column of each cache size denoted 'No-prefetch' shows the hit-rate in the instruction cache if the effect of the 'go'-instruction is ignored i.e. no prefetch

Table 6.1 Effect of prefetching on hit-rate in instruction cache

Benchmark	1K bytes 32 bytes/line Direct mapped Eff. Hit Rate [%]		2K bytes 32 bytes/line Direct mapped Eff. Hit Rate [%]		4K bytes 32 bytes/line Direct mapped Eff. Hit Rate [%]		8K bytes 32 bytes/line Direct mapped Eff. Hit Rate [%]	
	No prefetch	Prefetch	No prefetch	Prefetch	No prefetch	Prefetch	No prefetch	Prefetch
cacti	91.9	95.2	93.9	96.3	95.2	97.0	98.3	98.8
dhrystone	91.4	96.4	93.2	97.2	94.0	97.3	98.4	99.3
espresso	95.4	98.1	96.6	96.6	97.7	99.0	98.9	99.5
flex	91.4	96.3	93.7	97.6	97.8	99.2	99.2	99.7
hello	85.4	95.0	88.7	95.8	90.7	96.7	95.6	98.2
stcompiler	90.2	97.0	92.9	96.3	93.9	98.3	95.7	99.0

is initiated. It is an approximation to the hit-rate that could be expected with a branch architecture like that in the SPARC architecture [Weaver].

The second column shows the effective hit-rate that can be expected in an implementation of the HORN architecture. It is assumed that the potential prefetch initiated by the go-instruction is transparent to the rest of the program execution and that the prefetch will have completed before the branch is taken.

From a hit-rate perspective the table shows that there is a significant advantage in introducing the two-part CTI especially for small caches (less than 8K bytes). For larger caches the gain is reduced. Furthermore, there is a clear advantage for small benchmarks, such as hello and dhrystone where the spatial locality is low and where compulsory misses [Patt] dominate.

6.1.2 Performance measurements

The performance of the two-part CTI has been assessed through a large number of simulations of the all the benchmarks in the suite, see Chapter 4. This thesis will only report on the results from two of the benchmarks, dhrystone and espresso. The dhrystone benchmark has been chosen because it clearly shows how the value of the two-part CTI is reduced as the cache size increases; while espresso has been chosen because the results from this benchmark are typical for the rest of the benchmark suite. Table 6.2 and Table 6.3 show two sets of figures for each benchmark:

1. A column denoted ‘no-Prefetch’. This set of simulations have ignored the go-class instruction completely and have counted them neither towards the number of instructions nor towards the execution time. This is an approximation to the execution time on a system with a conventional branch architecture as used in MIPS and SPARC.

2. The column denoted ‘Prefetch’ shows the results of simulations including the go-instructions, which count both towards the total number of instructions executed and the execution time. It is assumed that a go-instruction will have been placed early enough in the program execution that it will have prefetched the target before the branch is taken thus eliminating stalls in the instruction flow. This assumption is clearly very optimistic, especially for the longer memory latency. Furthermore the column shows, in parentheses, the percentage reduction in cycle count compared with the number in the ‘no-Prefetch’ column.

Simulations have been carried out for two memory latencies, 5 and 10 cycles. These approximate to the latencies that can be expected for systems built with either (fast) static RAM or (slower) dynamic RAM. The memory model assumes that memory banks are interleaved so that, after the initial latency of 5 or 10 cycles, the remaining words in the cache line will be filled at a rate of one word per cycle.

The results show that longer cache lines generally perform better and that prefetching reduces the execution time significantly, by more than 10% for small caches (less than 8K

Table 6.2 Execution time[ab], dhrystone

Inst. cache size[c] [bytes]	Memory latency = 5 cycles				Memory latency = 10 cycles			
	Line size: 16 bytes		Line size: 32 bytes		Line size: 16 bytes		Line size: 32 bytes	
	No prefetch	Prefetch	No prefetch	Prefetch	No prefetch	Prefetch	No prefetch	Prefetch
1K	1.165	1.014 (13%)	1.192	0.982 (18%)	1.525	1.219 (20%)	1.443	1.106 (23%)
2K	1.057	0.976 (8%)	1.062	0.914 (14%)	1.350	1.157 (14%)	1.259	1.009 (20%)
4K	1.017	0.965 (5%)	1.009	0.905 (10%)	1.285	1.138 (11%)	1.185	0.995 (16%)
8K	0.695	0.749 (-8%)	0.699	0.738 (-6%)	0.761	0.788 (-4%)	0.745	0.758 (-2%)

a. cycles divided by 1,000,000

b. 100% hit-rate in the data cache is assumed.

c. Only direct mapped caches have been examined.

bytes). For the largest configurations, the cycle count increases due to the increased number of instructions in the 'prefetch'-versions and the fact that prefetching does not have any significant effect on the hit-rate for those configurations, see section 6.1.1.

The same set of simulations has been carried out on a significantly larger benchmark, espresso. The results are shown in Table 6.3.

Table 6.3 Execution time[ab], espresso

Inst. cache size[c] [bytes]	Memory Latency = 5 cycles				Memory Latency = 10 cycles			
	Line size: 16 bytes		Line size: 32 bytes		Line size: 16 bytes		Line size: 32 bytes	
	no-Prefetch	Prefetch	no-Prefetch	Prefetch	no-Prefetch	Prefetch	no-Prefetch	Prefetch
1K	6.143	6.076 (1%)	5.959	5.669 (5%)	7.581	6.979 (8%)	6.842	6.101 (11%)
2K	5.525	5.703 (-3%)	5.373	5.385 (-0.2%)	6.576	6.372 (3%)	6.011	5.699 (5%)
4K	4.937	5.352 (-8%)	4.862	5.139 (-6%)	5.622	5.802 (-3%)	5.287	5.351 (-1%)
8K	4.376	4.988 (-14%)	4.345	4.881 (-12%)	4.710	5.211 (-11%)	4.554	4.985 (-9%)

a. cycles divided by 1,000,000

b. 100% hit-rate in the data cache is assumed.

c. Only direct mapped caches have been examined.

For this benchmark only the smallest (1K and 2K bytes) cache configurations benefit from prefetching, and the gain is only significant for the long memory latency. Indeed, prefetching is disadvantageous for the larger cache configurations. This is due to espresso's access pattern which yielded hit-rates of over 95% even for a 2K byte cache without prefetching; thus prefetching can not improve the hit-rate very much.

The 'hello.world' benchmark produces results similar to those for 'dhrystone', while 'cacti', 'flex' and 'stcompiler' produce results similar to those reported for 'espresso'.

The results confirm what might have been expected: as the cache size and thereby the hit-rate increase, the gain from the go-type instructions is reduced. The exact break-even configuration is a function of the benchmark.

Furthermore, the 'go'-instruction can clearly not migrate further up the program than the previous CTI-structure. From the simulation statistics it can seen that the average distance between CTI-structures is less than 5 instructions, see Table 6.4. Thus go- and the leap-

Table 6.4 Average distance between CTIs

Benchmark	Distance [instructions]
cacti	4.60
dhrystone	4.96
espresso	3.73
flex	4.59
hello	4.59
stcompiler	4.28
Average	**4.46**

class instructions can not, on average, be separated by more than 4.46 instructions[1]: not enough to fetch a full cache line, even for short cache lines (16 bytes = 4 words) and short memory latency (5 cycles). The assumption that the compiler can migrate the go-

1. An unbroken sequence of instructions is thus composed of 4.46 'normal' instructions plus two instructions related to the CTI, i.e a sequence of 6.46 instructions

```
               go label
               ....
        label: .....
               .....
               .....
               leapZ        } loopbody
               .....
Leap shadow {  .....
               .....
               .....
```

Figure 6.2 Example of go-instruction migrating outside loopbody

instructions so far up the instruction stream that prefetching will always have completed, is therefore very optimistic.

Future releases of the HORN-compiler are not expected to need to plant a go-instruction within the same basic block as the leap instruction. In case of simple loops, it is expected that a the compiler will be able to migrate some go-instructions outside loop-bodies as shown in Figure 6.2 and as explained in Chapter 3. This will reduce the instruction overhead due to the scheme as the go-instruction will be executed only once, rather than once per iteration.

An approximation to this, which ignores a go-class instruction if the previous go-class instruction prefetched the same address, has been implemented and examined. Eliminating ‘unnecessary’ go-instructions reduces the execution time of the benchmarks as well as reducing the energy consumption in the cache due to fewer references. This reduces the completion time for the ‘prefetch’ configurations shown in Table 6.2 and Table 6.3, but not enough to yield an improvement in performance for the largest, 8K byte, instruction caches.

The results presented here are therefore believed to show a correct trend although the exact values might change as the compiler technology improves. The two-instruction CTI is therefore expected to improve the performance for small caches, but the advantage is expected to decrease with increasing cache size/hit-rate.

6.1.3 Energy efficiency

The two-instruction CTI will cause more cache traffic and hence a higher energy consumption than a conventional branch architecture;

- Cache lines may be fetched without being required.
- The increased number of instructions will cause more cache accesses.

Based on the power consumption figures from Chapter 2, the energy efficiency, (EE), of different instruction cache configurations has been calculated[1]:

(EQ 6.1)

$$EE = \frac{1}{\left(miss \times \frac{linesize}{4} \times E_{Cache,Write} + request \times E_{Cache,Read} + miss \times \frac{linesize}{4} \times E_{Mem} + E_{ProcCore}\right) \times cycles}$$

$E_{Cache,Read}$ and $E_{Cache,Write}$ scale with the cache parameters, as shown in Chapter 5.

Table 6.5 shows the EE for different instruction cache configurations for the ‘dhrystone’ benchmark. The ‘Prefetch’-column shows, in addition to the EE value, the improvement over the ‘no-Prefetch’-results.

The cache architecture used for these simulations was a simple direct-mapped cache without any of the energy reducing features to be proposed in Chapter 8. The table shows clearly that the two-instruction CTI-structure has a positive effect on EE for small caches, but the EE for larger caches (8K and 16K bytes) is lower for the two-instruction CTI than for the conventional single-instruction CTI.

This decrease in EE is partly explained by the increased number of instructions for the two-instruction CTI and partly by the reduced effect prefetching has on the cache

1. The cycle time of the processor is assumed to be constant and thus independent of the instruction cache configuration. The cycle time is therefore left out of the calculations of EE.

Table 6.5 EE for different cache- and memory configurations[a], dhrystone

Inst. cache size[b] [bytes]	Memory Latency = 5 cycles				Memory Latency = 10 cycles			
	Line size: 16 bytes		Line size: 32 bytes		Line size: 16 bytes		Line size: 32 bytes	
	no-Prefetch	Prefetch	no-Prefetch	Prefetch	no-Prefetch	Prefetch	no-Prefetch	Prefetch
1K	0.320	0.375 (17%)	0.246	0.283 (15%)	0.139	0.179 (29%)	0.115	0.140 (22%)
2K	0.396	0.426 (8%)	0.324	0.365 (13%)	0.182	0.215 (18%)	0.159	0.192 (21%)
4K	0.401	0.409 (2%)	0.344	0.365 (6%)	0.192	0.216 (13%)	0.177	0.201 (14%)
8K	1.109	0.864 (-22%)	0.990	0.807 (-18%)	0.767	0.656 (-14%)	0.687	0.605 (-12%)
16K	1.202	0.768 (-36%)	1.180	0.772 (-35%)	1.066	0.706 (-34%)	1.051	0.715 (-32%)

a. 100% hit-rate in data cache is assumed

b. Direct mapped

performance of a system with a large instruction cache as it does not improve the hit-rate significantly.

Table 6.6 shows the results for the ‘espresso’ benchmark. In contrast to the ‘dhrystone’ benchmark the EE does not improve, even for small cache configurations, with the new CTI-structure. The two-instruction CTI improves the EE measure only for the smallest cache configurations with long memory latency (10 cycles).

For all the benchmarks examined it is characteristic that the highest EE *values* are found for cache sizes larger than the limit where the two-instruction CTI is energy efficient. This section therefore concludes that if the early specification of the branch target is only used to increase the performance of the instruction cache, the two-instruction CTI improves the energy efficiency of systems with small caches, but is in general not energy efficient.

Table 6.6 EE for different cache- and memory configurations[a], espresso

Inst. cache size[b] [bytes]	Memory Latency = 5 cycles				Memory Latency = 10 cycles			
	Line size: 16 bytes		Line size: 32 bytes		Line size: 16 bytes		Line size: 32 bytes	
	no-Prefetch	Prefetch	no-Prefetch	Prefetch	no-Prefetch	Prefetch	no-Prefetch	Prefetch
1K	0.135	0.132 (-2%)	0.124	0.125 (0.8%)	0.065	0.070 (8%)	0.064	0.070 (9%)
2K	0.175	0.158 (-10%)	0.165	0.154 (-7%)	0.093	0.092 (-1%)	0.092	0.094 (2%)
4K	0.223	0.177 (-21%)	0.209	0.173 (-17%)	0.135	0.118 (-13%)	0.131	0.118 (-10%)
8K	0.269	0.179 (-33%)	0.258	0.179 (-31%)	0.200	0.146 (-27%)	0.195	0.147 (-25%)
16K	0.228	0.133 (-42%)	0.225	0.134 (-40%)	0.195	0.121 (-38%)	0.197	0.123 (-38%)

a. 100% hit-rate in data cache is assumed

b. Direct mapped.

6.2 Reduction of cache miss penalty through two-instruction CTI

In addition to prefetching the target of a branch into the instruction cache, as shown above, the go-instruction can be used to reduce or eliminate the miss-prediction penalty through speculative fetching of instructions into the first stage(s) of a shadow pipeline [Hill], see Figure 6.3.

Once the potential target for a CTI instruction is known, an instruction fetch engine can - speculatively - fetch instructions from the target specified by the ‘go’-instruction. This may be performed in parallel with the fetching of the instructions between the ‘go’-class instruction and the ‘leap’-class instruction, see Figure 6.1.

Speculative fetching can not proceed far since the prefetched instructions must not affect the state of the processor in any way. Consequently, only early stages in a pipeline such as ‘Instruction Fetch’ and ‘Decode’, (see [Patt]) can be completed before this ‘alternative’ instruction flow must stall and wait for the branch condition to be resolved.

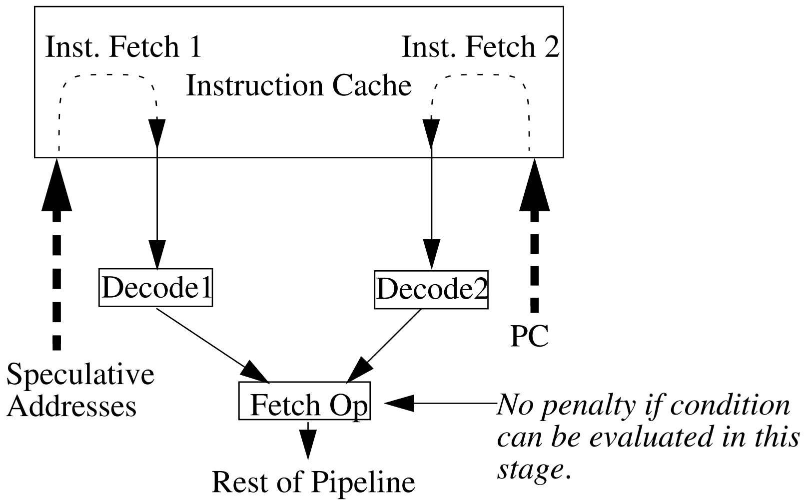


Figure 6.3 Doubling the early pipeline stages might eliminate branch penalty

Once the condition has been resolved there is no branch penalty associated with branching assuming the condition can be resolved in the 'Fetch op(erand)'-stage, see Figure 6.3[1]. This has the effect of replicating the early stages in the pipeline.

This technique for eliminating or reducing[2] the branch penalty clearly increases the energy consumption. The instruction cache needs to be dual-ported to be able to serve the two instruction steams. However, it is clear that the utilization of the second port will be relatively low. Section 5.1.3 has shown that the energy consumption per access increases with the number of ports in a RAM block.

Table 6.4 showed that the average distance between CTI structures is 4.46 instructions; i.e the average basic block is 6.46 instructions (4.46 'normal' instructions plus two CTI), while it is only 5.46 if a single-instruction branch is used. Given the pipeline structure of Figure 6.3, only two instructions can be fetched from the instruction cache before the

1. This assumes that the potential target of the CTI is in the instruction cache, which the previous section has shown is not always the case. If the target is *not* in the cache, some penalty is faced. However, for the rest of this section, it will be assumed that the target for the branch is already in the instruction cache.
2. In the case where the target is not in the instruction cache it is not expected that two instructions can be fetched before the branch is taken. In that case the branch penalty is 'only' reduced.

speculative prefetching must stop. The second port on the instruction cache will therefore only be utilized:

$$Utilization = \frac{2}{6.46} \times 100\% = 31.0\% \text{ of cycles} \qquad \text{(EQ 6.2)}$$

The number of cache requests, and thereby the energy consumption in the cache thus increases by 31%.

The issue of cache access is closely linked to the issue of variable-size instructions and instructions, which might straddle cache line boundaries, see Chapter 3. Chapter 8 proposes cache structures which solve this problem. Replicating one of these structures might also eliminate the need to make the instruction cache dual-ported and thus increase the energy consumption in the cache. The Dual Cache Line, DCL, architecture, presented in section 8.3.2, would be particularly suitable for this purpose. However, it might be necessary to enhance it to contain three or even four cache lines to enable it to serve its original purpose (composing instructions which straddle cache lines). This involves a very complicated structure, see Figure 6.4, which requires four tag compares per cycle, and which therefore will be relatively energy consuming.

Other techniques for reducing/minimizing the branch penalty are therefore required. [Patt] has shown that a single branch delay slot can be filled in approximately 50% of all CTIs. In those cases the branch penalty is reduced to 0 for correctly predicted branches; i.e. prefetching will not have any effect.

In this calculation the number of cycles to execute a benchmark on the HORN architecture, with the ‘go’-instruction speculatively prefetching from the potential target,

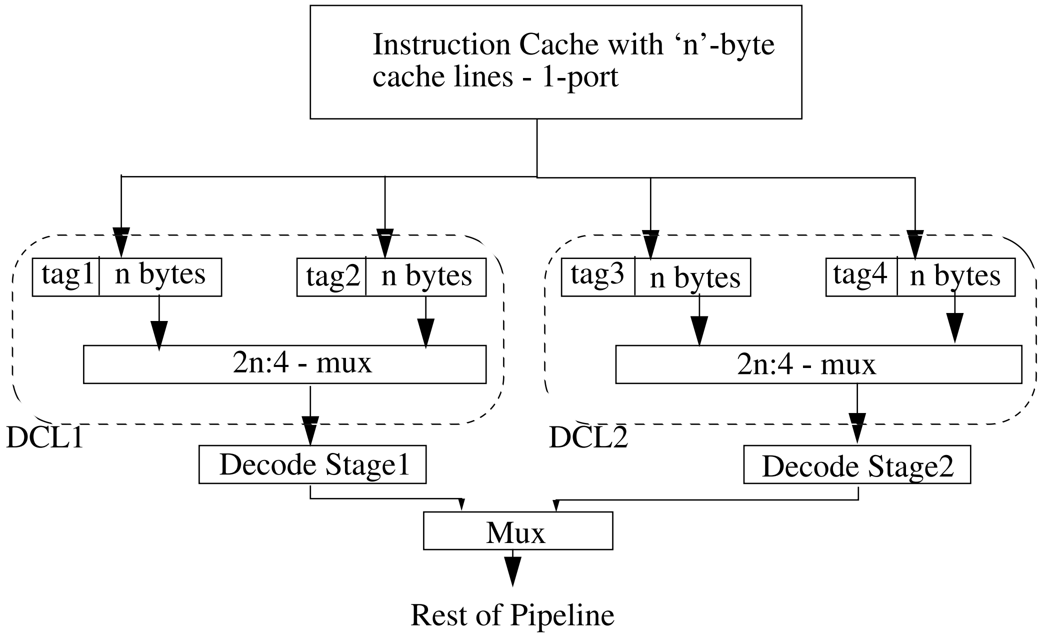


Figure 6.4 Replication of instruction alignment structure

has been normalized to 1.0. Asssuming a 100% prediction accuracy, the execution of the same block with a single-instruction branch is:

$$T_{100\%\ \text{Accuracy}} = \frac{T_{SingleInstBranch}}{T_{HORN}} = \frac{5.46}{6.46} = 0.85 \qquad \text{(EQ 6.3)}$$

However, with a branch frequency as shown in Table 6.4, a branch prediction accuracy (*pa*) of 50%, a branch penalty of one cycle and a misprediction penalty of a further one cycle; the relative execution time is:

$$T_{50\%\ \text{Accuracy}} = T_{100\%\ \text{Accuracy}} + pa \times \frac{penalty_{taken}}{BranchFrequency} + (1-pa) \times \frac{penalty_{Nottaken}}{BranchFrequency} \qquad \text{(EQ 6.4)}$$

$$= 0.85 + 0.5 \times \frac{1}{5.46} + 0.5 \times \frac{2}{5.46} = 1.12$$

i.e. a 12% increase in execution time compared to the HORN architecture.

For the following evaluation the energy consumption of the instruction cache is normalized to 1.0 for the case where there is a 100% prediction accuracy of all branches, i.e only the instructions which are going to be executed will be fetched. Furthermore, all fetches carry the same energy cost. Relative to this ideal scheme, the prefetch scheme will consume the equivalent two extra cache accesses per branch:

$$Energy_{Relative}(Prefetch) = 1 + \frac{2}{6.46} = 1.31 \qquad \text{(EQ 6.5)}$$

while a scheme relying upon branch prediction will consume the equivalent to one extra cache access per branch which is predicted correctly and two extra cache accesses per miss predicted branch:

$$Energy_{Relative}(Predict) = 0.85 + 0.5 \times \frac{1}{5.46} + 0.5 \times \frac{2}{5.46} = 1.12 \qquad \text{(EQ 6.6)}$$

Based on these results the energy efficiency, EE, for the two schemes can be calculated,

Table 6.7 EE for prefetch and branch-prediction schemes - accuracy: 50%

	Prefetch	**Branch Prediction**
Energy	1.31	1.12
Delay	1.00	1.12
$EE = \frac{1}{Energy \times Delay}$	**0.76**	**0.80**

resulting in Table 6.7. The bottom line in the table shows that the energy efficiency of the prefetch scheme is not as good as that of the more conventional scheme relying upon branch prediction. However, the difference is small.

Various branch prediction schemes can be employed to improve the 50% prediction accuracy assumed above. A number of branch prediction schemes are described in [Patt] of which the simplest scheme, which simply assumes that branches will be taken, performs well for a minimum hardware cost.

Simulations show that prediction accuracies of more than 77% can be obtained with simple schemes such as ‘predict taken’, see Table 6.8.

Table 6.8 Prediction accuracy for the 'predict taken' model

Benchmark	Prediction Accuracy [%]
cacti	87.4
dhrystone	77.7
espresso	60.9
fft	79.4
flex	79.4
hello	77.9
stcompiler	81.2
Average	**77.7**

Recalculating the numbers from Table 6.7 with the prediction accuracy from Table 6.8 gives the results in Table 6.9. The gap between the prefetch model and the more conventional branch-prediction model has widened as a result of the improved performance and reduced energy consumption of the branch-prediction model. The performance of the single-instruction branch prediction scheme is still less than the performance of the prefetch scheme.

Table 6.9 EE for prefetch and branch prediction schemes - accuracy: 77.7%

	Prefetch	Branch Prediction
Energy	1.31	1.08
Delay	1.00	1.08
$EE = \frac{1}{Energy \times Delay}$	**0.76**	**0.86**

6.3 Alternative branch and loop architectures

According to the results in section 6.2 an energy-efficient architecture would specify conventional [Farquhar][Weaver], single-instruction CTIs. By letting the branch instruction itself contain information about the number of the branch delay slots [Mahon], some increase in code compactness could be obtained - over the MIPS and SPARC architectures - as not all branch delay slots can be filled [Patt]. This should lead to a higher hit-rate in the instruction cache as the cache would contain more 'useful' code.

Furthermore, an energy-efficient instruction set architecture should contain ‘touch’ instructions, which the compiler can use to prefetch instructions and data into the appropriate caches. However, there should be no link between the touch and the branch instructions as is the case in the HORN architecture between the go- and the leap-class instructions. Touch-instructions could be placed early in the instruction stream, even earlier than the equivalent go-instruction could have been placed, thereby increasing the chance of having pre-fetched the target for the branch into the instruction cache before the branch is taken, see Figure 6.5. Comparing the ‘HORN code’ and the ‘Energy-efficient code’ sequences it is clear that the ‘Energy-efficient code’ contains fewer instructions. The size of the codes is expected to be the same[1]. Consequently the performance of the instruction cache should be similar, and the overall performance of the ‘Energy-efficient code’ would be better than that of the HORN-code. The performance of the processor might even be better (lower CPI) in the ‘Energy-efficient code’ as the ‘touchI’-instruction has been migrated further up the code than the corresponding ‘go’ instruction in the HORN-code, thereby increasing the chance that the code in ‘procA’ is present in the

‘C’-code	**HORN-code**	**Energy-efficient code**
- - - - - -	- - - - - -	- - - - - -
- - - - - -	go ‘L1’	touchI @procA
- - - - - -	leapnZ Ra,0	- - - - - -
if (a == 0) exit(1);	go ‘exit’	bnZ Ra,L1
- - - - - -	leaplink 0	call ‘exit’
procA(c,a)	L1: - - - - - -	L1: - - - - - -
- - - - - -	go ‘procA’	*setup parameters*
- - - - - -	*setup parameters*	call procA
	leaplink 0	
procA: - - - - - -	procA: - - - - - -	procA: - - - - - -
- - - - - -		

Figure 6.5 C-code compiled into HORN code and Energy-efficient code

1. The ‘Energy Efficient code’ might even be smaller than the HORN-code dependent on how the literal field in the ‘bnZ’-instruction is encoded. If a range of offset sizes is possible the ‘bnZ’-instruction might not need to be larger than the corresponding ‘leapnZ’ instruction in the HORN-code and the overall size of the code is therefore reduced.

C-code	**HORN-code**	**Loop-instruction**
....	....	
for (i=0; i < N; i++)	go L1	....
s += a[i]*b[i];	L1:	loop N,size
....	*s += a[i]*b[i]*	size *s += a[i]*b[i]*
....	*i++*	*i++*
	cmp Ra,i,N	....
	leapNZ Ra,0	....
	.....	

Figure 6.6 The principle of a ‘loop’-instruction

instruction cache when the procedure is called. Furthermore, the ‘touchI’ instruction can be omitted if it is not expected to improve the performance.

A way of reducing the overhead of the branch instructions further would be to introduce a ‘loop’ instruction, see Figure 6.6, which can eliminate the branch penalty completely for simple loops. The loop instruction would iterate ‘N’ times over ‘size’ instructions. As the loop-body contains fewer instructions than the corresponding HORN- (or RISC-) code the execution time is reduced; by a factor dependent on the values of ‘size’ and ‘N’. This technique is similar to loop-pipelining described in [Bird2].

6.4 Two-instruction CTI in a dual-issue implementation

Previous sections have shown that a two part CTI is not energy efficient in a single instruction issue implementation of the HORN-architecture.

Although this thesis, in general, considers multiple instruction issue as an ‘implementation-technique’ which consequently is not investigated in the rest of this thesis, this section will briefly discuss the value of a two-instruction control transfer structure in a dual-issue implementation of the HORN-architecture.

The benchmark suite has been analysed for register dependencies and the instruction stream has been re-organised to try to form instruction packets of two instructions, which can be issued in parallel. Register dependencies in the packets were not permitted and each packet could contain only one memory accessing instruction: ‘load’, ‘store’, ‘spill’ or ‘fill’. This avoids coherency problems; for example if there were two ‘fill’ operations in the same instruction packet it would not be obvious which one to execute first and thereby which register gets which value.

The results presented in Table 6.10, show that a packet contains an average of 1.44 instructions. Table 6.4 showed that the average unbroken sequence comprises 6.46

Table 6.10 Average number of instruction issued per cycle

Benchmark	Average number of instructions issued per cycle
cacti	1.43
dhry	1.47
espresso	1.44
fft	1.40
flex	1.44
hello	1.47
stcompiler	1.44
Average	**1.44**

instructions. Combining these two results shows that an unbroken dual instruction sequence would contain:

$$\frac{6.46 Instructions}{1.44\frac{Instructions}{Issue}} = 4.5 Issues \qquad \text{(EQ 6.7)}$$

As there are few restrictions as to where within the sequence the ‘go’-instruction can be placed it would normally be possible to place it in an unused issue slot. Consequently, it is not expected that it will be necessary to introduce more issues than if a conventional - single instruction - branch architecture were adopted.

Simulation has shown that the average basic block contains 5.34 issues if a RISC style (single instruction) branch is assumed. If the two-instruction CTI is assumed the average number of issues increases to 5.69. There is therefore not the same cycle overhead associated with the CTI-architecture as was the case for the single instruction issue considered in sections 6.1 and 6.2.

Section 6.2 showed how the information from the go-class instruction can be used to prefetch the target of the branch and thereby eliminate any need for branch prediction by fetching from both targets. The instruction fetch architecture presented in Figure 6.4 can also be used in a dual-issue implementation however, the size and complexity of such a module may make its implementation impracticable. The energy consumption per cycle will go up as there will be relatively more cycles where two-instruction packets need to be fetched. The energy consumption in the instruction cache relative to an 'ideal' scheme with 100% branch prediction accuracy and hence no need for prefetching is:

$$Energy_{Relative}(Prefetch) = 1 + \frac{2}{5.69} = 1.35 \quad \text{(EQ 6.8)}$$

i.e. an energy increase of 35% per cycle.

As in section 6.2 the more conventional branch architecture together with a simple 'branch-taken' prediction scheme and delay slots will have a relative energy consumption of:

$$Energy_{Relative}(Prediction) = 0.94 + (1 - 0.77) \times \frac{2}{5.34} + 0.77 \times \frac{1}{5.34} = 1.23 \quad \text{(EQ 6.9)}$$

This differs by 12% from the energy consumption of the prefetch scheme which has to be compensated for by the faster execution under the two-instruction CTI architecture for the scheme to be energy efficient.

Relative to the single instruction branch, the execution time of a basic block under the dual instruction scheme will be 23% higher:

$$T_{\text{Dual,77\%Acc.}} = 0.94 + (1 - 0.77) \times \frac{2}{5.34} + 0.77 \times \frac{1}{5.34} = 1.23 \qquad \text{(EQ 6.10)}$$

Table 6.9 shows the energies and execution times calculated above and calculates the energy efficiency, EE. The result shows that the energy efficiency is higher for the prefetch scheme than for the prediction scheme.

Table 6.11 EE for prefetch and branch prediction schemes (dual issue)

	Prefetch	**Branch Prediction**
Energy	1.35	1.23
Delay	1.00	1.23
$EE = \frac{1}{Energy \times Delay}$	**0.74**	**0.66**

This discussion therefore concludes that the two instruction CTI architecture would be significantly faster (23%) and more energy efficient that a simple single-instruction ‘predict taken’ branch scheme in a dual instruction issue implementation of the HORN architecture.

6.5 Summary

This chapter has shown how the two-instruction control transfer instruction architecture affects cache performance and thereby the energy efficiency for the processor. Section 6.1.1 showed that the structure had a positive effect on the hit-rate for all benchmarks.

Subsequent sections assumed that the target of a CTI could always be prefetched into the instruction cache before it was required. This was clearly a very optimistic assumption especially considering the short average distance between CTIs presented in Table 6.4. Despite this, the increase in hit-rate is not sufficient to compensate for the increased instruction count, relative to conventional branch and jump instructions, to have a positive effect on execution time and energy efficiency except for small caches and long memory latencies.

Section 6.2 analysed the effect of the two-part CTI as a way of eliminating or reducing the branch penalty. It was shown that, although there is a gain in performance associated with fetching instructions from both targets of a branch, this gain is not sufficient to offset the significant increase in energy consumption (and complexity) in the instruction cache. The architecture is therefore not as energy efficient as a conventional single instruction branch architecture. Ways of improving the branch prediction were introduced, but these further widened the difference between the go-leap-scheme and the traditional RISC branch scheme in favour of the latter.

This section therefore concludes that two-part control transfer instructions may improve the performance if instructions from both targets are fetched in parallel as described in section 6.2. Simply using the information in the go-instructions to improve the hit rate and/or reduce the miss penalty in the instruction cache is not sufficient to make the scheme perform better than a conventional single-instruction branch. Despite this potential performance advantage the scheme is not energy efficient due to the increased number of instruction cache accesses.

In a dual issue implementation the performance advantage of the two-instruction CTI is so high that it is not offset by the lower energy consumption of the conventional single-instruction branch, see section 6.4. The two-instruction CTI would consequently perform better and be more energy efficient than a single-instruction branch in a dual issue implementation.

Chapter 7 Register file architectures

7.1 Introduction

Registers are the lowest level in the memory hierarchy, i.e. closest to the processor core. In a RISC architecture [Patt] all instructions operate on registers. An instruction such as ‘add R1,R2,R3’ will read its input operands from register 2 and register 3 and store the sum into register 1. In a CISC architecture [Robin] one or more of the operands might come from a memory location, eventually referenced through a register: ‘add R1,R2,offset(R3)’.

[Tiwari] analysed the power consumption in a 486DX2 processor and found that instructions which accessed only the register file drew 300mA while instructions which fetched an operand from the data cache drew 430mA. Instructions which wrote their result to the cache drew 530mA. Instructions which accessed the cache were shown to consume significantly more energy than the ‘pure’ register instructions. The access to the large data cache should therefore be kept at a minimum. This can partly be done by specifying a register file architecture which minimizes the need to spill/fill registers to/from the data cache. In either class of architecture the register file will be accessed heavily and should therefore be designed carefully.

Some architectures do not specify a register file as described above. The Hobbit [Argade] and Transputer [Transputer] architectures use stacks for temporary storage and the Hobbit also allows memory-to-memory operations. These ‘local-storage’ architectures will not be discussed further as their performances are difficult to assess without access to compilers and simulators.

There are seldom enough registers to hold all the variables required through the execution of a program, so data must often to be saved to, and later retrieved from, elsewhere in the memory hierarchy. Fortunately, variables are not used evenly throughout a program. A local variable in a function will be required only within the function or its setup and return. Variables will therefore need to be in scope only at various phases in the execution of a program rather than throughout the whole program. It is not necessary to keep all variables in registers, only the ones currently in scope. Consequently, the way an architecture handles allocation and de-allocation of registers affects not only the performance of the processor but also the energy efficiency as these processes often involve memory accesses.

The HORN architecture [HORNV3, HORNV5], which forms the basis for this work, has undergone several changes of register file architecture during the evolution of the project; this has provided the opportunity to compare a number of register file architectures.

The first architecture described here uses a model where registers are memory mapped through a pointer; allocating/de-allocating was performed by adjusting this pointer. A second architecture, which works by renaming registers in a conventional register file, has also been investigated. This second scheme uses two instructions, spill and fill, to allocate and deallocate registers. Finally these two architectures will be compared with the commercially available SPARC register-window architecture [Weaver].

7.2 Temporary storage

The HORN-architecture specifies three types of local storage: Global registers, local registers and a four element operand queue. The first two types of register will be described in depth in the following sections, while this section evaluates the value of the operand queue for performance and energy efficiency.

As described in Chapter 3, the HORN-architecture defines a four element first-in-first-out operand queue (OQ). This is intended to store temporary values which need to be accessed only once. The chapter also showed how the implicit referencing of this queue can be used to reduce the size of instructions and how it reduces the need for registers.

As Chapter 8 and Chapter 9 will show, the reduced instruction size increases the performance of the instruction cache and consequently improves the performance and the energy efficiency of the entire system. Furthermore, fewer registers will need to be saved and later restored due to register shortage, implying a further improvement in both performance and energy efficiency.

The HORN architecture allows the queue to be addressed through special bits in the instruction format, but it can also be addressed through its register alias (register 63). Accessing a queue through a register alias would allow OQ's to be added to many other architectures. As long as the queue has a significant length (>2) it is expected to have a positive effect on performance as it effectively represents an extension of the register file, without increasing the number of bits required to access it.

7.3 Memory mapped registers

As well as the OQ, release 3 of the HORN architecture [HORNV3] specifies two other types of on-chip storage, register:

- 16 global registers organized in a conventional register file
- 32 memory mapped local registers

where the 32 local registers are mapped to memory through a pointer, LPTR.

The value of the LPTR is controlled via an instruction ‘adjust local register pointer’, ‘ajlp’. The instruction can take two formats:

1. ajlp <signed constant>
2. ajlp reg

In the first format, the LPTR is updated by adding the signed constant to the existing value of LPTR. The mechanism can be used to implement overlapping register windows. The second format overwrites LPTR with the contents of another register.

The architecture suggests that the memory mapped, local registers reside in a separate *register cache* backed up by the first level data cache.

Upon a function call the LPTR is decremented as shown in Figure 7.1. In this way, a number of new registers are allocated while another set goes out of scope. The contents of these newly allocated registers will, by definition, be invalid. It is therefore *not* necessary to fetch the contents of the addresses from the rest of the memory hierarchy, but only allocate the new registers. The first access to these registers should therefore be a write which will make the register ‘dirty’.

When a function returns to its caller it deallocates the registers which were allocated when it was entered. The deallocated registers will typically reside in the register cache and be marked ‘dirty’. However, according to the architecture specification, [HORNV3], the registers need *not* be written back to memory so this potential write-back should be avoided whenever possible as it is likely[1] to cause an off-chip reference and hence increase the energy consumption. It is sufficient just to let the de-allocated registers remain in the register cache and eventually be written back to memory whenever the cache line is reallocated later. The write-back can be avoided if, upon de-allocation, lines

1. Assuming a high hit-rate in the register cache, the *data* cache is not likely to contain the line which is written back from the register cache.

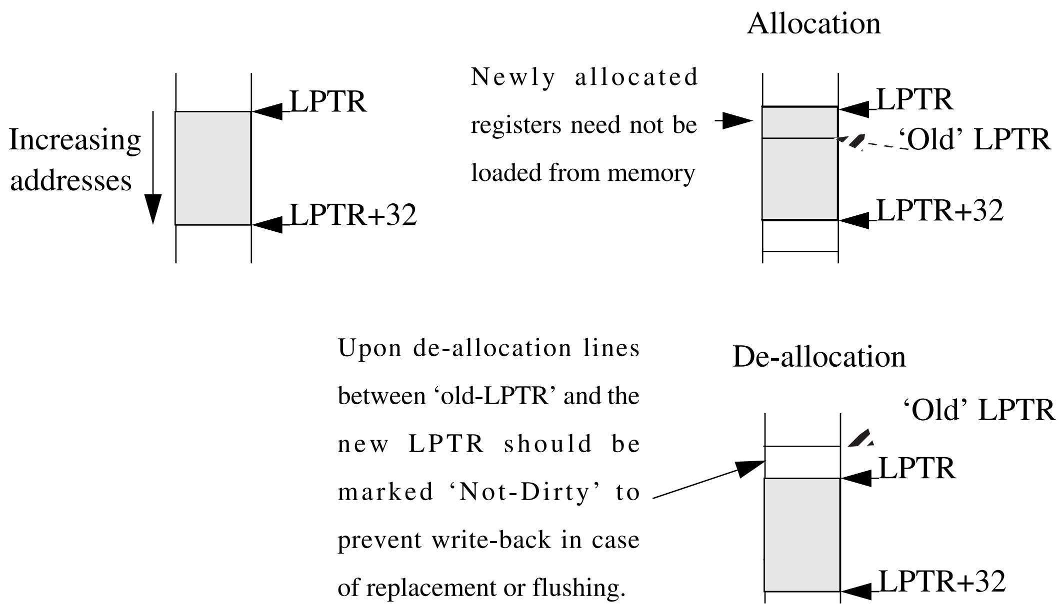


Figure 7.1 Allocating and de-allocating registers

between the ‘old’ and the new value of LPTR are marked ‘not-dirty’, see comment in Figure 7.1.

The architecture specifies that registers are allocated, and deallocated, in multiples of four. It is possible to (de-)allocate any multiple of four registers, but statistics show that the majority of changes to LPTR are four or eight (see Table 7.2, on page 141).

When an ‘ajlp’ instruction is issued with a register reference, i.e the LPTR is going to be overwritten rather than adjusted, the status of the cache lines should not be touched. This allows efficient handling of interrupts, process and thread changes.

The following sections describe how a register cache can be designed to yield an energy effective implementation.

7.3.1 Number of ports

The majority of instructions in the HORN architecture are of RISC style, i.e. two source registers and one destination register. This implies that the register cache needs to be able to handle a similar number of accesses per cycle; it should have two read ports and one write port. Only a single instruction format fails to match this scheme, the ‘st r1, r2, r3’ instruction[1] which reads three operands. This, infrequently used, instruction format[2] could be replaced by two instructions: ‘add rt, r2, r3 followed by st r1,rt,0’ and thus avoid extra hardware only used by this instruction. The penalty for the original instruction format might not be obvious in a non-pipelined architecture, but if the implementation is pipelined it will be necessary to add a fourth port to the register file or risk waiting for the pipeline to drain before the store instruction can be issued.

As many instructions do not use all three registers ports, it should be possible to disable the ports which are not required for a given cycle in order to save power [Yeung].

7.3.2 Total size

Figure 7.2 and 7.3 show the value of LPTR during the execution of two of the benchmarks, hello and stcompiler (LPTR was initialized to 1000). Hello is shown because it is a small benchmark and the variations are therefore easier to see in the figure.

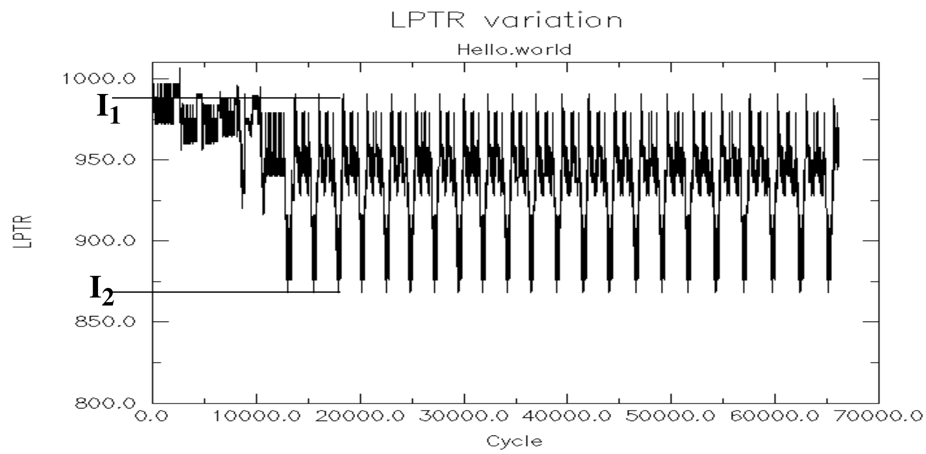


Figure 7.2 Variation of LPTR during execution, hello

1. The contents of r1 are written to the address given by adding the contents of r2 to the contents of r3
2. Eventually on the fly

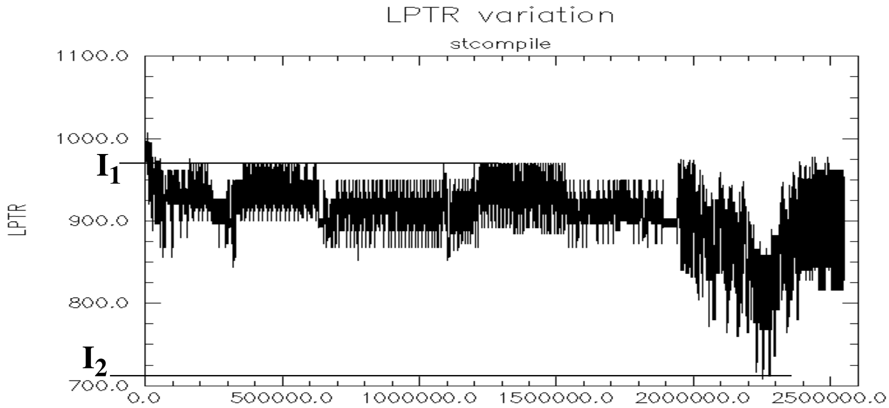


Figure 7.3 Variation of LPTR during execution, stcompiler

Stcompiler is chosen, as it is considered more representative of user programs and because it shows a significant deviation towards the end of the execution.

It appears that for most programs LPTR stabilize after an initialization period. For espresso, the 'LPTR' stabilizes in the interval $[I_1,I_2]$ = [-60,-136] relative to the initial address. For flex the equivalent interval is [-28,-116]. As each register window contains 32 registers the total number of registers accessed in the 'relevant' parts of the program is equal to I_1-I_2+32. As it can be seen from Table 7.1 this total does not exceed 128 registers

Table 7.1 LPTR limits

Benchmark	I_1	I_2	I_1-I_2+32
espresso	-60	-136	108
flex	-28	-116	120
hello	-10	-132	154
stcompiler	-30	-290	292

for the first two benchmarks. This means that if the register cache holds 128 words there should be no misses in the register cache except the compulsory misses. As shown in Chapter 5, the energy cost associated with fetching a word from a cache scales with the size of the cache. The energy cost of a hit in a register cache may therefore be small compared to the cost of accessing the first level data cache which may be 8K bytes or more. From an energy perspective, it is sensible to insert a ‘small’ register cache and thus reduce the number of accesses to the more energy consuming first-level data cache.

For stcompiler, see Figure 7.3, LPTR stabilizes very quickly and remains in a narrow band before descending to a relative offset of almost 300 registers. Finally it climbs back to an offset of around -100. Note that LPTR does not vary randomly between -30 and -300. It varies within a ‘band’ of fairly constant width. This also explains why stcompiler performs well even with small register caches, see Table 7.7, on page 147.

Hello stabilizes in the interval [-10,-132]. As Table 7.1 shows, the sum (I_1-I_2+32) exceeds 128. The fact that this program performs very well anyway is related to its very restricted register usage, see Tables 7.3 and 7.6.

7.3.3 Line size

The size of a cache line is closely related to the total size of the cache. As stated earlier the HORN architecture [HORNV3] uses the ajlp instruction to implement register windows with a variable sized overlap region.

Table 7.2 ajlp offset distribution and frequency

	ajlp offset					
Benchmark	**4**	**8**	**12**	**16**	**20**	**24**
espresso	33,156	36,988	7,024	301	0	8
	42.8%	*47.7%*	*9.1%*	*0.4%*	*0.0%*	*0.0%*
flex	133,859	20,944	7,668	470	0	3700
	80.3%	*12.6%*	*4.6%*	*0.3%*	*0.0%*	*2.2%*
hello	474	1,119	370	0	0	44
	23.6%	*55.8%*	*18.4%*	*0.0%*	*0.0%*	*2.2%*
stcompiler	63,796	31,176	6,124	8	0	363
	62.9%	*30.7%*	*6.0%*	*0.0%*	*0.0%*	*0.4%*

As can be seen in Table 7.2 the majority of ajlp instructions specify offsets of four or eight words and the register cache line size should reflect this. However, it is not clear whether which line size would be optimal. The register cache lines should be made short to allow as great a flexibility as possible and to minimize the traffic towards the rest of the memory hierarchy in case of misses. Once the line size has been determined the number of lines is calculated as:

$$lines = \frac{totalsize}{linesize} \qquad \text{(EQ 7.1)}$$

i.e. 32 or 16 lines.

As mentioned above, the line size should be as short as possible, i.e. a register cache with 32 lines/4 words per line performs better than a cache with 4 lines/32words per line. Similarly a 16/8 cache performs slightly better than a 8/16 cache, see Tables 7.4 - 7.7. It was decided to measure performance in terms of 'stalled cycles', which is the increase in the number of cycles required to execute a program due to the register cache. A high number of stalled cycles implies many fetches or writebacks from/to the rest of the memory hierarchy. The tables also show the stalled cycles as a percentage of the total cycle count for the configurations.

The results are optimistic because the model assumes that register cache misses will always hit the on-chip first level data-cache and that the register cache can always access the data cache with a rate of one word per cycle. This means that a miss on a line costs '*number of words per line'* cycles if the cache line to be replaced is clean, and *2*'number-of-words-per-line'* cycles if the line needs to be written back to the data cache. This model is accurate for small register cache configurations where capacity misses [Patt] dominate. In these cases the first-level data cache is likely to hold copies of the missing data and the register cache can therefore be served from it, implying the low miss-penalty. For large configurations compulsory misses dominate. Compulsory misses in the register cache can result in compulsory misses in the first level data cache as well. The likelihood of this is dependent on the ratio of line sizes in the register cache and the first-level data cache. If there is a compulsory miss in the data cache, the penalty for the register cache miss is clearly higher than stated above.

The hit-rate for all cache configurations greater than or equal to 128 words (512 bytes) is over 99%, see Tables 7.4 - 7.7.

7.3.4 Associativity

Although the register cache is not believed to be the component which determines the cycle time of the processor system; it should be fast to allow zero-detection etc. used by branch instructions to be carried out in the same cycle, see Chapter 6. Using "cacti"[1], Figure 7.4 shows how the cycle time for a single-ported cache increases with the degree of associativity. Going from a direct mapped cache to a 2-way set-associative cache increases the cache cycle time by more than 50%.

Furthermore the variation of LPTR in a very limited address space during the execution of a program, see Figures 7.2 and 7.3, makes it unlikely that the hit-rate of the register cache will be improved if the associativity is increased. Furthermore a direct-mapped cache will consume less energy due to a lower overhead in tag-comparisons.

Although the cycle time may not increase linearly with the number of ports, it is believed that the trend in the results presented for a single ported cache will not change with the number of ports; i.e. a direct mapped cache with short lines is the most energy efficient configuration.

1. a cache timing simulator from Digital [Wilton]

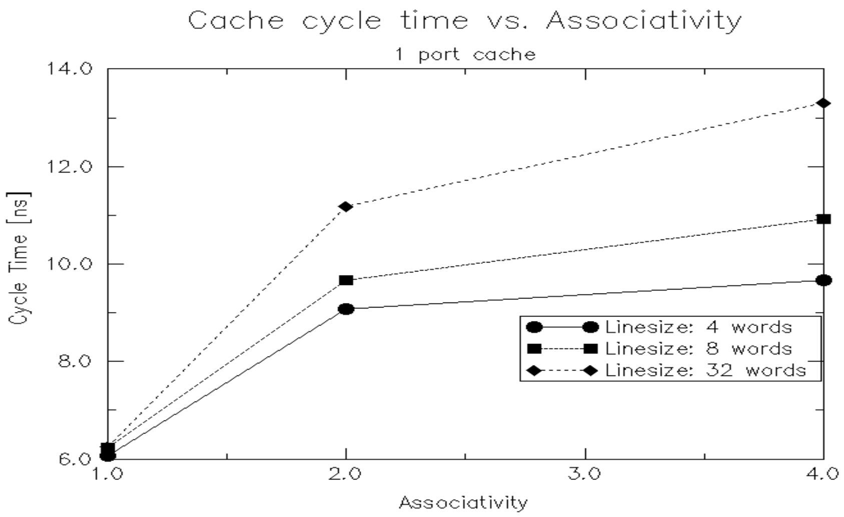


Figure 7.4 Cycle time vs. associativity for a 512-byte - 1 ported cache

7.3.5 Writeback policy

The writeback policy should be chosen to optimize for energy-efficiency. Of the two alternatives:

- Write through
- Copy back

copy back is the most energy efficient, especially when the hit-rate is very high. This policy ensures that accesses to the rest of the memory hierarchy are kept to a minimum. The cache should use a 'Write allocate' [Patt] strategy, since it is very likely that a write to a register will be followed by a read from the same register later.

It has been assumed that the memory mapped registers (in or out of scope) are never accessed through regular load and store instructions. This might be difficult to ensure, especially at the operating system level. It has therefore been suggested a 'flush' instruction should be introduced. This would flush the register cache to the next level in the memory hierarchy, thereby introducing synchronization points. There should be little dynamic use of such a flush instruction.

7.3.6 Results

Tables 7.4 - 7.7 show the number of stalled cycles, i.e. the number of cycles the program execution was stalled, due to register cache misses. Furthermore the tables show the number of stalled cycles as a percentage of the execution time, see Table 7.3.

Table 7.3 Execution time assuming 100% hit-rate in register cache

Benchmark	Number of cycles[a]
espresso	7,179,098
flex	10,092,296
hello	66,145
stcompiler	2,549,324

a. 8K byte instruction cache and 8K byte data cache, 32 bytes per line, direct mapped. Branch penalties and stalls due to register dependencies are ignored

The number of stalled cycles gets smaller for larger register cache, but it is a wrong to conclude that a smaller number of stalled cycles is better for energy efficiency, as will be explained below.

The register cache is assumed to be empty and 'clean' before a program starts executing. This implies that the first access to each line will cause a compulsory miss [Patt] and hence add 'number-of-words-per-line' cycles to the execution time. Hence if a number in the tables (Tables 7.4 - 7.7) is smaller than the total size[1] of the cache it means that not all the lines in the cache have been accessed. By looking at the results presented in Tables 7.4 - 7.7, one can see that for the respective benchmarks there were only 224, 192, 144 and 304 stalls for a cache with 32 lines each containing 16 words implying a total size of 512 words. This mean that the utilization of the cache is very low, 44%, 38%, 28% and 59% and unused lines implies wasted energy.

1. size = #lines * 'line size'

As the energy consumption of a cache increases with the cache size, the register cache should be made as small as possible; comparing Table 7.3 with Tables 7.4-7.7[1] it can be seen that the relative performance penalty for building a 128 word[2] register cache is very small, well under 1%. The utilization of the cache lines is high.

Table 7.4 Stalled cycles due to register cache misses, espresso

#lines line size [words]	4	8	16	32
4	834,120	111,588	18,556	1,384
	11.6%	*1.6%*	*0.26%*	*0.02%*
8	157,432	23,312	1,560	224
	2.2%	*0.32%*	*0.02%*	*0.0%*
16	34,128	2,000	224	224
	0.48%	*0.03%*	*0.0%*	*0.0%*
32	3,584	256	256	256
	0.05%	*0.0%*	*0.0%*	*0.0%*

Table 7.5 Stalled cycles due to register cache misses, flex

#lines line size [words]	4	8	16	32
4	1,964,616	291,172	6,084	384
	19.6%	2.9%	0.06%	0.0%
8	544,984	10,976	512	184
	5.4%	0.11%	0.01%	0.0%
16	22,176	544	192	192
	0.22%	0.01%	0.0%	0.0%
32	832	224	224	224
	0.01%	0.0%	0.0%	0.0%

From Table 7.2 it can be seen that the most frequent ajlp-offset is either four or eight words. This should indicate that the line size should be four or eight words. From examining Tables 7.4-7.7 there does not seem to be any advantage in choosing a line size of eight words. A line size of four words performs better than one of eight words and as

1. The 128 word configurations are highlighted
2. 512 bytes

Table 7.6 Stalled cycles due to register cache misses, hello

#lines line size [words]	4	8	16	32
4	41,532	12,164	5,068	144
	62.8%	*18.4%*	*7.7%*	*0.22%*
8	15,792	5,824	160	144
	23.9%	*8.8%*	*0.24%*	*0.22%*
16	5,920	160	144	144
	9.0%	*0.24%*	*0.22%*	*0.22%*
32	192	160	160	160
	0.29%	*0.24%*	*0.24%*	*0.24%*

Table 7.7 Stalled cycles due to register cache misses, stcompiler

#lines line size [words]	4	8	16	32
4	835,844	246,796	69,860	4,416
	32.8%	*9.7%*	*2.7%*	*0.17%*
8	346,532	96,560	6,232	400
	13.6%	*3.8%*	*0.24%*	*0.02%*
16	133,168	8,496	416	304
	5.2%	*0.33%*	*0.02%*	*0.01%*
32	18,528	448	320	320
	0.73%	*0.02%*	*0.01%*	*0.01%*

the energy consumption in a cache is more sensitive to increasing line size than increasing total-size the shorter 4-byte cache lines also yield better energy efficiency that 8-byte cache lines.

To determine the most energy efficient register cache configuration it will be assumed that all instructions access two source operands from the register cache and write one result back.

To simplify the energy expressions below, this section will assume that read and write requests consume the same amount of energy despite the findings in Chapter 5. Equation 5.33 is used to assess the energy consumption in the caches.

Assuming 100% hit rates in the instruction and data caches, the execution time of a system where both register- and data references are served from the data cache is proportional to the number of instructions and will be denoted T_0. The data cache in such a system needs to be multi-ported to accommodate both register and load/store references.

If a multi-ported *register* cache serves the register references, the cycle count increases due to misses in the register cache. It is assumed that misses in the register cache will have a 100% hit-rate in the data cache. The increase in cycle count will be denoted 'Stall'. The execution time of such a system is thus 'T_0 + Stall'. Note that the data cache does not need to be multi-ported in such a configuration.

The energy consumption in a system with just instruction and data caches, E_0, can be expressed as:

$$E_0 = E_{Core} + E_{Icache} + E_{Dcache, Multiported} \quad \text{(EQ 7.2)}$$

while the energy consumption in a system incorporating a register cache can be expressed as:

$$E = E_{Core} + E_{Icache} + E_{Dcache, Singleported} + E_{RegCache, Multiported} \quad \text{(EQ 7.3)}$$

As the energy consumption in the instruction cache and the processor core is independent of the register file implementation they will be left out of the computation:

$$E_0 = E_{Dcache, Multiport} \quad \text{(EQ 7.4)}$$

$$E = E_{Dcache, Singleport} + E_{RegCache, Multiport} \quad \text{(EQ 7.5)}$$

Section 5.1.3 has shown that the energy consumption in a multi-ported RAM scales with the number of ports. The energy consumption in a multi-ported cache is therefore approximated to be:

$$E_{Cache, MultiPorted} = NbPorts \times E_{Cache, Singleport} \quad \text{(EQ 7.6)}$$

The data cache in the system which serves all references from the data cache is multi-ported, it must be able to accommodate four accesses per cycle: three register references and one ld/st reference. The number of ports should thus be 4. However, the number of active, and thus energy consuming, ports is less than four.

The data cache in the system with a separate register cache need only be single ported. The register cache, however, should have three ports ($NbPorts_{Reg}$). All ports in the register cache are assumed active, and hence energy consuming, when executing instruction while only one port will be active when cache misses are being served.

E_0 can therefore be calculated as:

$$E_0 = Inst \times NbPorts_{Dcache,\ Multiport} \times E_{Dcache,\ 1port} \quad \text{(EQ 7.7)}$$

while E is calculated as:

$$E = (Inst \times NbPorts_{Reg} + Stall \times 1) \times E_{RegCache, 1Port} + (LdSt + Stall) \times E_{Dcache,\ 1Port} \quad \text{(EQ 7.8)}$$

The energy efficiency (EE) of a system with separate register cache relative to a system with a multi-ported data cache, (EE_0), can thus be expressed as:

$$EE = \frac{E_0 \times T_0}{E \times T} \times EE_0 \quad \text{(EQ 7.9)}$$

Inserting the expression from E,E_0, T and T_0 derived above gives:

$$EE = \frac{NbPorts_{DcacheMultiPort}}{\frac{(Inst + Stall) \times (Inst \times NbPorts_{Reg} + Stall)}{Inst^2} \times \frac{E_{Reg1Port}}{E_{D1Port}} + \frac{(LdSt + Stall) \times (Inst + Stall)}{Inst^2}} \times EE_0 \quad \text{(EQ 7.10)}$$

The number of active ports in the multi-ported data cache, $NbPorts_{DcacheMultiPort}$ is thus an important parameter. On average one in three instructions is a memory referencing instruction, see section 4.4. The average number of active ports is thus 3.33; three ports to serve register references plus 0.33 for the memory referencing instructions.

Only a direct mapped, 8K byte data cache with 32-byte cache lines will be examined. Tables 7.8 - 7.11 show the variations in EE/EE_0 for a number of register cache configurations. Changing the data cache from 8K-byte to 4K- or 16K-byte will not change the internal ordering, but only the actual values and the relative differences.

Table 7.8 EE/EE_0 for different register cache configurations, espresso

#lines / line size [words]	4	8	16	32
4	1.47	1.92	1.98	1.97
8	1.19	1.24	1.24	1.22
16	0.71	0.71	0.71	0.69
32	0.38	0.38	0.38	0.37

Table 7.9 EE/EE_0 for different register cache configurations, flex

#lines / line size [words]	4	8	16	32
4	1.39	1.81	1.88	1.86
8	1.12	1.20	1.20	1.18
16	0.70	0.70	0.69	0.68
32	0.38	0.38	0.38	0.37

Table 7.10 EE/EE_0 for different register cache configurations, hello

#lines / line size [words]	4	8	16	32
4	0.63	1.29	1.59	1.85
8	0.78	1.01	1.19	1.17
16	0.59	0.70	0.69	0.68
32	0.38	0.38	0.37	0.37

Table 7.11 EE/EE_0 for different register cache configurations, Stcompiler

#lines / line size [words]	4	8	16	32
4	0.96	1.52	1.78	1.87
8	0.92	1.12	1.20	1.19
16	0.63	0.70	0.69	0.68
32	0.38	0.38	0.38	0.37

Values in the tables less than 1.0 indicate that it is more energy efficient to omit a register cache and build a four-ported data cache.

The optimal configurations are highlighted. The optimal cache size is 128 words (512 bytes), and lines should be short, containing just 4 words (16 bytes). The tables show that short lines are essential as the $\frac{EE}{EE_0}$ measure decreases with increasing line size.

A cache size of only 16 words is included in the tables. This is clearly too small as version 3.0 of the HORN architecture specifies 32 visible local registers at any time, but it is interesting to see that despite all the extra traffic towards the data cache and the decreased performance, it yielded a higher EE than many of the configurations with long lines.

The performance measurements, (Tables 7.4 - 7.7) clearly indicate an optimal configuration of 128 words organized with short cache lines. The energy efficiency measurements also pointed towards a 128 word cache. For the Flex and Stcompiler benchmarks a 128 word cache was not the optimal configuration, however, the difference between the chosen and the optimal configurations for these benchmark is very small (approximately 1%).

7.3.7 Summary

Section 7.3 has, based on a number of benchmarks, determined the cache parameters for a register cache:

Total size:	128 words (512 bytes)
Line size:	4 words (16 bytes)
Associativity:	Direct mapped
Ports:	2 Read and 1 Write
Write back policy:	Copy back, blocking

The register cache is likely to 'receive' addresses from a very restricted area of the address space and the Tag-store in the cache is therefore expected to contain multiple identical values. Implementing the register cache as either a sectored-cache or CAT-cache, see Chapter 5, with only 2 or 4 tag-values stored would be likely to reduce the energy consumption of the register cache without affecting the performance.

7.4 Spill/fill

Release 5 of the HORN architecture [HORNV5] changed the register file architecture. It still specified 16 global registers, (*g*0-*g*15), 32 local registers, (*l*0-*l*31) and a 4 element operand queue (OQ), but the local registers were not mapped to memory.

This scheme makes use of 'spill' and 'fill' instructions. These rename registers and spill/fill four local registers to/from memory. This means that the local and global registers can all be held in a conventional register file. Since there are 32 + 16 = 48 registers and the first element of the operand queue is mapped to register 63, there are 15 unused registers addresses in the instruction format, see Figure 7.5, which are intended to contain constants. Alternatively the operand queue could be a part of the register file.

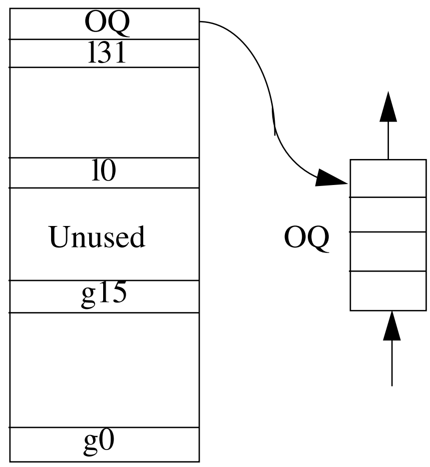


Figure 7.5 Register layout

The scheme is similar to the ajlp scheme described in section 7.3 in that a *spill* instruction has much the same effect as an ‘ajlp-4’ instruction and a *fill* instruction the effect of ‘ajlp+4’, see Figure 7.6.

Initially this was thought to imply a significant performance penalty (20%) over the register cache architecture, described in section 7.3, as the four memory references will block the data cache for four cycles hence preventing other memory referencing instructions from accessing the data cache. This section proposes implementation schemes which minimize this penalty by overlapping memory references from the spill/ fill activity as far as possible with other instructions. Simulation results will be presented.

HORN Release 3	***HORN Release 5***
ajlp -4	spill reg
--	--
--	--
ajlp +4	fill reg

Figure 7.6 Two register (de-)allocation schemes

In the ajlp-scheme, presented in section 7.3, the register cache minimized the traffic to the larger and hence more energy consuming first level data cache. Data was not moved repeatedly between the two levels in the memory hierarchy as the spill/fill scheme would require and minimizing ‘unnecessary’ traffic helps to minimize the energy consumption and increases performance hence optimizing the energy efficiency, EE. Use of the spill/fill scheme implies that the first-level data-cache traffic will increase significantly, by as much as 100%, compared to the scheme presented in section 7.3.

7.4.1 The spill/fill scheme

The semantics of the spill/fill instructions are such that a small offset counter is required to map the register numbers in instructions to addresses in the register file. The offset is decremented by four for a spill instruction and incremented by four for a fill. The registers which are allocated following a spill instruction are undefined. A simple way of implementing this is to consider the 32 local registers as a circular buffer and every spill/fill instruction marks the four registers (to be spilled/filled) as being “unavailable” and sets off a spill/fill engine. Program execution can then continue and the four registers will be spilled-to/filled-from memory at the same time as other instructions execute, assuming there are ports available on the register file for the spill/fill engine access its operands.

As energy consumption grows linearly with the number of ports in the register file [VLSI], it is proposed to have only three ports on the register file. Thus, if an instruction requires access to the two read ports and another instruction completes writing to the register file, the ‘spill/fill’ engine will have to stall until a register port becomes available. Here two things can happen:

1. An instruction sequence does not use all three ports to the register file all the time and the spill/fill engine will eventually write/fetch all the data to/from memory.

2. An instruction tries to access a register which is reserved by the spill/fill engine. The instruction issuing will stall while the spill/fill completes and resume once the register has been freed by the engine. Progress is thus ensured.

To implement this a register scoreboard is necessary. Scoreboarding should be based upon the physical address in the register file rather than on the register number itself, as the mapping changes continuously. This implies that the scoreboard architecture becomes complicated. It is not sufficient only to check three references per cycle. It will also be necessary to 'reserve[1]' four registers while the spill/fill engine should be able to release a register once it has been spilled/filled, see Figure 7.7. Furthermore it will be necessary to

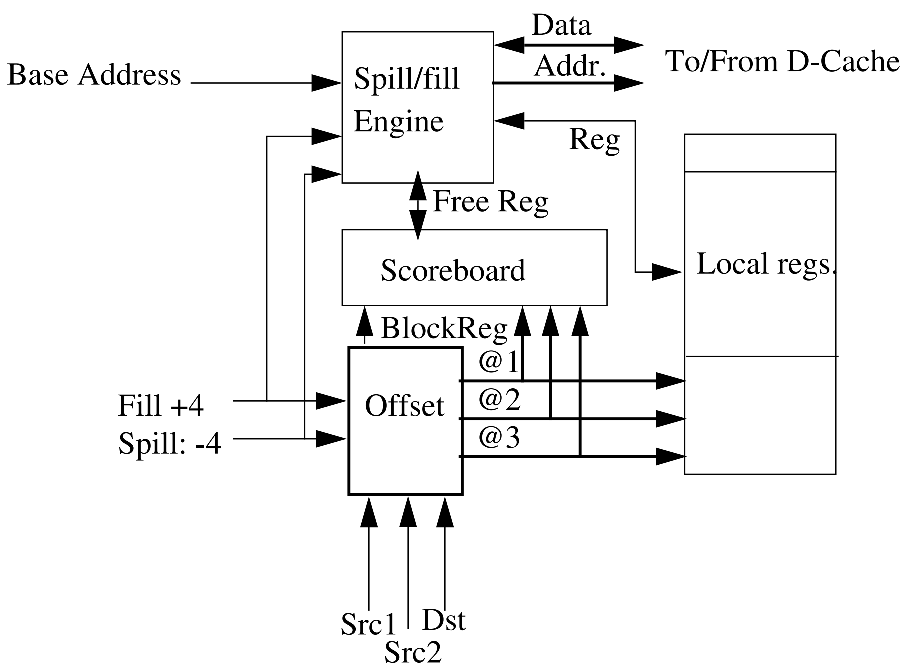


Figure 7.7 Block diagram of register file

1. i.e. mark them as being used

'scoreboard' the spill/fill engine itself. It will be very complicated to let the spill/fill engine serve a queue of spill/fill instructions. Hence if a spill or fill instruction is ready to be issued and the spill/fill engine has not completed a previous spill/fill instruction, the issuing of the new instruction should be delayed.

In addition, it is possible that there could be a load or store instruction in the shadow after the spill/fill instruction. This could lead to congestion of the bus to/from the first level data cache. To prevent this and more complicated scenarios, it is proposed that load/store instructions should be prevented from issuing until the spill/fill process has completed.

7.4.2 Statistics

Simulations show that when using this spill/fill mechanism, the memory traffic[1] might increase by up to 100% when compared to the ajlp scheme presented in section 7.3. Table 7.12 shows how the number of memory references may grow from one memory reference for every nine instructions to one for every three instructions. The conclusion is that the spill/fill activity seriously affects the cache reference pattern and increases the memory traffic significantly. The following sections describe different approaches to the

Table 7.12 Memory access statistics

Benchmark	instructions	spill/fill[a]	ld+st	$\frac{Instructions}{(ld + st + 4 \times spill)}$ [V.5 of HORN architecture]	$\frac{Instructions}{(ld + st)}$ [V.3 of HORN architecture]
espresso	7,179,096	129,456	1,260,241	4.04	5.70
flex	10,039,395	166,642	1,878,977	3.94	5.34
hello	67,382	4,089	10,115	2.55	6.66
stcompiler	2,616,691	146,730	292,224	2.98	8.95

a. The number of spill/fills was determined by counting the number of ajlp instructions in an instruction trace under a tool generated with the tools for the Version 3 of the HORN architecture. An 'ajlp n' was converted into 'n' spill/fill instructions. This allows a fair comparison of the two schemes. Relying upon the compiler that came with the introduction of the spill/fill scheme would not be fair, as it also introduced many other optimizations.

1. defined as traffic towards the first level data cache

implementation of the spill/fill architecture, but it is clear that there will be a *performance* degradation compared to the 'ajlp' scheme. The different implementations will just limit the penalty.

The spill/fill architecture implies many more copy operations (between the register file and the first level data cache) than the earlier memory mapped scheme, see Table 7.13.

Table 7.13 References to the 1st level data cache due to the two schemes

Benchmark	128 word Register Cache[a]	4 * #Spill/Fill - instructions
espresso	1384	517,824
flex	384	666,568
hello	144	16,356
stcompiler	4,416	586,920

a. Stalled cycles in a 128 word register cache with short cacheline, 4 words/line, see Tables 7.4 - 7.7.

The two schemes are believed to lead to equivalent cycle times as both will involve adding an offset to the register numbers. A lookup in a small 128-word direct-mapped cache is not believed to be significantly slower than a lookup in a 64-word register file. The cache timing analysis tool Cacti [Wilton] shows a cycle time of 6.02ns for the cache and a 6.01ns cycle time for the register file, a difference of less than 1%.

The number of references to the large first-level data cache goes up in the spill/fill scheme, but the energy consumption per register reference is clearly lower than if a 128 word cache was accessed. A cache lookup is more energy consuming than a lookup in a normal register file, due to the overhead of the tag-store, This overhead might be more than 100%, depending on the size of the cache and number of bits in the tag-store relative to the size of the normal register file, see Chapter 5.

Given the number of load/store instructions, together with the number of spill/fill and register file accesses and the size of the data and the register caches, the following ratio will be used to assess the most energy consuming architecture:

$$Ratio = \frac{E_{Total,RegCache}}{E_{Total,\text{Spill/Fill}}} \qquad \text{(EQ 7.11)}$$

If the ratio is greater than 1.0 the register cache architecture will consume more energy than the spill/fill architecture. Note that the expression favours the spill/fill architecture in that it assumes that the hit-rate in the data cache will be the same for the two architectures. Simulations have shown that the hit-rate in an 8K-bytes data cache drops from 98.5% to 97.1% for the espresso benchmark when spill/fill references are passed to it as well as the load- and store instructions. To simplify the expression the energy consumption of read and write accesses is assumed the same[1]. $E_{Total,RegCache}$ and $E_{Total,Spill/Fill}$ are thus the sum of the energy consumptions in the different levels of the memory hierarchy for the two architectures, based upon the number of accesses to each level. The expression for *Ratio* is therefore:

$$\text{(EQ 7.12)}$$

$$Ratio = \frac{inst + (miss + wback)_{RegCache} \times lsize_{RegCache} \times \left(1 + \frac{E_{Dcache}}{E_{RegCache}}\right) + ldst \times \left(\frac{E_{Dcache}}{E_{RegCache}}\right)}{inst \times \left(\frac{E_{RegFile}}{E_{RegCache}}\right) + spillfill \times 4 \times \left(\frac{E_{RegFile}}{E_{RegCache}} + \frac{E_{Dcache}}{E_{RegCache}}\right) + ldst \times \left(\frac{E_{Dcache}}{E_{RegCache}}\right)}$$

In section 7.3 the optimal size of the register cache was determined to be 128 words (512 bytes). A reasonable data-cache size is 8K byte with lines of 32 bytes. As the energy consumption per request is proportional to the number of ports, see Chapter 5, and the register cache needs 3 ports while the data cache only needs one, the $E_{Dcache}/E_{RegCache}$ ratio can be approximated to:

$$\frac{E_{Dcache}}{E_{RegCache}} \cong \frac{E_{Cache,\,Read,8Kbytes,32\frac{bytes}{line}}}{3 \times E_{Cache,Read,\,512bytes,16\frac{bytes}{line}}} \qquad \text{(EQ 7.13)}$$

1. Chapter 5 has shown the difference.

Inserting the expressions for $E_{Cache,Read}$ from Chapter 5 gives:

$$\frac{E_{Dcache}}{E_{RegCache}} \cong 0.73 \qquad \text{(EQ 7.14)}$$

Equally for $E_{Regfile}/E_{RegCache}$:

$$\frac{E_{RegFile}}{E_{RegCache}} \cong \frac{3 \times E_{RAM,\ Read, 128 bytes,\ 4\frac{bytes}{line}}}{3 \times E_{Cache, Read,\ 512 bytes, 16\frac{bytes}{line}}} \qquad \text{(EQ 7.15)}$$

$$\frac{E_{RegFile}}{E_{RegCache}} \cong 0.29 \qquad \text{(EQ 7.16)}$$

Where the register file is composed of 32 words (lines) each of 4 bytes.

From the four benchmarks selected statistics have been collected, see Table 7.14.

Table 7.14 Program statistics collected for four benchmarks

	espresso	flex	hello	stcompiler
instructions	7,179,096	10,039,395	67,382	2,616,691
ld/st	1,260,241	1,878977	10,115	292,224
spill/fill	129,456	166,642	4,089	146,730
miss + writeback in a 128 word register cache with 4 words per line	346	96	36	1,104

Inserting the results from Table 7.14 into Equation 7.12 above gives:

$$Ratio_{espresso} = \frac{7179096 + 346 \times 4 \times (1 + 0.73) + 1260241 \times 0.73}{7179096 \times 0.29 + 129456 \times 4 \times (0.73 + 0.29) + 1260241 \times 0.73} = 0.94 \qquad \text{(EQ 7.17)}$$

$$Ratio_{flex} = \frac{10039395 + 96 \times 4 \times (1 + 0.73) + 1878977 \times 0.73}{10039395 \times 0.29 + 166642 \times 4 \times (0.73 + 0.29) + 1878977 \times 0.73} = 2.30 \qquad \text{(EQ 7.18)}$$

(EQ 7.19)

$$Ratio_{hello} = \frac{67382 + 36 \times 4 \times (1 + 0.73) + 10115 \times 0.73}{67382 \times 0.29 + 4089 \times 4 \times (0.73 + 0.29) + 10115 \times 0.73} = 0.82$$

(EQ 7.20)

$$Ratio_{stcompiler} = \frac{2616691 + 1104 \times 4 \times (1 + 0.73) + 292224 \times 0.73}{2616691 \times 0.29 + 146730 \times 4 \times (0.73 + 0.29) + 292224 \times 0.73} = 1.81$$

The ratios for stcompiler and flex show that the spill-fill scheme will be less energy consuming for these benchmarks, even if the hit-rate of the first level data cache decreases and the energy consumption of the cache increases correspondingly. For the espresso and hello benchmarks the results indicates that a register cache consumes less energy. However, analysing the expression for ‘Ratio’ the sensitivity to the number of instructions is very high, a small *decrease* in the number of instructions will *increase* the value of ‘Ratio’. As improved versions of the compiler would be expected to decrease the number of instructions considerably the ratio for these benchmarks is expected to increase to beyond 1.0. Consequently the spill-fill scheme is in general less energy consuming than the ‘ajlp’-scheme.

The performance of spill-fill scheme depends much on its implementation. This will be assessed in the following sections.

7.4.3 Implementing the spill/fill scheme

When more registers are required, ‘n’ registers are spilled to memory through ‘spill’ instructions and the remaining registers are renamed to give ‘room’ for the new registers. Similarly, registers can be de-allocated using ‘fill’ instructions. The instructions specify a register containing a source/destination for the contents of the registers coming into scope from or going out of scope. The HORN architecture manual [HORNV5] encourages an implementation to spill and fill to/from a special ‘spill/fill-cache’, which, in many ways, is similar to the register cache described in section 7.3; note however that the ‘active’ set of registers is *not* mapped to memory. A register cache in this architecture is therefore an extra level in the memory hierarchy as illustrated in Figure 7.8

The major difference between the two schemes is that the contents of the new spill/fill cache *must* be coherent with the rest of the memory system. This means that loads from an address used by a spill have to read the data which was spilled, i.e. the contents of the spill/fill cache. Similarly the spill/fill cache needs to be updated if a store instruction writes to an address currently held in the spill/fill cache. Thus the total number of cache accesses to the storage blocks in the architecture increases significantly as two caches[1] need to be accessed for each access.

Load instructions that try to access data which has been spilled, or store instructions that try to write to an address held in the spill/fill cache are expected to be rare; however the hardware needs to be able to handle such a situation as the architecture does not provide ways to synchronize the two caches as was the case in the ‘ajlp’-scheme used in release 3

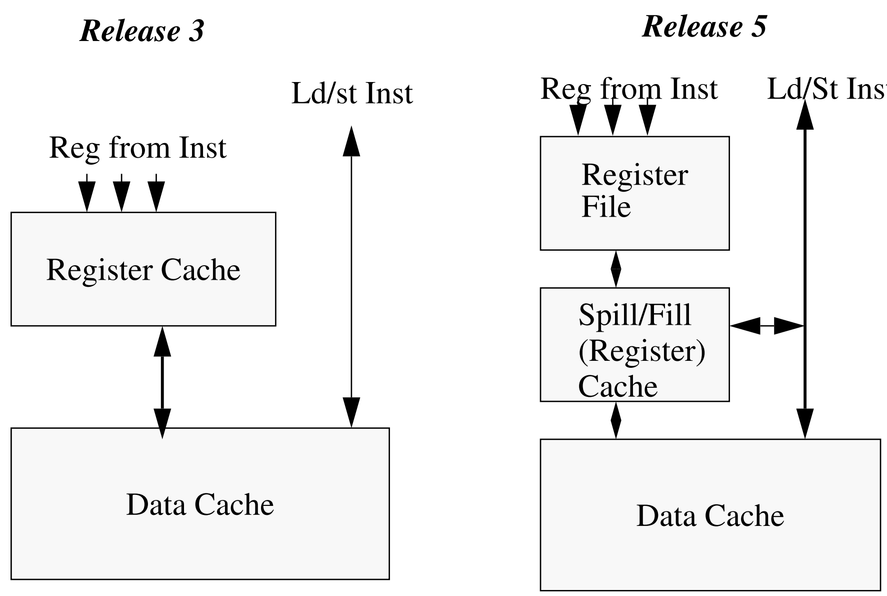


Figure 7.8 Principle difference between Release 3 and Release 5

1. The spill/fill-cache and the data-cache.

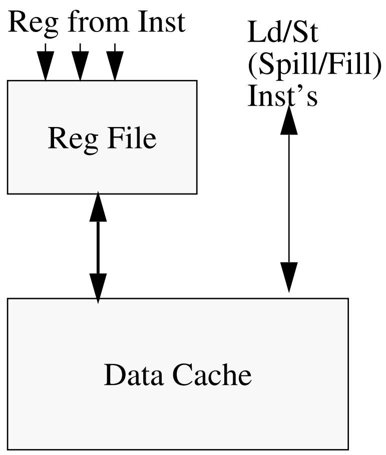


Figure 7.9 Simulated model

of the architecture. The architecture, described in section 7.3, used a special form of the ajlp instruction to ensure coherency. The frequency of that instruction type can be used as an indicator of the need for register-file/memory coherency it suggests that coherency problem is not significant; it will only appear once per benchmark, at initialization.

Despite this, it *is* necessary to maintain coherency between the two caches even though this is not simple to implement. The coherency constraint has been included to make process swapping simpler.

Despite the performance advantages of separating the spill/fill cache and the first level data cache, as shown in Figure 7.8, the following sections will describe two experiments, which assume an implementation without the spill/fill cache, see Figure 7.9. Based on the results from these simulations the value of building a separate spill/fill cache will be extrapolated.

7.4.4 Three ways of implementing the spill/fill scheme

Assuming the architecture presented above, spill and fill instructions can be handled in several ways. This work has explored three options:

- Firstly, a scheme where a spill instruction is converted into four store instructions and an add instruction to update the base register. This implies a severe performance degradation, high CPI, but yields a simple implementation with a minimum of 'special cases' to be considered. Fill instructions are handled in a similar manner; the fill is converted into four load instructions and a subtract instruction.
- Secondly, a scheme employing a spill/fill engine which works in parallel with the normal pipeline. This architecture allows the program execution to continue under some restrictions (to be described later) while the spill/fill goes on in the background. Some extra hardware will be required but the complexity should be low.
- Thirdly, an optimistic and potentially expensive scheme utilises a separate spill/fill-cache which 'catches' all the spill- and fill instructions and spills or fills four registers to or from the spill/fill-cache in one cycle while the base register is updated in parallel.

For the second scheme, problems may occur if precautions are not taken: WaR and RaW hazards [Patt] could occur as well as congestion of the port to the data cache. Furthermore the number of ports to the register file may be a restriction as the spill/fill engine will require a Read/Write port to operate. However, simple restrictions ensure that these hazards do not occur. Instructions from the following list will stall until the spill/fill operation has completed:

1. Any instruction which accesses local registers 0 to 3 during a spill
2. Any instruction which accesses local registers 28 to 31 during a fill
3. Any load or store instruction
4. Any spill or fill instruction

It is obvious that a four cycle penalty (the time it takes to complete the first spill/fill instruction) should be expected when a spill or fill instruction succeeds another.

These three models will be investigated in the following sections.

7.4.5 The cache and memory models

The underlying cache model for all the simulation results to be presented later assumes separate instruction and data caches. The instruction caches are assumed to yield a 100% hit-rate. The data caches are direct-mapped cache organizations with the following parameters:

- Total size
- Line size
- Number of cycles for first fetch from off-chip memory
- Number of cycles for successive fetches from off-chip memory

For off-chip references an assumption about the speed of the processor is necessary as well. For these experiments the cycle time for the processor design is set to 20ns, equivalent to 50MHz.

A data-sheet for a typical 4Mx1 Toshiba DRAM[1] [TOSHIBA] provides the following:

- Random access: 70ns
- Sequential access: 40ns[2]
- Recovery: 60ns

Conservatively the '*#cycles for first fetch from off-chip memory*' parameter is set to 4 cycles, the '*#cycles for recovery following memory reference*' to 2 cycles and the *'#Cycles for recovery following off-chip references'* to 3 cycles. These numbers will remain the same for all the simulations.

1. TC514100ASJ/AZ/AFT70
2. Page Access mode.

7.4.6 Results

For each of the models described, simulations have been run for the following data cache configurations:

- Total size: 4K bytes, 8K bytes, 16K bytes
- Line size: 32 bytes, 64 bytes, 128 bytes
- Latency for first fetch from memory: 4 cycles
- Latency for successive fetches: 2 cycles
- Recovery: 3 cycles

7.4.6.1 Model 1, A conservative scheme

In this model fill and spill instructions are converted into five instructions: four load or store instructions and an update of the offset into the register file.

The results collected from espresso, flex, stcompiler and cacti are presented in the tables below. As a model where only one register can be spilled/filled per cycle is assumed, there is a lower limit for the system performance:

$$CPI_{Ideal} = \frac{Inst + (spill + fill) \times 4}{Inst} = 1 + \frac{(spill + fill) \times 4}{Inst} \qquad \text{(EQ 7.21)}$$

This yields an optimal CPI of 1.048[1] for cacti, 1.05 for espresso, 1.17 for flex and 1.22 for stcompiler. These values have to be compared against the CPI values shown in Tables 7.15 - 7.18 below, which show that a 4K-byte or a 8K-byte data cache with 32-byte lines performs very well, i.e close to CPI_{ideal}.

Figure 7.10 summarises the relation between cache size and CPI for a cache line of 32 bytes.

1. See Table 7.19 for the number of instructions and the number of spill/fill.

Table 7.15 Data cache simulations, cacti - CPI_{ideal}=1.048

Cache size	2K bytes		4K bytes		8K bytes		16K bytes		32K bytes	
Line size [bytes]	Hit Rate [%]	CPI	Hit Rate [%]	CPI	Hit Rate [%]	CPI	Hit Rate [%]	CPI	Hit Rate [%]	CPI
32	99.8	1.049	100	1.049	100	1.049	100	1.049	100	1.049
64	99.8	1.049	100	1.049	100	1.049	100	1.049	100	1.049
128	99.0	1.055	99.2	1.052	99.2	1.052	100	1.049	100	1.049

Table 7.16 Data cache simulations, espresso - CPI_{ideal}=1.05

Cache size	2K bytes		4K bytes		8K bytes		16K bytes		32K bytes	
Line size [bytes]	Hit Rate [%]	CPI	Hit Rate [%]	CPI	Hit Rate [%]	CPI	Hit Rate [%]	CPI	Hit Rate [%]	CPI
32	93.6	1.147	95.6	1.115	97.6	1.084	99.4	1.058	100	1.057
64	87.2	1.463	94.2	1.169	96.8	1.107	99.5	1.060	100	1.050
128	83.8	2.258	91.9	1.492	95.6	1.247	99.2	1.073	100	1.050

Table 7.17 Data cache simulations, flex - CPI_{ideal}=1.17

Cache size	2K bytes		4K bytes		8K bytes		16K bytes		32K bytes	
Line size [bytes]	Hit Rate [%]	CPI	Hit Rate [%]	CPI	Hit Rate [%]	CPI	Hit Rate [%]	CPI	Hit Rate [%]	CPI
32	94.0	1.297	95.8	1.256	97.6	1.223	99.3	1.189	99.7	1.181
64	93.7	1.397	95.6	1.309	97.7	1.235	99.2	1.194	99.7	1.183
128	91.8	1.788	94.1	1.537	97.6	1.233	99.1	1.221	99.7	1.191

Table 7.18 Data cache simulations, stcompiler - CPI_{ideal}=1.22

Cache size	2K bytes		4K bytes		8K bytes		16K bytes		32K bytes	
Line size [bytes]	Hit Rate [%]	CPI	Hit Rate [%]	CPI	Hit Rate [%]	CPI	Hit Rate [%]	CPI	Hit Rate [%]	CPI
32	93.5	1.342	95.4	1.308	98.4	1.255	99.2	1.224	99.4	1.219
64	92.9	1.418	94.6	1.366	98.1	1.268	99.2	1.246	99.4	1.242
128	90.9	1.831	93.4	1.646	97.6	1.337	99.0	1.287	99.2	1.280

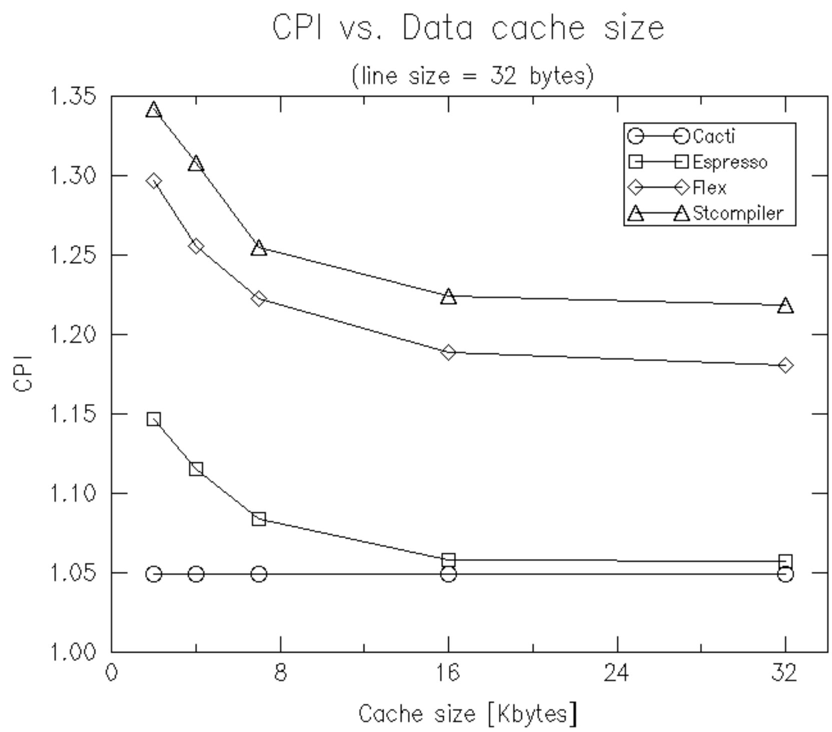


Figure 7.10 CPI vs. Cache size, Model 1

7.4.6.2 Model 2, A spill/fill engine

The performance of the architecture employing a spill/fill engine is given by the distance between the spill/fill instruction and the first instruction mentioned in the list in section 7.4.4. The shorter the distance, the more cycles the 'normal' instruction flow has

Table 7.19 Distance between spill/fill and first ld/st/spill/fill

	Distribution [%]			
Distance	**cacti (18.9MI/ 0.23MSF[a])**	**espresso (4.8MI/ 0.06MSF)**	**flex (10.7MI/ 0.46MSF)**	**stcompiler (1.87MI/ 0.11MSF)**
1	11.0	36.4	45.0	25.9
2	1.0	9.9	21.0	10.2
3	1.8	5.6	1.7	16.6
4	1.3	22.4	4.2	9.2
5	2.9	6.9	8.0	3.5
>5	82.0	18.8	20.1	34.6
CPI_{ideal}	1.001	1.03	1.11	1.10
$CPI_{16K,32bytes/line}$	1.001	1.059	1.153	1.115

a. Format: MI: Millions of Instructions, MSF: Millions of Spills and Fills to be stalled. Table 7.19 shows the distribution of distances between a spill/fill instruction and the instructions mentioned earlier. Distance 1 means successive spill/fill instructions or a spill/fill immediately followed by a load or a store instruction. As one of the assumptions is that only one spill or fill can be handled at a time, this class of sequences imply the full penalty, four cycles, is taken by the first of any two consecutive spill or fill instructions. It is thereby assumed that the register update will take place in parallel with one of the spills/fills. In the case of Distance = 2 the penalty is reduced to three cycles. Distance = 3 implies a penalty of two cycles etc. Assuming a 100% hit-rate in the data-cache and based on these numbers the ideal CPI-values can be calculated as:

$$CPI_{ideal} = 1 + P(Spillfill)\,(P(1) \times 4 + P(2) \times 3 + P(3) \times 2 + P(4)) \quad \text{(EQ 7.22)}$$

The values for CPI_{ideal} for the four benchmarks are shown in Table 7.19 together with the CPI values obtained with a 16K-byte data cache using the model discussed in the previous section. The results shows an improvement for most of the benchmarks compared to the scheme presented in section 7.4.6.1. These numbers do not allow for the normal ‘loss’ in performance due to the instruction cache having a hit-rate of less than 100%. From Table 7.19 it can be seen that data-cache misses from loads and stores alone degrade the performance by only 0.1% for Cacti, 3% for Espresso, 11% for Flex and 10% for stcompiler.

The number and the sequence of requests to the instruction cache will remain the same as under model 1, so it can be assumed that the same hit-rate can/will be obtained. Based on this assumption and the results from model 1; a better estimate for the CPI under this model can be calculated. The cycle count will be less than under model 1 dependent on the distance from a spill/fill instruction to the next memory referencing instruction:

$$CPI = CPI_{model1} - \frac{nbSpillFill}{nbInst} \times (P(2) + 2 \times P(3) + 3 \times P(4) + 4 \times P(\geq 5)) \quad \text{(EQ 7.23)}$$

The results for a 16K-byte data cache with 32-byte cache lines are shown in Table 7.19.

As compiler technology improves, it is expected that the number of spills and fills will decrease. In particular a significant decrease in cases where multiple spills/fills are required, due to better register allocation techniques would be expected. That will reduce the “Distance = 1” percentage significantly and thereby increase the value of a separate Spill/Fill Engine.

7.4.6.3 Model 3, A spill/fill cache

A third implementation of the architecture has been modelled so that the effect of a separate spill/fill cache can also be observed. This model assumes that all spills and fills are passed to a separate spill/fill-cache which yields a 100% hit-rate and which is special in that 4 words can be read from/written to simultaneously. These assumptions may seem optimistic or unjustified, but simulations have indicated that such a cache does not need to be very large (256 words) to be ‘self-sufficient’ after compulsory misses have been met and should therefore not put pressure on the rest of the memory hierarchy. Furthermore, it has been observed that a load or store instruction never accesses the addresses kept in the spill/fill-cache and vice versa.Table 7.20 shows the results of running simulations and

Table 7.20 Hit-rate in data cache[a] and CPI assuming a separate spill/fill cache

Cache Size [bytes]	cacti 18.9MI/1.1MR		espresso 4.8MI/ 1.4MR		flex 10.7MI/2.5MR		stcompiler 1.87MI/0.3MR	
	Hit Rate [%]	CPI	Hit Rate [%]	CPI	Hit Rate [%]	CPI	Hit Rate [%]	CPI
2K	99.8	1.001	93.9	1.079	92.8	1.082	92.3	1.054
4K	99.9	1.000	95.8	1.053	95.4	1.049	95.1	1.033
8K	100.0	1.000	97.4	1.032	97.5	1.025	97.0	1.020
16K	100.0	1.000	99.5	1.007	99.0	1.009	98.2	1.012
32K	100.0	1.000	99.9	1.001	99.5	1.004	98.8	1.008

a. **Line size = 32 bytes**

not passing the spill/fill entries to the D-cache. Associated with the name of the benchmark are two numbers ‘x’MI / ‘y’MR, where ‘MI’ means Millions of Instructions and ‘MR’ means Millions of data cache References. As it shown, espresso is the most memory intensive program in that the ratio ‘x/y’ is only 3 while it is 17 for cacti. Note that MR has nothing to do with ‘MSF’ mentioned in Table 7.19. The number of references to the data cache is significantly lower using this model than under the two other models explored above, as the spill/fill references have been removed. Figure 7.11 presents the data from Table 7.20 in a graphical form.

Although the optimal data-cache size is still 16K bytes (as it was for Models 1&2), a comparable performance can be obtained with a much smaller data-cache, for example 4K bytes. If the assumptions about a ‘close to’ 100% hit-rate in the spill/fill-cache and no coherency problems hold, then Model 3 is therefore more energy efficient than any of the others given a fixed performance requirement.

It may therefore be desirable to build a 4K bytes data-cache and a 1K bytes spill/fill-cache, which ‘in total’ is smaller than one 16K bytes data-cache serving all the types of

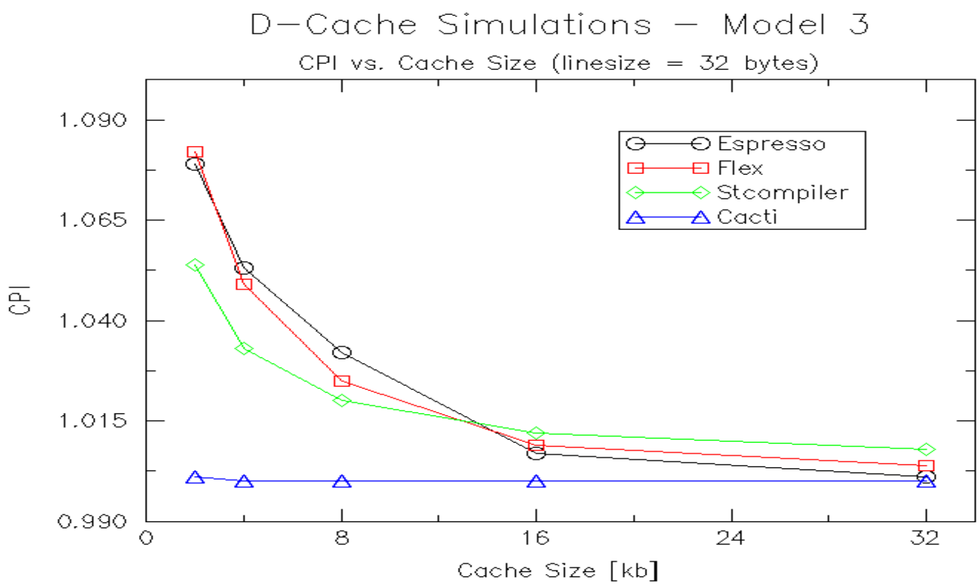


Figure 7.11 CPI vs. data cache size, 32 bytes/line, Model 3

memory requests. The disadvantage of this is that, although coherency problems have never been encountered, hardware *has* to be provided to detect and handle such problems, for the reasons listed in sections 7.4.6.1 and 7.4.6.2.

To obtain the highest possible performance, a special spill/fill cache needs to be able to handle requests for 4 words in one cycle. This is relatively simple to implement as one cache line could simply be 4 words wide (16 bytes) making the access to the cache very simple and fast. Assuming that the data is aligned on a 16-byte boundary, the data cache can be accessed through a block-buffer, see section 5.8 on page 100 and Chapter 9.

As for energy consumption, a separate spill/fill-cache is likely to address only a very restricted area of the total address space. Many of the tags present in the cache are therefore likely to be the same. Chapter 5 has suggested cache architectures, sectored caching and CAT-caching, which will reduce the energy consumption of such caches by exploiting the high degree of locality.

7.4.7 Summary

The spill/fill architecture replaced the ajlp-architecture to make the porting of operating systems easier. Section 7.4 has examined the spill/fill architecture and compared it to the ajlp-architecture described in the previous section. The spill/fill scheme does not perform as well as the ajlp-scheme due to the increased number of instructions and the increased number of accesses to the 1st-level data cache. Furthermore, it is in general more energy consuming than the 'ajlp'-architecture, see section 7.4.2. Consequently is it not as energy efficient as the ajlp-scheme either. Section 7.4.6.3 presented the most energy-efficient way of implementing the spill/fill scheme, a separate spill/fill cache.

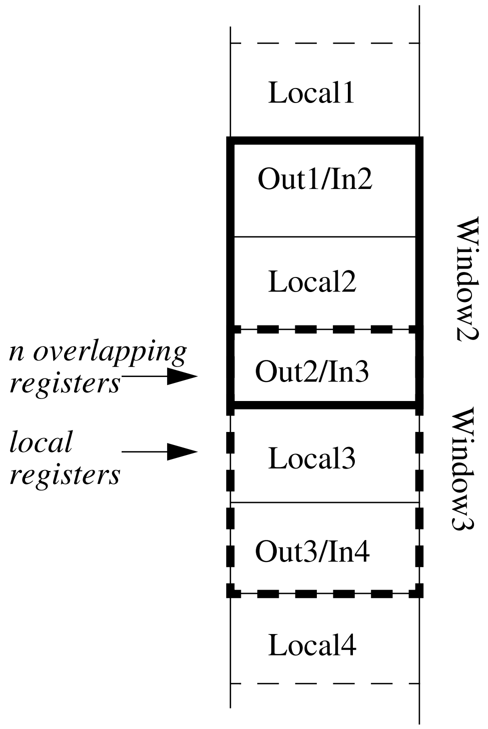


Figure 7.12 Principle of overlapping register windows in SPARC

7.5 Register windows (SPARC)

The architectures described above are in some ways equivalent to that of SPARC [Weaver]. The SPARC architecture specifies eight global registers and 24 local registers arranged in an overlapping configuration. A SPARC register file might contain many register windows, but only one will be active at any given instant during execution. Changing a register window is a side-effect of calling/returning-from a subroutine. The configuration is shown in Figure 7.12. An implementation of the SPARC architecture is free to implement as many register windows as desired for the performance target. If a program requires more register windows than available, the operating system is invoked and more windows are allocated by spilling early register windows to memory.

This means that if a program has a deep call tree, there is a high probability that the operating system needs to be invoked regularly to allocate more register windows and similarly later to restore them. Invoking the operating system is time consuming as it typically includes saving and restoring some of the state of the processor, implying a degradation of performance, and consequently represents an overhead to the execution time. To assess the timing implications of invoking the operating system the following program was written and run on a SPARC-5 workstation:

```
void proc(int N)
{
    if (N > 0)
        proc(N-1);
}
void main(int argc, char *argv[])
{
    int index;
    for (index = 0; index < 10000; index++)
        proc(atoi(argv[1]));
}
```

Figure 7.13 shows that the execution time does not increase significantly as the recursion depth increases from one to five but there is a significant overhead going from a depth of five to six and beyond. This indicates that the SPARC-5 has 7 overlapping windows; indicating a total of 128 registers.[1]

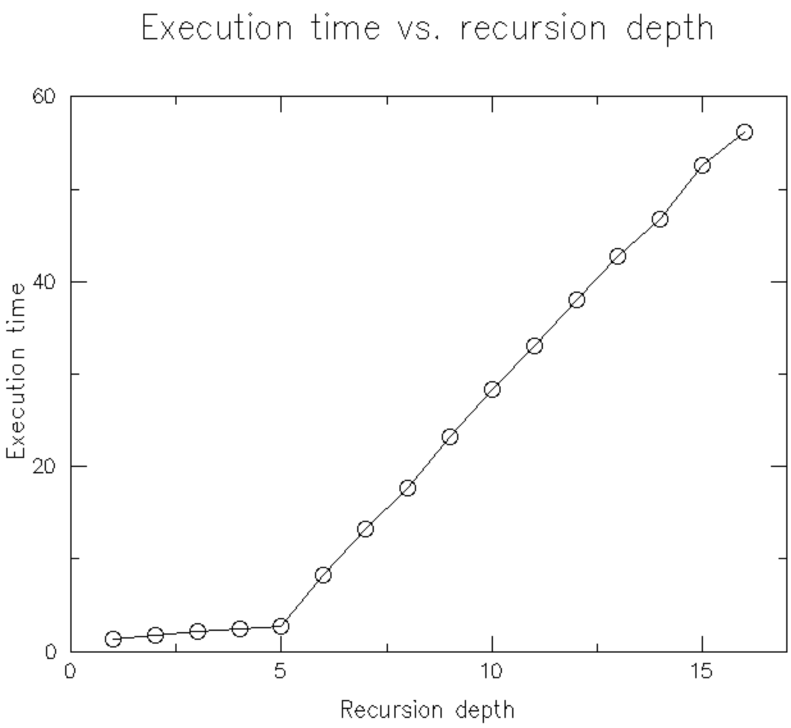


Figure 7.13 Execution time versus recursion depth on a SPARC station 5

1. 8 global registers + “number of windows” * 16 registers + 8 registers.

An approach to minimize this overhead is to let the compiler know how many register windows are available; it may be able to in-line function calls and thereby minimize the risk of overflowing the register file.

Comparing the energy consumption of a system with a 128-word register file and a system with a 128-word register cache is not simple. The 128-word register file should be less energy consuming given that it contains less storage[1]. In terms of access time the two are comparable[2]. However, the register window scheme is clearly not as flexible as the two schemes presented in sections 7.3 and 7.4 and is therefore expected to cause more memory traffic implying higher energy consumption.

The earlier schemes allow a finer grain use of the register file since register windows can overlap by multiples of four registers, whereas the SPARC architecture specifies a fixed overlap of eight registers. [Mulder] shows that for performance the optimal organization of the overlapping register file with 32 active registers is 8 global, 8 local and 8 overlapping registers (see Figure 7.12) as implemented in SPARC. [Mulder] shows that a performance penalty of 15% is encountered if the configuration is changed to 8 global, 16 local and 4 overlapping registers, due to the extra load/store instructions which would be required to save and restore registers. However, that work assumed a fixed organization with a fixed number of overlapping registers; i.e. a scheme which is not as flexible as the HORN scheme, described in section 7.3.

The fact that register spilling/filling is a side effect of the call/return instructions in the SPARC architecture has a dramatic impact on the number of instructions to be issued and thereby on the performance of the system. Chapter 5 has shown that calling a subroutine

1. Register file: 128 registers each 32 bit: = 4096 bits
Register cache: 128 registers each 32 bits + 16 tags each 24 bits = 4480 bits
2. The access time to the cache is dominated by the access time to the data storage rather than the tag storage.

requires two instructions in the HORN-architecture while it requires only one instruction in the SPARC instruction set. Furthermore the HORN architecture would normally insert one or two spill or fill instructions to (de-)allocate registers. In total this means that the HORN-architecture needs to execute four instructions for the same functionality as the 'call'- or 'return'-instructions in a SPARC-architecture.

The general expression for this overhead is:

$$Overhead = \frac{\#\text{spill} + \#\text{fill} + \#\text{leaplink}}{\#\text{instructions}} \times 100\% \qquad \text{(EQ 7.24)}$$

Where '#spill' and '#fill' denote the number of spill- and fill-instructions in the code. '#leaplink' is the number of 'calls-to' plus the number of 'returns-from' subroutines/ functions.

Note that with every leaplink-instruction there is a 'go'-class instruction; the overhead, in terms of instruction count, is 'number-of-leaplink instructions'. Spill and fill instructions are pure overhead as they would be side effects of a 'call' or 'rtn' instruction in the SPARC instruction set.

Table 7.21 shows the instruction overhead of the HORN-architecture scheme compared to the 'conventional' SPARC architecture scheme described above. As the table shows there is a significant, 5.45%, instruction overhead associated with the HORN-register file architecture. This overhead will translate into decreasing performance. However, it is not possible to quantify this decrease as it will depend on whether the 'number of register windows' contained in the SPARC register file is sufficient, or if the operating system needs to be invoked as described above.

If the operating system does *not* need to be invoked, the instruction count overhead calculated in Table 7.21 will translate directly into a performance degradation. However,

Table 7.21 Instruction overhead with the spill/fill scheme

	cacti	dhry	espresso	flex	hello	stcompiler	Aver
Instructions[a]	18,921,879	686,059	4,703,874	10,685,992	51,366	1,873,152	
#Call/Return	277,473	13,490	38,687	145,110	952	51,327	
#Spill+#Fill	262,557	27,971	65,692	473,075	3,013	110,064	
Overhead [%]	2.85	5.50	2.22	5.79	7.72	8.62	5.45

a. Note that these instruction count and spill/fill numbers are significantly lower that those presented in Table 7.12 and elsewhere in section 7.4. This is due to compiler technology improvements between the releases for version 3 and version 5 of the HORN architecture.

if there is an insufficient number of register windows and the operating system does need to be invoked, it carries a significant penalty. The gradients of the two parts of the curve in Figure 7.13 show that it takes approximately 18 times longer to allocate registers once all the register windows have been used. The number and frequency of register file overflows varies from benchmark to benchmark.

Statistics collected with the SHADE tools [Shade], see Table 7.22, show that the number of register file overflows is small for the benchmarks examined. The implication of the increased time to allocate registers will have very little impact on the overall execution time.

Table 7.22 Overflows in SPARC register file

	cacti	dhry	espresso	fft	flex	hello
Instructions	14,291,743	8,740,025	5,543,933	42,586,229	5,967,239	242,555
Call/jmpl	508,671	673,218	99,899	772,648	159,851	16,374
Overflows	120	55	274	58	74	28
$\frac{\text{Overflow}}{\text{Call/jmpl}} \times 100\%$	**0.02**	**0.01**	**0.27**	**0.01**	**0.05**	**0.17**

For applications where constant throughput is important it is clear that the SPARC architecture has disadvantages for scalable problems where the number of overflows increases with the parameters. The spill/fill architecture might therefore be desirable when constant throughput is more important than peak-performance as a given problem scales.

7.6 Summary

This chapter has presented a number of register file architectures and evaluated them in an energy efficiency perspective. The study has shown that a register cache is very energy efficient and dependent on the compiler may also outperform than the alternative, spill/fill architecture.

Section 7.2 showed how extra local storage, organized as a queue, improves both performance and energy efficiency.

Section 7.3 presented a register file architecture based on the ‘ajlp’ scheme. The results indicated that a very small cache (128 words if optimizing for performance, 64 words if optimizing for energy efficiency) ensures almost no traffic between the register cache and the first level data cache. However, each register reference consumes much more energy under this scheme than if the instructions accessed a conventional 32 word register file.

Section 7.4 presented an alternative to the ajlp-architecture based on two instructions: spill and fill. The section proposed three ways of implementing the spill/fill mechanisms. Model 2, the spill-fill engine yields a significant performance increase compared to the conservative scheme presented as Model 1, but without the complexity overhead required by Model 3. Also, Model 2 has the advantage that improvements in the compiler technology will almost certainly improve the performance, which is not necessarily the case for Model 3.

It is therefore not obvious which model is the most energy efficient. It will be necessary to evaluate the cost in terms of the extra hardware required to implement Model 2 and as mentioned, it is very likely that an extra port on the register file will be required. A slightly larger register file (32 + 4) might be desirable as it will allow the spilling to be done in the background. The resulting configuration may consume as much energy as the separate Spill/Fill cache presented as Model 3.

Section 7.4.2 presented results indicating that the spill/fill architecture is not more energy consuming than a register cache. However, the register cache yields the best performance. Overall, the energy efficiency of the register cache is better than the architectures presented in section 7.4.

The energy efficiency of the SPARC style architecture described in section 7.5 is very dependent on the ‘performance’ of the compiler. If the compiler can keep the number of ‘window’ overflows at a minimum, the data cache need not be accessed very often, and the energy consumption of the register file is therefore comparable to that of the register cache presented in section 7.3. Given the smaller semantic content of the HORN-instruction set, the instruction count, and thereby the execution time, is favouring the SPARC architecture; so the energy efficiency of the SPARC-architecture might be better than that of the HORN-architecture. However, if the SPARC register file ‘overflows’ the performance penalty is more severe than that for any of the HORN architectures as a trap would normally be generated and the operating system invoked. In those cases the SPARC register file is *not* considered as energy efficient as the two architectures proposed in sections 7.3 and 7.4. Table 7.13 showed that register file overflows do not happen very frequently.

Chapter 8 Instruction fetching

When specifying a processor architecture one of the most fundamental decisions which must be taken is the instruction set architecture. Following the introduction of the RISC concept in the early 1980's [Patt], processor designs have been classified as either RISC or CISC, see e.g. [Robin]. This chapter describes existing instruction formats and evaluates the effect the variable-size instruction format described in the HORN architecture has on the performance of the instruction cache. Furthermore the chapter presents three instruction fetch mechanisms which address the issue of instructions straddling cache lines and thus improves the energy efficiency of the instruction cache.

Section 8.1 is an introduction to instruction formats in existing processors and their implications for the rest of the processor architecture. The section describes how instructions can straddle cache lines in CISC processors and how some Multi-Instruction-Issue RISC processor architectures have been defined to avoid multiple cache accesses to assemble instruction packets.

Section 8.2 presents statistics collected as a part of this study which justify the use of variable-size instructions as a way to improve both performance and energy efficiency of the instruction cache and hence of the processor system.

Section 8.3 presents three cache architectures which improve both the performance and energy efficiency of a RISC design with variable-size instructions. The novel architecture described in section 8.3.3 provides a simple, energy efficient way of virtually eliminating the performance penalties associated with variable-size instructions.

8.1 Introduction

RISC instruction sets are characterized by few instruction formats. Operands for arithmetic and logical instructions are always kept in registers while only load and store instructions access data in main memory. CISC instruction sets, however, typically specify numerous instruction formats and operands for instructions may be fetched from the register file or from memory.

These differences imply that the semantic contents of the instructions in the two classes of architectures are very different. Consequently, a RISC program will typically contain more instructions than the corresponding CISC program.

There are processor architectures which are not easy to classify as either RISC or CISC. For example, the Hobbit architecture [Avgade] retains most characteristics of a RISC, but specifies variable-size instructions, where the size is dictated by the semantic content of the instruction. In contrast, the Intel i960 [Wharton] specifies fixed size 4-byte instructions[1], but retains all the characteristics of a CISC: a large number of instructions, addressing modes and primitive data types.

Due to their simple instruction formats RISC architectures are typically simpler to implement than CISC architectures. It is especially simple to pipeline an implementation of a RISC architecture, while it may be more complicated for CISC. Consequently, RISCs typically have shorter cycle times and thus yield completion times which are comparable or shorter than those of CISC processors, despite a higher number of instructions [Johnson]. Furthermore, the performance of CISC processors does not increase as fast with the semiconductor technology as that of RISC designs [Segar].

1. Certain instructions may specify a second word containing constants

In RISCs the single instruction size is typically the size of the data bus. Simple encoding makes the decoding simple and modular. The uniform instruction size makes the size of programs relatively large; even simple instructions such as ‘mov R1,#1’ will require the same storage as more complicated instructions such as ‘add R1,R2,R3’.

In contrast, CISC instructions are usually of variable sizes. Instruction size depends on the semantic content and varies significantly. For example, the 68000 instructions [Robin] vary in size from 1 to 5 words[1] requiring multiple accesses to the memory hierarchy before an entire instruction can be composed.

CISC processors often make use of micro programming, where the conventional user-provided codes are re-coded into sequences of micro-instructions. Some implementations pipeline the execution of the microcode [Johnson], allowing the fetching of the instruction to be broken into multiple cycles and the words for a given instruction are therefore only fetched when required. This can result in long pipelines and very high CPI values as the next instruction can be issued only once it is clear where it starts; i.e. when the size of the previous instruction has been determined.

Implementations of modern 64-bit RISC architectures such as the DEC Alpha and the IBM PowerPC601 [Case2] issue multiple instructions simultaneously. The DEC Alpha architecture also specifies that instruction packets must be aligned at 8-byte boundaries for an implementation to yield the optimal performance. Traps are generated if an instruction packet is not properly aligned [DEC21064] thus, although valid programs might be written composed of instruction packets which straddle cache lines, the performance of such programs is significantly reduced relative to an equivalent program with properly aligned instruction packets. In ‘well-assembled’ programs, instruction

1. a word being defined as the size of the data bus.

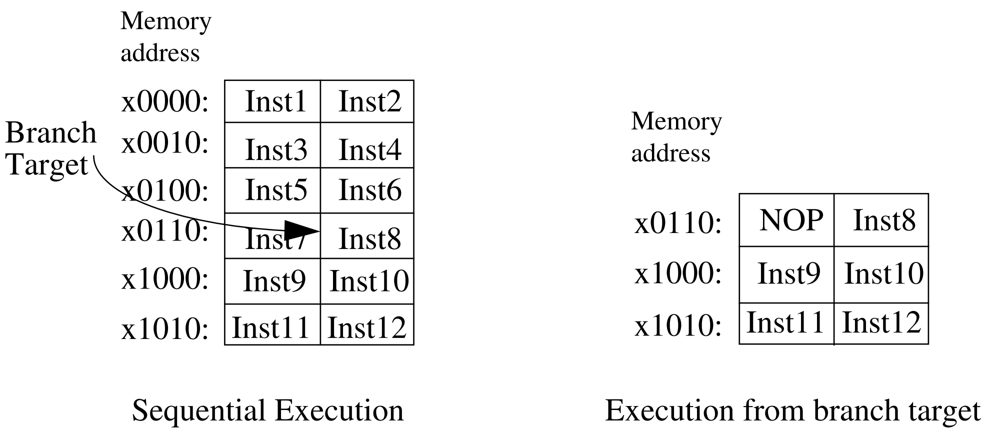


Figure 8.1 Branch to a non-aligned instruction

packets to be issued do not straddle cache line boundaries. Branch instructions may target an instruction in the middle of a packet. This will cause the entire word-aligned packet of instructions to be fetched, but only the instruction following the branch target will be issued. Consequently the following packet of instructions is properly aligned, see Figure 8.1. Similarly ARM has introduced the Thumb-format [Segar] where two 16-bit instructions are packed into a 32-bit word. A Thumb code fragment is guaranteed to contain an even number of instructions.

The HORN-architecture, which forms the basis of the work described in this thesis, cannot easily be characterized as either RISC or CISC. It retains most of the characteristics of a simple RISC in that all operations operate on registers; other parts of the memory hierarchy are only accessed through dedicated load/store-type instructions. Conversely, the architecture differs from a traditional RISC architecture in that its instructions have variable sizes, see Chapter 3. Instructions can be 1, 2, 3 or 4 bytes i.e *less than or equal to* the size of the 32 bit[1] data bus. There are no constraints on how instructions are aligned with respect to word boundaries; instructions may therefore

1. The HORN architecture specifies a 64 bit databus, but as discussed in Chapter 3 this thesis considers it a 32-bit processor

straddle cache lines. Furthermore, as the size of the opcode field varies between 6 and 10 bits, even the opcode may be split across two cachelines.

Instructions which straddle cache lines are not a significant problem for the Hobbit processor [Avgade], where instructions can be of 2, 6 or 10 bytes. More than 80% [Slater] of instructions are of the smallest, 2-byte, format which can always be fetched in one cycle, given any sensible cache line size. Furthermore, the operands for the larger instructions are typically fetched from memory. Due to the sequential process of fetching the data operands for these larger instructions their performance is reduced. The importance of fetching a full instruction per cycle is reduced as well [Avgade].

8.2 Variable-size instructions in the HORN architecture

The HORN instruction format differs from that normally associated with a RISC architecture. There are few restrictions on the size or ordering of instructions and an instruction may therefore straddle cache lines. For example, the first byte of an instruction may be in one cache line while the remaining bytes are in another. This would normally require two accesses to the instruction cache and a cross-bar network to compose the instruction, with a consequent negative effect on both performance and energy consumption. The number of instructions which would require two cache accesses is a function of the instruction sizes and of the length of the instruction cache lines. The architecture specifies that the variable-size instructions should not affect the instruction flow/performance.

Table 8.1 shows the average instruction size for the programs in the benchmark suite. The average instruction size is approximately 3 bytes implying 10 instructions per cache line of size 32 bytes. Hence every 10th or 11th[1] instruction may require two cache lookups i.e a 10% degradation in performance relative to a format where such problems do not arise.

Table 8.1 Average instruction sizes for the benchmarks

Benchmark	Average instruction size [bytes]
cacti	3.45
dhry	3.15
espresso	2.93
fft	3.10
flex	3.06
hello	3.11
stcompiler	3.05
Average	**3.12**

The results in Table 8.2, however, show that in practice instructions straddle cache lines less frequently. The results for a 32-byte cache line show that, on average, only 6.65% of instructions cross cache line boundaries. This difference is due to short basic blocks: non-broken sequences of instructions which branch before crossing the cache line boundary. Only cacti is significantly different; as explained in Chapter 4, this benchmark is characterized by very long basic blocks and is therefore expected to encounter the problem more frequently.

Table 8.2 shows that the frequency of instructions straddling cache lines is very sensitive to the size of the cache line. This suggests that cache line should be made as long as possible. However, Chapter 5 has shown that the energy consumption of a cache is also sensitive to the line size. There is a trade-off to be made.

Taking an energy efficiency perspective, the 'double accesses' are doubly degrading in that they both increase the number of energy-consuming cache accesses, and also reduce performance as the number of cycles for a program to execute is increased with the

1. corresponding to 10% or 9% of all instructions.

Table 8.2 Percentage of instructions which straddle cache line boundaries

	Cache line size		
Benchmark	**16 bytes**	**32 bytes**	**64 bytes**
cacti	15.0	8.94	4.62
dhry	14.4	6.92	3.02
espresso	13.4	6.57	3.36
fft	12.8	5.50	2.13
flex	12.0	6.71	3.40
hello	13.6	5.72	3.40
stcompiler	12.5	6.20	2.29
Average	**13.4**	**6.65**	**3.17**

resulting effect on energy consumption. The EE of the cache design is therefore significantly lower than if instructions could not straddle cache line boundaries, 12% lower for a 32-byte cache line:

$$EE = \frac{1}{E_{Total} \times time} = \frac{1}{(1.0665 E_{Total,0}) \times (1.0665 time_0)} = 0.88 EE_0 \qquad \text{(EQ 8.1)}$$

As the cache-line size increases, the miss rate decreases but past a certain point the decrease in miss-rate is insufficient to compensate for the increase in I/O traffic caused by the longer cache lines. Furthermore, the energy consumption per access increases with the line size, see Chapter 5. Choosing long cache lines merely because they reduce the problem of instructions straddling cache line boundaries may not be an energy efficient strategy.

The reasoning above suggests that the use of variable-size instructions is not an energy efficient strategy. Table 8.3, however, shows that the miss-rate in the instruction cache is reduced, and the performance consequently improved, when a program is compiled for variable-size instructions rather than for the conventional 4-byte RISC style. For six of the benchmarks the variable-size instructions reduce the overall miss-rate in the instruction cache despite increasing the number of references to the cache caused by instructions

Table 8.3 Instruction cache miss rate for 4 byte- and variable-size instructions.

	Miss Rate[a] [%]		Improvement
Benchmark	4 byte Inst	Var Size Inst	[%]
cacti	1.9	1.4	26.3
dhry	0.88	1.3	-80.7
espresso	0.94	0.77	13.5
fft	0.60	0.33	45.0
flex	0.84	0.35	58.3
hello	4.20	2.94	30.0
stcompiler	3.25	1.64	49.5

a. **Cache parameters:** 8K bytes, 32-byte cache lines, 2-way set-associative, random replacement.

which straddle cache lines. For the dhry benchmark, however, the increase in instruction cache references and the cache reference pattern change so much that the miss-rate for the 4-byte instruction format is lower than that for the variable-size instruction format.

Eliminating or reducing the penalties of instructions straddling cache lines will lead to an improvement in energy efficiency due to reduced execution time. The following section will describe architectures which reduce the performance penalty significantly *and* reduce the number of cache accesses; i.e. architectures which will reduce both the energy consumption and the execution time; thus improve the energy efficiency of the entire processor system.

8.3 Instruction fetch mechanisms

Section 8.2 showed that reducing the size of instructions improves the hit rate of the instruction cache, but Table 8.2 and Equation 8.1 showed that the number of cache references increases and the energy efficiency decreases due to the double accesses that are required for the instructions which straddle cache lines.

Furthermore it is important to remember that, although the instances of instructions straddling cache lines may be infrequent (for long cache lines), the implementation must accommodate this type of access. Three instruction cache architectures have therefore been proposed and analyzed to reduce or eliminate the disadvantages of variable-size instruction while retaining the advantages of improved hit rate.

8.3.1 The alignment architecture

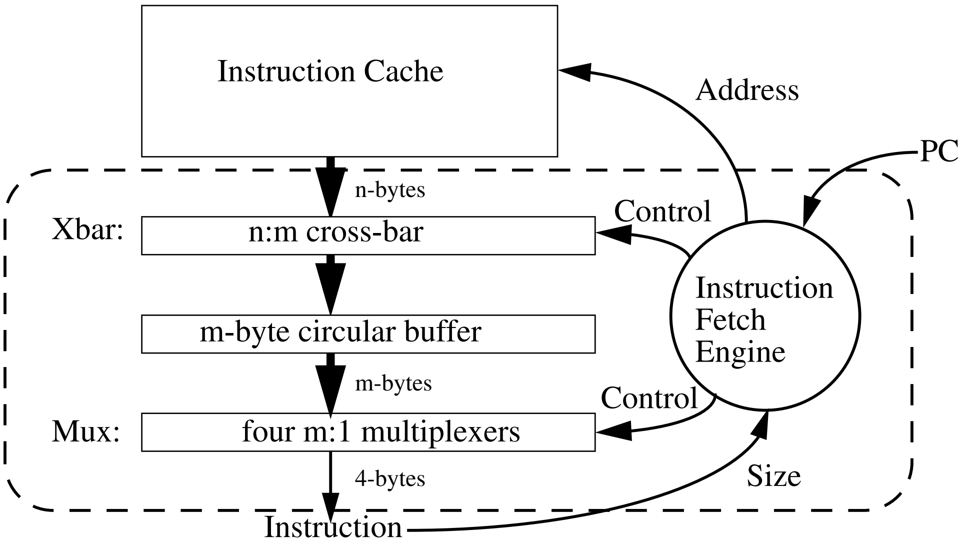


Figure 8.2 The alignment architecture

The first architecture explored is shown in Figure 8.2, here n-bytes are fetched into a m-byte circular buffer, where m > n. The n:m cross-bar facilitates the insertion of n-bytes from the cache line into any position in the circular buffer. The instruction fetch engine then fetches instructions from the head of the circular buffer through a head-pointer; while keeping track of the size of instructions it can fetch complete instructions from any position in the buffer. A tail-pointer indicates the last valid byte in the buffer. When the difference between the head-pointer and the tail-pointer is ‘small’[1] the succeeding n-bytes should be fetched from the cache and inserted into the buffer at the position specified by the tail pointer. This will overwrite some parts of data from preceding words dependent on the values of ‘m’ and ‘n’, see Figure 8.3. Besides providing a solution to the issue of

1. to be quantified later

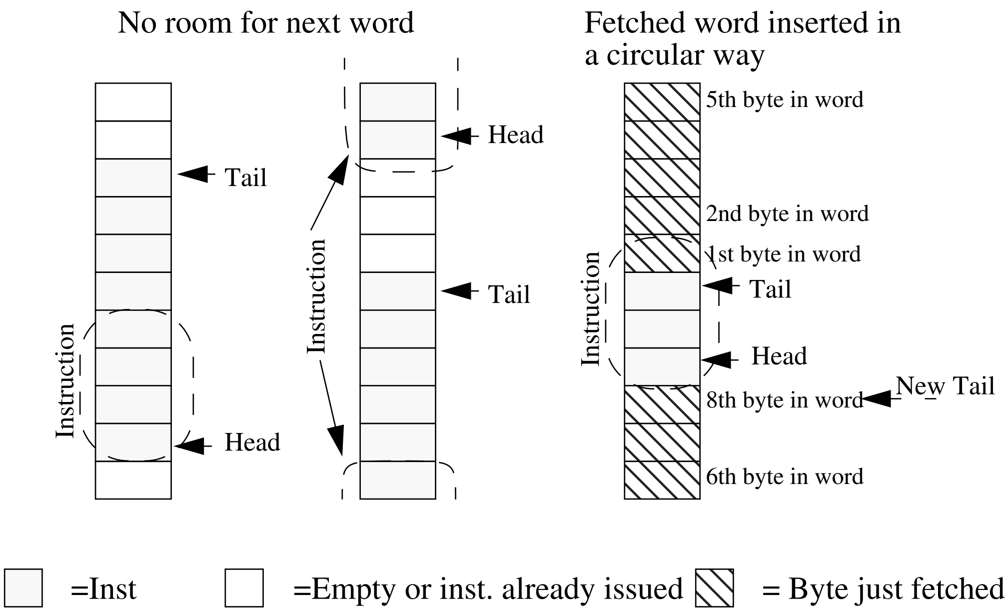


Figure 8.3 Principle operation of 11 byte circular buffer

cache-line straddling instructions, the number of energy consuming accesses to the cache is reduced; the cache can also be made simpler by omitting the output multiplexer if the contents of the entire cache line is fed directly into the input cross bar structure, see Figure 8.2. However, if 'n' is less than the line size of the cache, some level of multiplexing is required.

To determine 'n' and 'm', the worst-case scenario must be considered. The structure has been introduced to ensure a high issue rate; i.e. the number of instructions causing two cache accesses is reduced as much as possible. The worst-case scenario is where three bytes of a four-byte instruction are left in the buffer, lacking the last byte before it can be issued. The circular buffer therefore needs to ensure that the three bytes remain in the buffer while the succeeding 'n'-bytes are brought in from the cache. The following relation must therefore be satisfied:

$$m \geq n + 3 \qquad \text{(EQ 8.2)}$$

To limit the amount of overfetching the difference should be kept at as low as possible i.e:

$$m = n + 3 \qquad \text{(EQ 8.3)}$$

The results presented in section 8.2 showed that basic blocks are rarely as long as 32 bytes, which indicates that ‘n’ should be kept relatively small. Large values of ‘n’ will reduce the number of energy-consuming fetches from the cache but will make the cross-bar more complex. Moreover, the larger ‘n’ the greater the probability of overfetching; i.e that a control transfer instruction will force the m-byte register to be flushed before all the bytes in it have been used.

Given this architecture, two cache lookups for a single instruction will be required only when the target instruction of a CTI straddles two cache lines. Simulation shows that the number of instructions requiring two cache accesses can be reduced dramatically. Table 8.4 shows the percentage of instructions which require two cache accesses to be composed, for different line sizes in the instruction cache, assuming n = line size. Comparing these numbers to those in Table 8.2 it is clear that the performance penalty of variable-size instructions has been reduced significantly by introducing this *alignment architecture*. The performance penalty associated with variable-size instructions has been reduced from 6.7% to less than 0.5% for a 32-byte cache line. Furthermore it removes the performance incentive to build long cache lines which are energy consuming.

As shown in Chapter 5 the reduction in cache accesses translates into a reduction in energy consumption of the cache, as the energy consumed is proportional to the number of accesses. However, controlling and maintaining the structure with its cross-bars and finite state machines will be energy consuming.

From an implementation perspective, the choice of ‘n’ is between four bytes (a word) or the size of a cache line (16, 32 or 64 bytes). Table 8.5 shows the number of fetches from

Table 8.4 Percentage of instructions which require two cache accesses

	Cache line size		
Benchmark	**16 bytes**	**32 bytes**	**64 bytes**
cacti	0.57	0.47	0.06
dhry	0.17	0.01	0.002
espresso	0.50	0.31	0.15
fft	0.52	0.32	0.21
flex	2.10	1.31	0.46
hello	2.43	0.57	0.19
stcompiler	1.30	0.39	0.20
Average	**1.08**	**0.48**	**0.21**

the instruction cache into the structure for different values of 'n'[1], see Figure 8.2. The number of requests to the cache is reduced significantly when an entire cache line is fetched into the structure rather than just 4 bytes. Fetching 16 bytes, or more, at a time reduces the number of requests to the cache by at least 70%. However, there is a trade-off to be made; as 'm', and thereby the cache-line size, increases, so does the amount of external memory traffic from the cache.

Table 8.5 Number of fetches from instruction cache into alignment structure

		Quantity fetched into Alignment structure							
		n=4 bytes		n=16 bytes		n=32 bytes		n= 64 bytes	
Benchmark	**Inst**	**Fetch**	**R**[a]	**Fetch**	**R**	**Fetch**	**R**	**Fetch**	**R**
cacti	18,852,828	17,157,588	9	5,370,467	72	3,460,167	82	2,336,587	88
dhry	688,173	598,134	13	211,585	69	144,929	79	107,785	84
espresso	4,630,599	3,874,383	16	1,361,445	71	928,084	80	707,870	87
fft	1,104,931	912,559	17	287,090	74	181,509	84	117,849	89
flex	10,688,269	9,169,008	14	3,326,011	69	2,407,401	77	1,858,854	83
hello	52,175	45,060	14	15,989	69	10,695	80	8,904	83
stcompiler	1,865,924	1,614,804	13	593,979	68	428,284	77	332,242	82
Average			**14**		**70**		**80**		**85**

a. $R = \frac{Inst - Fetches}{Inst} \times 100\%$

Evaluating the cycle-time of this architecture is difficult, without reference to a detailed design. Fetching an instruction which is fully contained within the buffer is expected to

1. m= n + 3

be fast; if, however, it is necessary to fetch a second word from the cache, there is a significant timing overhead in fetching the word and passing it through the cross bar before the full instruction can be fetched through the multiplexers.

The energy consumption of the structure is equally difficult to assess. As shown in Table 8.5 there is a significant reduction in the number of requests to the cache and as the cache normally consumes a significant percentage of the power budget, see Chapter 2, this should yield some reduction in the total power budget. The energy consumption in the alignment structure is not negligible however. The m-byte storage block cannot be an integral part of the cache storage as it is controlled by instruction fetch engine and separate control logic rather than the program counter. The circuit surrounded by a dotted line in Figure 8.2 must therefore be considered a separate block adding to the overall size of the design.

In summary, the alignment architecture promises a performance and energy efficient solution to the problem of instructions which straddle cache lines.

8.3.2 The dual cache line architecture

Despite its advantages the 'alignment architecture' described in section 8.3.1 fails to exploit any spatial locality, which might exist within the 'n'-bytes, brought into the circular buffer. Once an instruction word is in the circular buffer, all information about the program counter value is lost. Loop bodies which are short enough to reside in a single cache line will be fetched repeatedly. To eliminate or reduce this 'overfetching' the 'Dual-Cache-Line' (DCL) architecture was developed.

Figure 8.4 shows the principle of the DCL architecture. The scheme works as follows: an instruction cache line is fetched into the 'CacheLine 1' register and its associated tag placed in the 'Tag 1' register. The tag-latches need to contain all the bits from the address,

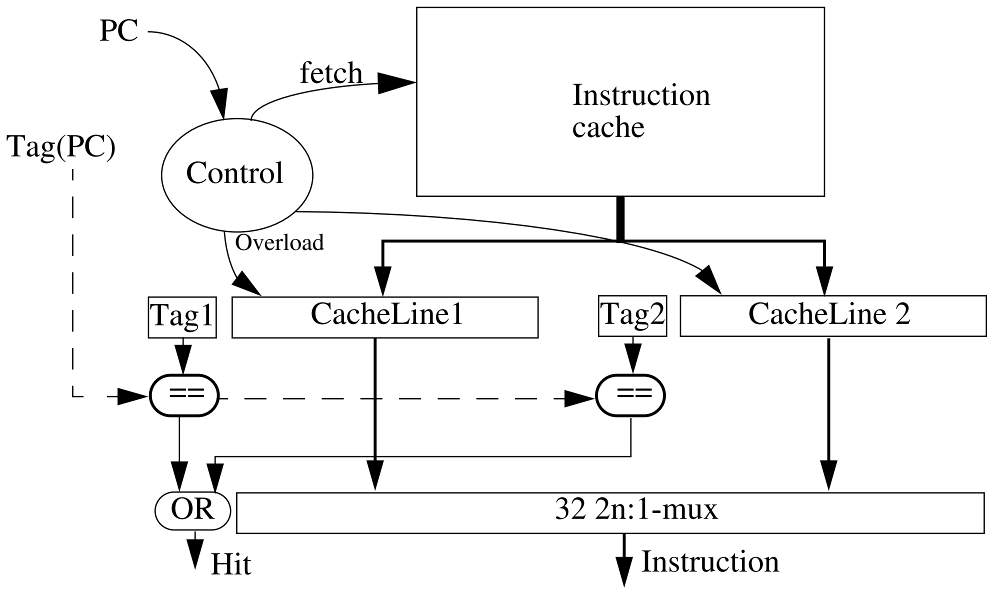


Figure 8.4 Dual cache line architecture

which are not used to access the byte within the line; this is equivalent to the number of bits in the tag-store of a fully associative cache. As long as the PC maps to that cache line, instructions are fetched from the ‘CacheLine 1’ register. When the byte-offset of the PC comes close to the end of the cache line the succeeding cache line is fetched into the ‘CacheLine 2’ register and the associated tag placed in the ‘Tag 2’ register. Therefore, when an instruction *may* cross the cache line boundary of ‘CacheLine 1’ the remaining bytes of the instruction will already be in the ‘CacheLine 2’ register[1]. This works on a cyclic basis so that when the program execution approaches the end of ‘CacheLine 2’ its successor will be fetched into the ‘CacheLine 1’ register. Note that if the successor is already in the alternate ‘CacheLine’ register no requests to the instruction cache are required. This reduces the performance penalty associated with variable-size instructions, as did the alignment architecture described above. In addition dual cache requests are reduced even further due to the detection of spatial locality, see Table 8.6. This is particularly true for the longer cache lines of 32 or 64 bytes. Furthermore, if a branch is taken and the target is in either ‘CacheLine 1’ or ‘CacheLine 2’ no cache access will be

1. Assuming the request hit in the instruction cache.

Table 8.6 Percentage of instructions which cannot be fetched in one cycle

Benchmark	Cache line size 16 bytes	32 bytes	64 bytes
cacti	0.6	0.5	0.06
dhry	0.8	0.3	0.1
espresso	1.2	0.2	0.06
fft	0.5	0.3	0.1
flex	2.0	0.8	0.09
hello	1.2	0.3	0.1
stcompiler	1.0	0.2	0.1
Average	**1.04**	**0.37**	**0.09**

initiated. The two registers ‘CacheLine 1’ and ‘CacheLine 2’ effectively form a small, dual-ported fully-associative level-0 instruction cache with two lines and a LRU replacement algorithm.

The DCL architecture requires more storage than the Alignment-architecture; ‘2x(8n+32-$\log_2$M)’ vs. ‘8x(n+3)’ latches registers are required, where ‘M’ is the cache size in bytes and ‘n’ is the line size in bytes.

Compared to the alignment architecture presented in section 8.3.1, this architecture eliminates the need for the large cross-bar, see Figure 8.2; the output from the cache is simply latched into the ‘CacheLine 1’ or ‘CacheLine 2’ as required. The output multiplexer can be made of separate 2n:1 multiplexers which will allow any bytes in the output to be selected from any position within the two registers.

The architecture is in many ways similar to the technique used in the MII-architectures [Conte] and in the HP PA7100LC[1] [Case] to ensure a high content of instructions in a multi-instruction-issue architecture, where instructions for an instruction packet can come from multiple - not necessarily successive - cache lines.

1. Dual instruction issue processor

Table 8.7 shows how the number of requests to the instruction cache itself is reduced by 74% or more when instructions are fetched from one - or both of the two registers, ‘CacheLine 1’ and ‘CacheLine 2’.

If the numbers in Table 8.7 are compared to those in Table 8.5 it is clear that this DCL architecture shows a greater reduction in the number of requests to the cache itself than the alignment architecture described in section 8.3.1.

Table 8.7 Number of fetches from instruction cache into DCL

		Cache line size					
		16 bytes		32 bytes		64 bytes	
Benchmark	**Inst**	**Fetch**	**R[a]**	**Fetches**	**R**	**Fetch**	**R**
cacti	18,921,879	5,333,913	72	3,205,773	83	2,031,377	89
dhry	688,173	119,181	83	75,772	89	52,811	92
espresso	4,822,863	1,247,040	74	706,542	85	444,593	91
fft	1,104,931	278,167	75	150,505	86	82,148	93
flex	10,688,269	3,133,591	71	2,016,871	81	1,144,008	89
hello	52,173	13,876	73	8,618	83	5,891	89
stcompiler	1,865,924	534,206	71	347,720	81	234,804	87
Average			**74**		**84**		**90**

a. $R = \frac{Inst - Fetch}{Inst} \times 100\%$

From an energy perspective, this architecture reduces the number of energy consuming cache references. The two extra cache line registers can be built as a simple static latch circuit where the energy cost of reading is negligible compared to that of a RAM-access. There is an overhead relative to the alignment architecture in that two tag-comparators are required to compare each incoming address to the contents of the tag-latches, ‘Tag 1’ and ‘Tag 2’ in Figure 8.4. The number of bits in these tag-registers might be higher than in the tag-store in the cache. These two comparators will clearly consume energy, but it is expected to be minimal. The energy consumption of the design compared to a conventional cache is therefore approximately proportional to the reduction in references to the cache, see Table 8.7.

In summary, the DCL-architecture provides a way of assembling instructions which straddle cache lines. Due to the increased detection of spatial locality the number of cache references is reduced further than for the alignment architecture presented above. Consequently it is a more energy efficient architecture.

8.3.3 The eXtra-line architecture

The results in section 8.3.2 showed that spatial locality could be exploited by introducing a small level-0 cache formed by two registers. The number of references to the highly energy consuming main cache was reduced by 74% or more. Spatial locality was detected by keeping tags together with the cachelines when these were fetched from the cache.

The DCL architecture compared the tag of the incoming value of the program counter with the contents of the 'Tag' fields. However, for the majority of references such tag comparisons are not necessary as the accesses are sequential. Most tag comparisons can be replaced by a simple circuit which detects if the new program counter value maps to the same line as the previous one. Tag comparisons will therefore be necessary only when a control transfer instruction has been taken and the program counter consequently overwritten. Occurrences of this are simple to detect.

If the cache-line registers presented in section 8.3.2 are replaced by a single register, the architecture in Figure 8.5 evolves. This may appear similar to the block buffering described in Chapter 5, [Su] and [Okada]. However, it represents a novel extension to the block buffering scheme as the output multiplexer is wider than a cache line and thereby permits cache-line straddling instructions to be assembled. Furthermore, tag comparisons are only carried out when required, see above.

Note that the output multiplexer is considerably simpler to control than that of the DCL-architecture as the ordering of the bytes for an instruction is always the same. The latches

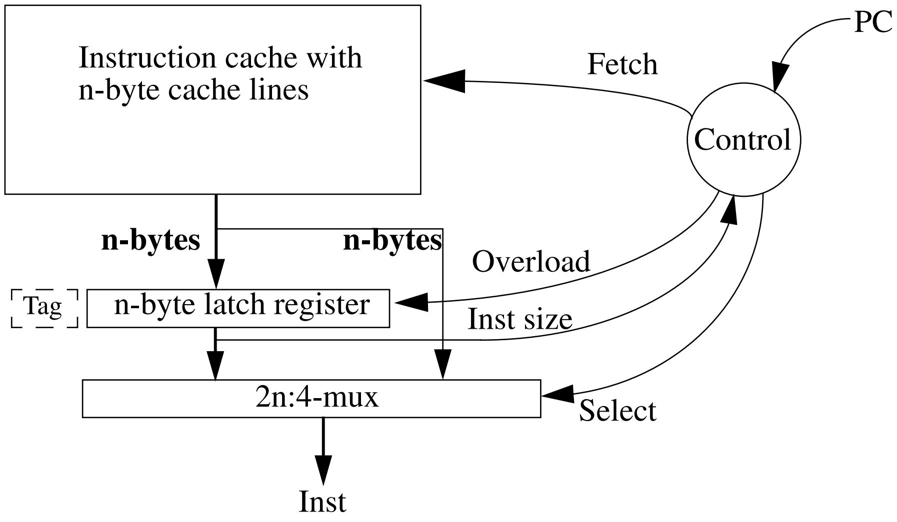


Figure 8.5 The eXtra-line architecture

in this architecture are ‘Master-Slave’ type, i.e. one value can be read out of the register while another value is being latched.

When an instruction, which straddles cache lines, is issued the first bytes will already be in the register. The control circuit will detect that the following line needs to be fetched for the instruction to be issued. This line will be fetched from the instruction cache and, while the instruction is sent off into the rest of the processor through the multiplexer, the newly fetched cache line will be latched into the register.

The control overhead for this architecture is minimal. The program counter circuit needs to detect whether an address does not map to the same line as the previous instruction and whether an instruction *may* straddle cache lines. The latter is very simple to detect from bit-changes in the least significant bits of the program counter and the bits of the instructions which indicate the size.

Compared to the architectures previously described, see section 8.3.1 and 8.3.2, this approach represents the smallest hardware overhead in terms of storage and control logic. There is a need to store only the amount of data equivalent to one cache line. The tag information is only required after taken branches; i.e. there is no need to perform a tag comparison for all requests. The ‘tag’ associated with the cache line held in the latch register, see Figure 8.5, has been drawn with dotted lines as the tag need not be an integral part of the cache/eXtra-line architecture. The program counter unit can simply track branch instructions and only perform a comparison between the PC-value before and after the branch to detect if a new line needs to be fetched into the register. This makes the energy consumption per access to the structure lower than for any of the architectures described in section 8.3.1 and 8.3.2.

Table 8.8 shows the number of fetches from the cache and the reduction in cache traffic. As for the architectures described in previous sections, the benefit increases with the line size. Comparing it to the DCL architecture from section 8.3.2, the average number of cache requests per instruction is 2.7% higher for this architecture.

Table 8.8 Number of fetches from instruction cache into eXtra-line

		Cache line/eXtra-line size							
		8 bytes		16 bytes		32 bytes		64 bytes	
Benchmark	**Inst.**	**Fetch**	**R**[a]	**Fetch**	**R**	**Fetch**	**R**	**Fetch**	**R**
cacti	18,921,879	9,085,543	52	4,952,058	74	2,941,767	84	2,040,187	89
dhry	688,173	330,385	52	203,445	70	136,593	80	86,120	87
espresso	4,822,863	2,107,520	56	1,227,371	75	792,147	84	562,239	88
fft	1,104,931	489,042	56	284,313	74	174,211	84	99,540	91
flex	10,688,269	5,104,123	52	3,156,604	70	2,140,844	80	1,446,826	86
hello	52,173	25,419	51	15,469	70	10,677	80	6,727	87
stcompiler	1,865,924	914,678	51	562,533	70	387,996	79	252,164	86
Average			**53**		**72**		**82**		**88**

a. $R = \frac{Inst - Fetch}{Inst} \times 100\%$

Like the DCL architecture, this provides both a significant performance improvement and reduces the energy consumption. The former results from the problem of instructions straddling cache lines[1] being almost eliminated, see Table 8.9; the latter from the

Table 8.9 Percentage of instructions which cannot be fetched in one cycle

	Cache line size			
Benchmark	**8 bytes**	**16 bytes**	**32 bytes**	**64 bytes**
cacti	0.8	0.2	0.1	0.06
dhry	3.1	1.7	1.1	0.16
espresso	2.4	1.1	0.8	0.08
fft	1.8	0.5	0.3	0.10
flex	2.7	2.0	0.8	0.11
hello	3.8	1.2	0.7	0.10
stcompiler	2.9	1.1	0.3	0.17
Average	**2.5**	**1.1**	**0.59**	**0.11**

reduction in he frequency of accesses to the highly energy consuming RAM-structures.

As the eXtra-line can be an integral part of the storage block in the cache the routing overhead is significantly lower for this architecture than for the DCL-architecture where a full cache line needs to be routed to either the CacheLine 1 or the CacheLine 2 register. This will imply lower energy consumption and a smaller overall design. These advantages are expected to offset the slightly higher number of references to the cache memory in the eXtra-line architecture and thus yield a more energy efficient architecture overall.

8.4 Summary

This chapter has proposed three ways of reducing the performance penalty caused by instructions which straddle cache line boundaries and hence might require two accesses to the instruction cache, with consequent degradation in performance.

1. and hence would require two cache lookups - in two cycles

All the strategies have the advantage that they reduce the number of references to the instruction cache. The energy consumption in RAM (mainly in caches) is a significant factor in the energy consumption of an entire microprocessor. Reducing the number of requests to the cache reduces the energy consumption of the cache proportionally and hence improves the energy efficiency. The energy consumption in the extra blocks introduced in the architectures described in section 8.3.1 - 8.3.3, i.e. in the latches and control circuits, is considered small compared to the energy consumed within the cache even though it must be clear that the structure in the DCL architecture is more energy consuming than the eXtra-line architecture due to the tag comparisons and routing overhead. The slight increase in number of cache references moving from the DCL-architecture to the eXtra-line architecture is therefore expected to be offset by the smaller routing overhead, fewer tag comparisons and smaller multiplexers.

Chapter 9 Cache design and dimensioning

The results presented in section 8.3 showed the effect of adding each of three instruction fetch mechanisms to the instruction cache to compensate for the problems which variable-size instructions introduce.

The results in Tables 8.5, 8.7 and 8.8 show that longer cache lines reduce the number of references to the instruction cache and therefore its energy consumption. However, Chapter 5 has shown that energy consumption per cache request increases with the line size. Furthermore, the I/O-activity and thereby the number of stalled cycles, the energy consumption in the I/O subsystem and the energy consumption in the external RAM increase with increasing line size. The optimal cache line size is therefore a compromise between these parameters: the energy consumption per cache access and the number of accesses to the cache. To assess the value of the architectural features the following sections will determine the performance and energy efficiency of systems specifying different cache configurations.

This chapter describes the results from a large number of simulations aiming at determining the optimal cache system for a HORN-processor system. Section 9.1 describes the system considered and presents expressions for performance and power consumption used to calculate the energy efficiency for each of the simulated configurations. Section 9.2 presents the results of a large number of simulations aimed at determining the cache parameters for the most energy efficient and best performing configurations of systems containing separate instruction and data caches. Section 9.3 presents the result from a set of similar simulations but using a unified cache. Section 9.4 summarizes the results.

9.1 Background for cache evaluation

Consider a simple system comprising the processor and a number of 8-bit Hitachi HM65256B memory modules [Hitachi]. The modules are organized in banks of 4 chips and are interleaved to allow the fetching of one 32-bit word per cycle, after some initial latency. The initial latency is given by the ratio of the access time of the RAM and the cycle time of processor. The RAM modules have an access time of 100ns[1] and a cycle time of 190ns.

Each memory *bank* consumes 4x200mW = 0.8W, worst case when active and 4x0.5mW = 2mW when idle [Hitachi]. The power consumption in the processor core is assumed equal to that of an ARM3 core, i.e. 453mW at 12MHz, see Chapter 2. The power consumption of the system comprising the processor core, the cache and the external RAM is therefore expressed as:

$$P_{System} = P_{Core} + P_{Cache\,(s)} + P_{Mem} \qquad \text{(EQ 9.1)}$$

The power consumption of the processor core scales with the cycle time:

$$P_{Core} = \frac{cycletime_{ARM3}}{cycletime} \times P_{Core,ARM3} \qquad \text{(EQ 9.2)}$$

The power consumption in the cache(s) is proportional to the frequency of accesses:

(EQ 9.3)

$$P_{Cache} = \frac{1}{cycles \times cycletime} \times \left(req \times E_{Cache,\,RR} + (nbWback + misses) \times linesize \times E_{Cache,RW}\right)$$

where ‘nbWback’ denotes the number of dirty lines written back to external memory and ‘misses’ denotes the number of cache lines fetched from external memory. The expressions for $E_{Cache,RR}$ and $E_{Cache,RW}$ are as derived in Chapter 5.

1. Other manufacturers such as Toshiba [TOSHIBA2] have faster RAMs, however they are more power consuming

By defining *memBusy* as the percentage of all cycles during which the memory is busy:

$$memBusy = \frac{memLat \times (nbWback + misses) \times linesize}{cycles} \qquad \text{(EQ 9.4)}$$

the power consumption in the external memory is expressed as:

$$P_{Mem} = memBusy \times 0.8W + nbBanks \times \left(1 - \frac{memBusy}{nbBanks}\right) \times 0.002W \qquad \text{(EQ 9.5)}$$

Where *nbBanks* denotes the number of banks in the external memory system.

The cache timing analysing tool Cacti [Wilton] was used to calculate cycle times for the different cache configurations, see Chapter 5; the cache was assumed to be the speed limiting component in implementation of the architecture [Juan]. Table 9.1 shows the cycle times computed by Cacti for direct mapped (DM) and a number of set-associative configurations.

Table 9.1Cache cycle time [ns] for different configurations

Cache size [bytes]	Line size [bytes]								
	16			32			64		
	Associativity:			Associativity:			Associativity:		
	DM	2-way	4-way	DM	2-way	4-way	DM	2-way	4-way
4K	7.02	9.29	9.63	6.66	9.45	10.14	6.49	10.13	11.32
8K	7.80	9.78	10.15	7.21	9.97	10.47	6.99	10.44	11.54
16K	8.58	10.43	11.04	8.00	10.74	11.17	7.62	11.14	11.97
32K	9.45	11.53	11.81	8.91	11.41	12.14	8.51	12.01	13.01
64K	10.8	12.85	13.12	9.82	12.69	13.23	9.50	13.10	14.15

These cycle times are clearly much faster than any cache technology available when the ARM3 was designed. Extrapolating these numbers onto the results from the ARM3 might therefore seem inappropriate as the ARM3 technology might not scale as easily. However, lacking more detailed information on the ARM3 technology it was decided to use the timing information obtained from Cacti.

The number of external memory banks required is therefore a function of the cycle time of the processor (given by the cycle time of the cache) and the memory chips:

$$nbBanks = \frac{T_{RAM}}{T_{Cache}} = \left\lceil \frac{190ns}{T_{Cache}} \right\rceil \qquad \text{(EQ 9.6)}$$

The number of cycles required to execute a program is calculated as:

(EQ 9.7)

$$\begin{aligned} cycles = {} & inst + (misses + nbWback)_{Dcache} \times \left(memLat + linesize_{Dcache} - 1 \right) \\ & + misses_{Icache} \times \left(memLat + linesize_{Icache} - 1 \right) + BranchesPenalties \end{aligned}$$

where *memLat* is the memory latency in cycles given as the ratio of the cache cycle-time and the RAM access-time of 100ns mentioned above.

A pipeline as shown in Figure 6.3 has been assumed. All branches are predicted taken. As described in Chapter 6 this implies a one cycle penalty for correctly predicted branches and a two cycle penalty for branches which are predicted wrongly. *BranchPenalties* represents this penalty.

The performance of the processor is calculated as:

$$Performance = \frac{inst}{cycles} \times \frac{1}{Cycletime} \qquad \text{(EQ 9.8)}$$

and, as shown in section 4.1, the EE can be calculated as:

$$EE = \frac{(Performance)^2}{PowerConsumption_{System}} \qquad \text{(EQ 9.9)}$$

9.2 Performance and energy efficiency of separate cache configurations

Consider the system architecture shown in Figure 9.1. Using Equation 9.8 to quantify performance and Equation 9.1 to quantify power consumption, a large number of cache

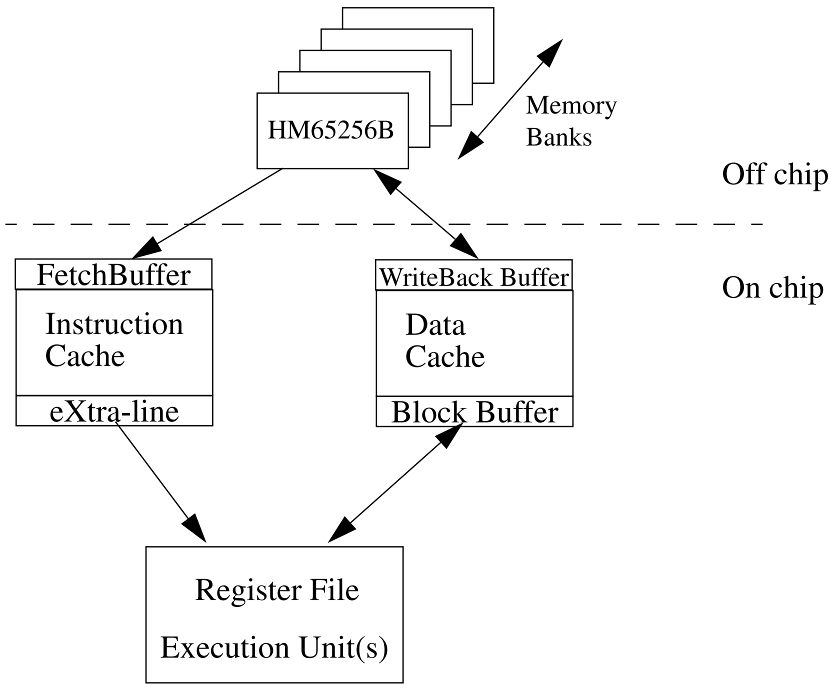


Figure 9.1 System architecture with separate caches

configurations have been simulated, see Table 9.2. The power consumption, performance and energy efficiency of each configuration has been calculated.

The instruction cache (Icache) architecture is the eXtra-line architecture described in section 8.3.3 combined with a fetch buffer as described in section 5.9. The Fetch Buffer, separate from the 'eXtra-line', will allow the prefetching of instructions, see section 3.3, while instructions can continue to be fetched from the eXtra-line. Furthermore, writing a

Table 9.2 Simulated configurations

Parameter	Values simulated	
Cache architecture	Icache:	eXtra-line
	Dcache:	Block Buffering
Cache size [bytes]	Icache:	8K, 16K, 32K, 64K
	Dcache:	4K, 8K, 16K, 32K
Line size [bytes]	Icache:	16, 32, 64, 128
	Dcache:	8, 16, 32, 64, 128
Associativity	Direct Mapped, 2-way skewed-associativity[a]	
Replacement Strategy[b]	Pseudo-LRU	

a. See [Seznec]

b. When 'associativity' = 2-way skewed-associative whole line into the cache in one operation will consume less energy than writing N words one per cycle, as the overhead of tag-comparisons are eliminated/reduced. There is no need for a Write-Back Buffer as there will be no dirty lines in the instruction cache.

The data cache (Dcache) should contain a Block Buffer and a Write-Back Buffer as it greatly improves the performance of the cache and thereby of the system. The Block Buffer can also serve as a Fetch Buffer and thus reduce the energy consumption further without affecting performance. The words for the new cache line will be latched in the Block Buffer and only when all words have arrived will the entire line be written into the data cache. Individual writes are dealt with in a write through manner as described in section 5.9 to avoid the need for a coherency protocol. Thus the number of buffers is limited to two in each of the separate caches.

The value of reducing the number of cache references through the Fetch and Write-Back Buffers over the system just comprising the Block Buffer or the eXtra-line varies depending on the performance and hence on the parameters of the specific cache configurations; it is clear however that the value increases with the line size in the caches, see Table 9.3. Throughout these calculations it has been assumed that all words in a Write-Back Buffer will have been written back to external memory before the next data cache miss is encountered.

The total power consumption of the system is an important measure. The power consumption for each configuration has therefore been calculated and the configurations sorted accordingly. This allows energy efficiency, EE, and performance to be plotted against the power consumption. The results for two benchmarks are shown Figures 9.2 and 9.3[1], while those for the remaining benchmarks are shown in Appendix A.

Table 9.3 Reduction in cache accesses due to fetch- and writeback-buffers [%]

	Instruction cache[a] line size: [bytes]				Data cache line size: [bytes]				
Benchmark	**16**	**32**	**64**	**128**	**8**	**16**	**32**	**64**	**128**
cacti	6.34	11.5	17.8	24.8	0.12	0.24	0.41	0.69	10.4
dhry	5.72	12.2	21.0	28.3	0.77	2.07	4.0	11.5	16.3
espresso	5.30	10.8	19.5	33.1	1.83	4.13	8.52	16.4	29.2
fft	1.98	4.52	9.74	16.8	8.97	17.5	28.4	39.3	51.0
flex	2.64	5.33	11.1	22.1	2.76	6.44	13.1	24.1	41.7
hello	16.5	31.2	45.4	57.6	2.86	5.85	10.9	18.2	27.8
stcompiler	11.8	22.4	39.1	57.7	2.52	5.91	12.2	22.2	36.5

a. Both the instruction and the data cache were 8K bytes and direct mapped

The graphs are ‘performance against power consumption’ plots where constant energy efficiency, EE, levels are shown with dotted lines. The energy efficiency levels increases towards the top-left. The difference between the energy efficiency contours, the EE stride, is shown at the right of each graph. Each graph highlights the most/least energy efficient

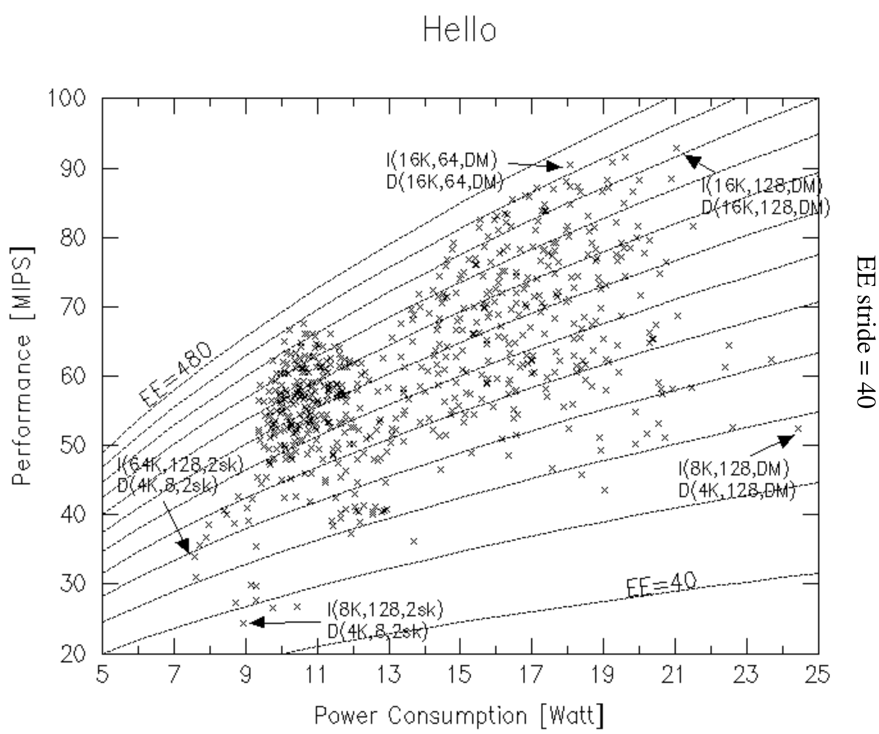


Figure 9.2 EE and performance versus power consumption, hello

1. Key to read the configurations:
I(Cache size [bytes], Line size [bytes], associativity) = Instruction Cache
D(Cache size [bytes], Line size [bytes], associativity) = Data Cache
associativity: DM = Direct Mapped; 2sk = 2-way skewed-associative

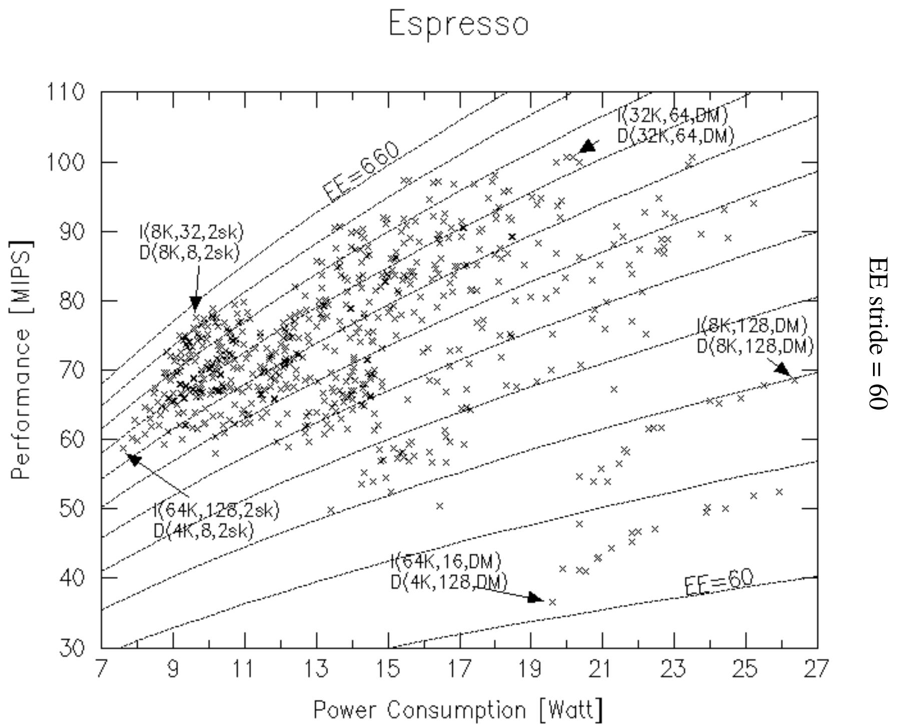


Figure 9.3 EE and performance versus power consumption, espresso

and best/worst performing configurations as well as the most/least power consuming configurations. Note that the most energy efficient configurations are often smaller than the best performing configurations. However, the common characteristic is that the most energy efficient configurations have shorter cache lines than the best performing ones.

The graphs show that the most energy efficient configurations are between 4% and 23% more energy efficient than the best performing one. The typical difference is approximately 10%. For some[1] of the benchmarks the optimal energy-efficient configuration consumes significantly less power (reduction greater than 30%) than the best performing configuration. Consequently, the performance for the most energy efficient configurations is also lower than that of the best performing configurations.

1. Especially espresso, flex and stcompiler

The graphs clearly highlight the most energy efficient configurations. The graphs for espresso, flex and hello show that energy efficiency values ‘close to’ (within 5%) the optimal value can be obtained across a wide range of power consumptions. This means that a designer can choose the configuration which meets the system requirements for performance and power consumption while maintaining a high energy efficiency.

Consequently, if the performance of the most energy efficient configuration is too low, the graphs can be used to choose the most energy efficient configuration which will meet a performance requirement. Equally, if the power budget for the product does not permit the implementation of the most energy efficient configuration, the graphs can be used to choose the most energy efficient and best performing configurations within the power budget.

All the graphs show that the most power consuming configurations are small direct mapped caches with long cache lines, while the least power consuming configurations specify very large, 2-way skewed-associative instruction caches with long lines.

The data caches in the optimal configurations are smaller than the associated instruction cache and have shorter lines. The exception is for the FFT benchmark, see Appendix A; this benchmark differs from the rest of the benchmarks in that it is characterised by a very large and regular data set.

As Tables 8.4 and 9.3 suggest that there is a significant gain from the use of long cache lines in both instruction and data caches, the energy efficiency has been plotted against the line size in the instruction cache for different instruction cache sizes. The data cache was fixed at the configuration found to be the most energy efficient, see Figures 9.2 and 9.3 and Appendix A. The results are shown in Figures 9.5, 9.4 and in Appendix B. It is characteristic that the line size in the optimal instruction cache configuration is the

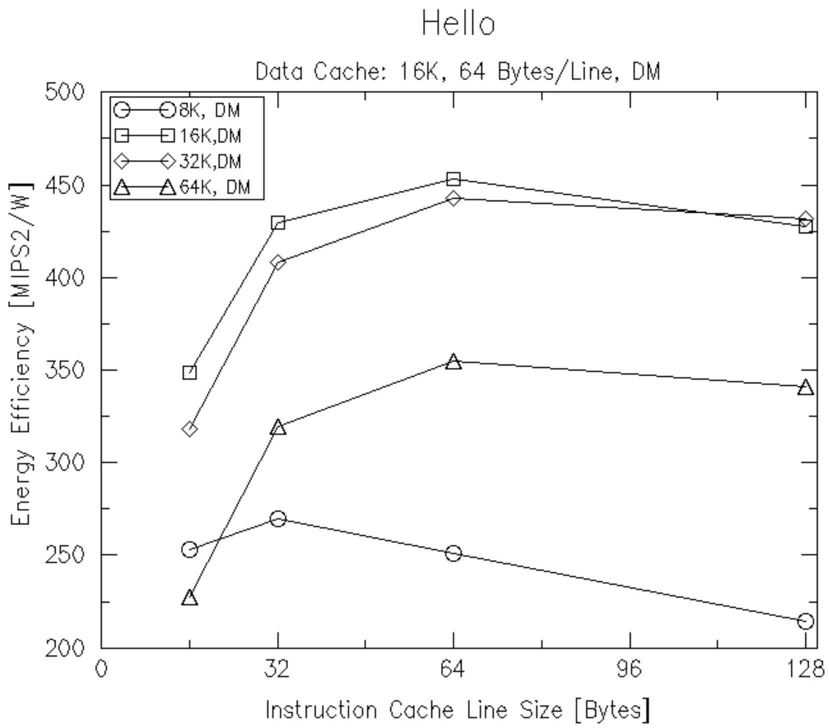


Figure 9.4 Energy Efficiency versus instruction cache line size, hello

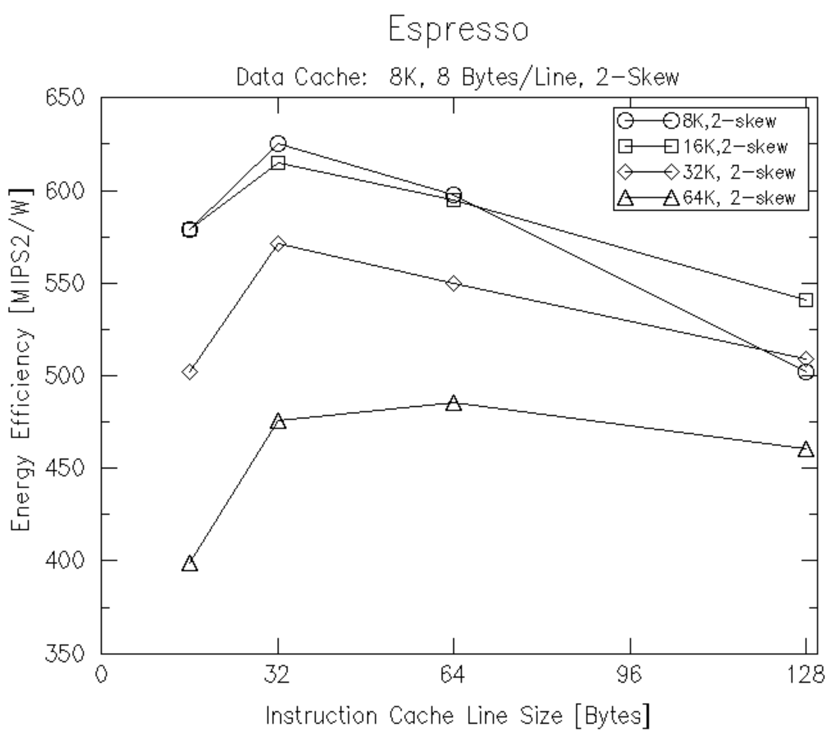


Figure 9.5 Energy Efficiency versus instruction cache line size, espresso

optimal line size for almost any instruction cache size. This is due to the basic block characteristics for the benchmarks described in Chapter 4 and section 8.3.3. Furthermore, the trend is that it is ‘less damaging’ to the energy efficiency if the line size is increased rather than reduced.

Tables 9.4 and 9.5 present the best performing and most energy efficient configurations. Table 9.4 indicates that a 16K choice for both instruction and data caches would be near optimal. For energy efficiency the optimal sizes are smaller, 8K or 16K bytes. This is not a surprise despite the finding in Chapter 5 that the energy consumption of the cache increases with the size and the line size. The eXtra-line architecture ensures that accesses to the cache itself are rare and the power consumption in the cache is therefore not as significant as the results in Chapter 2 may have suggested. The results also show that

Table 9.4 Optimal performance configurations

	Instruction cache - eXtra-line			Data cache - Block Buffer		
Benchmark	**Size [bytes]**	**Line size [bytes]**	**Assoc.**	**Size [bytes]**	**Line size [bytes]**	**Assoc.**
cacti	16K	128	DM	16K	128	DM
dhry	16K	128	DM	8K	32	DM
espresso	32K	64	DM	32K	64	DM
fft	16K	128	DM	16K	64	DM
flex	16K	64	DM	16K	32	DM
hello	16K	128	DM	16K	128	DM
stcompiler	16K	64	DM	16K	64	DM

cache lines should be long, although no line size can be identified as optimal; 64 bytes per line appears to be a good compromise. Again, the tables show that the line sizes should be shorter when optimizing for energy efficiency rather than for performance. When optimizing for performance it is clear that the caches should be direct-mapped to yield a cycle-time as fast as possible, see section 5.7 on page 98. In contrast, when optimizing for energy efficiency, where the optimal caches are generally smaller with shorter lines, half

Table 9.5 Optimal Energy Efficiency configurations

	Instruction cache - eXtra-line			Data cache - Block Buffer		
Benchmark	**Size [bytes]**	**Line size [bytes]**	**Assoc.**	**Size [bytes]**	**Line size [bytes]**	**Assoc.**
cacti	16K	64	DM	4K	64	DM
dhry	16K	64	DM	8K	32	DM
espresso	8K	32	2sk	8K	8	2sk
fft	8K	64	DM	32K	64	DM
flex	8K	32	2sk	8K	8	2sk
hello	16K	64	DM	16K	64	DM
stcompiler	16K	32	2sk	16K	64	2sk

of the benchmarks obtain higher energy efficiency with a skewed-associative configuration.

Table 9.4 and 9.5 also show that the optimal data caches are often smaller than the instruction caches for the same benchmark and are usually smaller than the largest ones simulated. The line sizes for the optimal data cache configurations are often shorter than those in the instruction cache and shorter than the longest ones simulated. It is a general result that the skewed-associative configurations neither perform as well, nor are as energy efficient, as the direct-mapped caches due to the slower cache access time.

However, if the cache lookup is not the time critical stage in a pipeline (for example due to slow functional units or register files) the cycle time of the processor is given by the delay through these stages independently of the cache configuration. If the cycle time of the entire processor is fixed at e.g. 15ns (66MHz), slower than any cache configuration reported above, the results change, see Tables 9.6 and 9.7. The optimal configuration is now a 2-way skewed-associative configuration for both performance and energy efficiency. Optimal performance is obtained with large instruction caches: 64K bytes, 128 bytes/line and large data caches: 32K bytes. The optimal line size in the data cache varies but it is clear that it is longer than for the simulations presented in

Table 9.6 Optimal performance configurations (cycle time = 15ns)

Benchmark	Instruction cache - eXtra-line Size [bytes]	Line size [bytes]	Assoc.	Data cache - Block Buffer Size [bytes]	Line size [bytes]	Assoc.
cacti	64K	128	DM	32K	8	DM
dhry	64K	128	2sk	32K	32	2sk
espresso	64K	128	2sk	32K	128	2sk
fft	64K	128	2sk	32K	32	2sk
flex	64K	128	2sk	32K	16	2sk
hello	64K	128	2sk	32K	32	2sk
stcompiler	64K	128	2sk	32K	16	2sk

Table 9.7 Optimal Energy Efficiency configurations (cycle time = 15ns)

Benchmark	Instruction cache - eXtra-line Size [bytes]	Line size [bytes]	Assoc.	Data cache - Block Buffer Size [bytes]	Line size [bytes]	Assoc.
cacti	32K	64	2sk	4K	8	2sk
dhry	32K	64	2sk	4K	8	2sk
espresso	16K	32	2sk	4K	8	2sk
fft	8K	64	2sk	32K	32	2sk
flex	16K	64	2sk	8K	8	2sk
hello	16K	32	2sk	8K	16	2sk
stcompiler	32K	64	2sk	4K	8	2sk

Tables 9.4 and 9.5. The results also show that the most energy efficient instruction cache configurations now are larger than was the case for the most energy efficient configurations as shown in Table 9.5 while the data cache configurations in general are smaller than those shown in Table 9.5.

The optimal configurations might well be too large to implement on a chip. A choice of the most energy efficient configuration must therefore consider implementation feasibility. The tables in Appendix C present the results of cache configurations simulated and these can be used to choose the best configuration given the constraints of silicon area and power consumption.

Appendix C also shows the power consumption in the caches as a percentage of the power consumption in the entire system. This ratio varies dramatically dependent on the cache configuration. For the optimal configurations this ratio is between 25% and 30%. As the percentage of the power consumed in the external RAM is also very low (less than 10%) further improvement in the energy efficiency of the processor system might obtained by examining other blocks than the cache.

9.3 Unified cache

A unified cache (combined instruction- and data cache) often implies a significant performance penalty when compared to separate caches as each data reference causes contention at the cache port. However, as the number of instruction references to the cache can be greatly reduced due to the instruction cache architectures discussed in section 8.3, the available cache bandwidth can be used to serve data references, see Figure 9.6.

The probability of contention in a conventional unified cache, which serves instruction and data requests from the same port, is equal to the probability of a memory referencing instruction. Chapter 4 showed that approximately one in four instructions is a memory

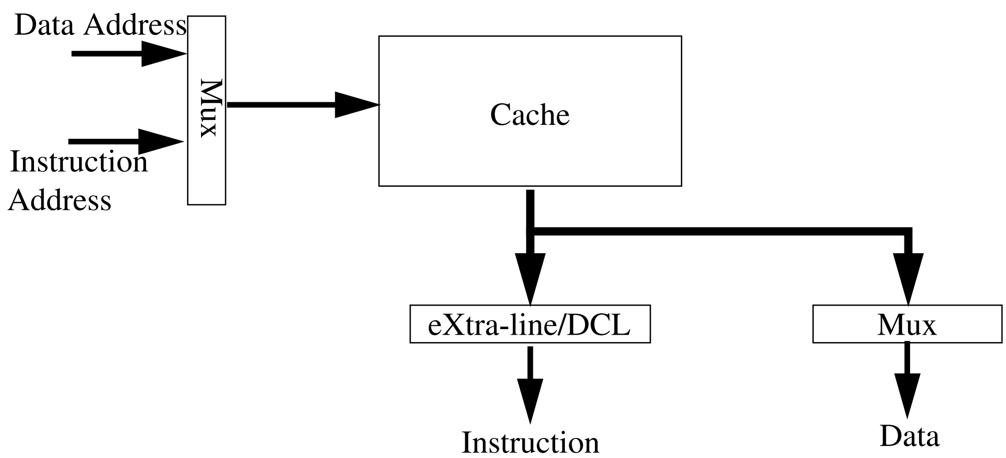


Figure 9.6 Unified cache serving both instruction- and data requests

referencing (load/store-type) instruction. The probability of contention, $P_{contention}$, at the cache port is therefore:

$$P_{contention} = P_{\mathrm{instref}} \times P_{\mathrm{memref}} = 1.0 \times 0.25 = 0.25 \qquad \text{(EQ 9.10)}$$

i.e 25%. By introducing the eXtra-line architecture, see section 8.3.3, the number of instruction references to the cache was reduced by 88% for long cache lines. The probability of contention, $P_{contention}$, at the cache port is therefore reduced to:

$$P_{contention} = P_{\mathrm{instref}} \times P_{\mathrm{memref}} = (1.0 - 0.88) \times 0.25 = 0.03 \qquad \text{(EQ 9.11)}$$

i.e a 3% probability of contention. This is a small penalty compared to the 25% probability of contention in a conventional unified cache. The system thus behaves almost as well as a dual-ported unified cache, but with a much reduced energy budget.

The hit rate of the unified cache will typically be lower [Patt, chap. 8.3] than that of a system comprised of separate instruction and data caches. In order to perform as well as separate caches the unified cache will therefore have to be larger than the largest of the two separate caches.

9.3.1 Performance and energy efficiency of unified cache configurations

Using Equation 9.8 to quantify performance and Equation 9.1 to quantify power consumption, a large number of unified cache configurations have been simulated, see

Table 9.8 Simulated configurations

Parameter	Values simulated
Cache architecture	eXtra-line, see Figure 9.6
Cache size [bytes]	4K, 8K, 16K, 32K, 64K
Line size [bytes]	16, 32, 64, 128
Associativity	Direct Mapped, 2-way skewed-associativity
Replacement Strategy[a]	Pseudo-LRU

a. When ‘associativity’ = 2-way skewed-associative

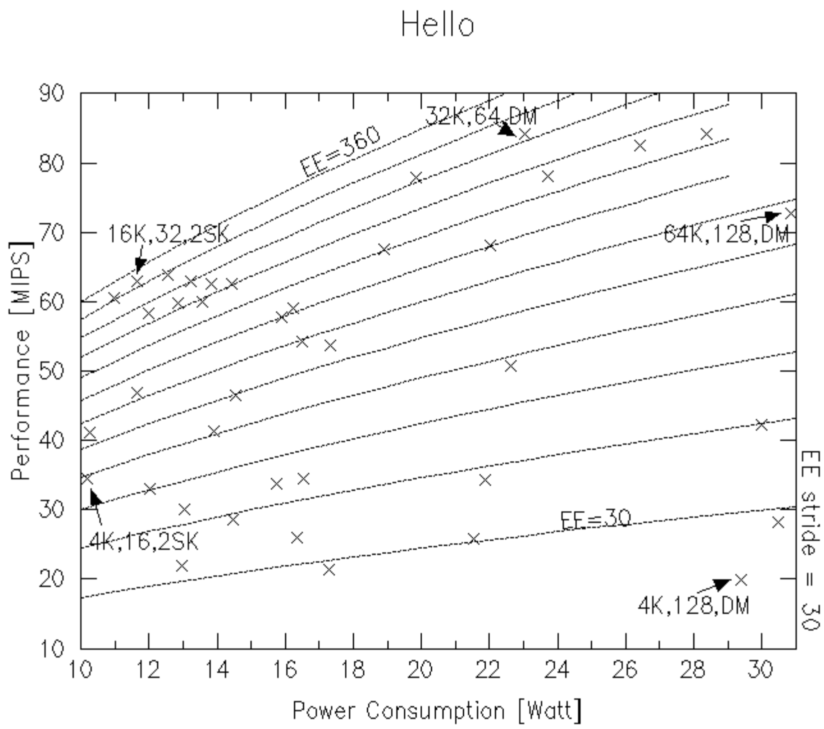


Figure 9.7 EE and performance versus power consumption, hello

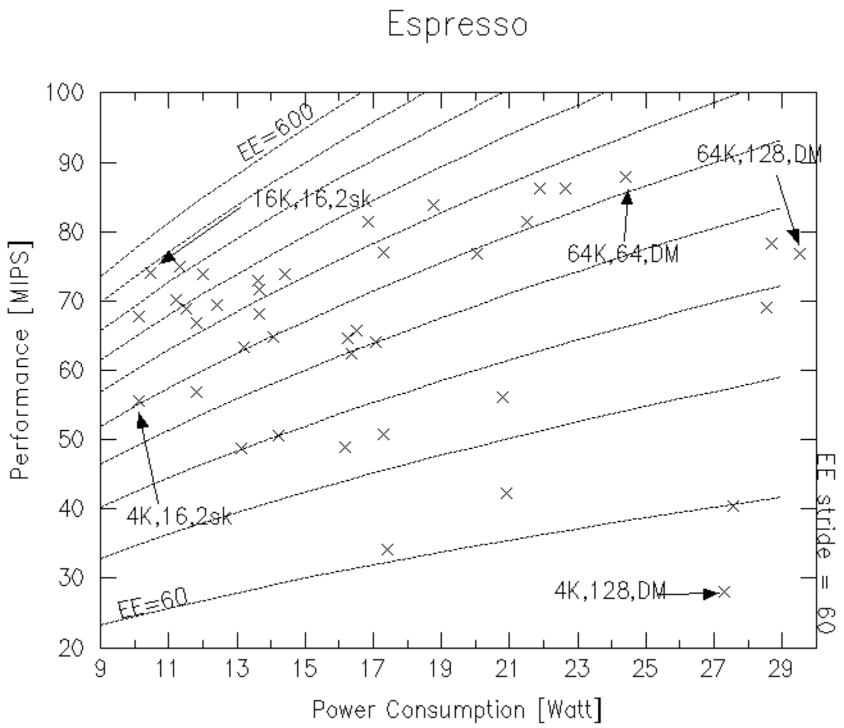


Figure 9.8 EE and performance versus power consumption, espresso

Table 9.8, and the power consumption, performance and energy efficiency of each of them have been calculated. The results for two of the benchmarks are presented in Figures 9.7-9.8 while the results for the remaining benchmarks are shown in Appendix A.2.

Comparing these graphs to the graphs in section 9.2 and Appendix A.1 shows that the performance of the best performing unified configurations is 8% lower than that obtained with separate caches. Moreover the energy efficiency of the most energy efficient configuration is 24% lower than that obtained with separate caches. This suggests that future processors concerned with energy efficiency should specify separate caches.

There are however other reasons for building a unified cache: A unified cache retains the same memory model as if no cache was present, the cache is simply a 'buffer' between the processor core and the rest of the memory hierarchy; separate caches require more control such as bus-arbitration. Furthermore, self-modifying code is simple to handle in a unified cache, whereas it requires special handling in separate caches. Also if code and data segments are not distinct, a cache line may contain both instructions and data. Consequently there may be replication of data in the two caches which may imply a higher energy efficiency in the unified configuration than in the separate caches. The HORN compiler lays-out code such that code and data segments are distinct, see section 3.5, so separate caches are feasible.

Despite these other considerations, a significant reason for building a unified cache is chip-area. It is often claimed that a unified cache of size 'X' performs better that two separate caches of size 'X/2'. From this premise it may be argued that if data/instruction collisions have been almost eliminated due to the structures presented in Chapter 8 then the overall performance for a given 'total cache size' would be better in a unified configuration than in a configuration of separate caches. Results collected for this work show that this is not necessarily the case, mainly due to the faster cycle time of the smaller caches, see Table 9.9

Table 9.9 Comparison between large unified cache and smaller separate caches

Benchmark	Unified Configuration	Performance [MIPS]	Separate Configuration[a]	Performance [MIPS]
Cacti	16K,128,DM[b]	114	8K,128,DM/8K,64,DM[c]	109
Dhry	16K,64,DM	104	8K,128,DM/8K,64,DM	99
Espresso	16K,32,DM	81	8K,64,DM/8K,32,DM	90
Fft	16K,64,DM	92	8K,128,DM/8K,64,DM	94
Flex	16K,32,2sk	75	8K,64,DM/8K,32,DM	86
Hello	16K,32,2sk	63	8K,64,DM/8K,64,DM	67
Stcompiler	16K,32,DM	64	8K,32,2sk/8K,32,2sk	67
Average		**85**		**87**

a. Instruction cache with eXtra-line and data cache with Block Buffer

b. Format: Total cache size [bytes], Line size [bytes], Associativity (DM: Direct Mapped; 2sk: 2-way skewed-associative)

c. Format: Instruction cache/Data cache

9.4 Summary

Based on the equations derived in Chapter 5, section 9.1 derived expression for performance, power consumption and energy efficiency for a HORN-processor system. Section 9.2 presented the most energy efficient instruction configurations of separate instruction and data caches as well as the best performing configurations for a range of benchmarks. The results showed that the optimal configurations were smaller than the largest ones simulated and that the best performing instruction cache configurations were 16K bytes in size with long cache lines of 64 or 128 bytes. It was demonstrated that the most *energy efficient* cache configurations are smaller and have shorter lines than the best performing cache configurations. The optimal cache configurations are mostly direct mapped for both performance and energy efficiency as they yield the fastest cache and hence processor cycle time.

Larger caches lead to longer cycle times, see Chapter 5, and thereby lower overall performance. Furthermore, large caches consume more energy per access than small caches. Long cache lines maximize the effect from the DCL- or eXtra-line architectures where there is a high degree of spatial locality, as is the case in instruction caches. For data caches the spatial locality is lower and shorter cache lines are therefore beneficial. The worst performing and least energy efficient configurations are consequently: instruction caches with very short cache lines and data caches with very long cache lines.

The results (shown in Appendix C) have shown that the caches do not dominate the power budget when the eXtra-line and block-buffer architectures are introduced. Future energy effective processors which specify eXtra-line or block-buffer cache architectures should therefore address other blocks in the processor system.

Chapter 10 Conclusions

This thesis has investigated how the careful specification of a microprocessor architecture can improve both the performance and energy efficiency of the final product. It has identified the blocks in the design which most affect these two measures and has investigated a number of architectures to improve them. This final chapter summarizes the conclusions from each chapter, draws further conclusions, assesses the work and suggests areas for future research with the goal of optimizing the energy efficiency of microprocessor systems.

10.1 Summary

Chapter 2, *'Power consumption in an ARM3-system'* presents results from an earlier study which measured the power consumption of various blocks in the ARM3 microprocessor. The study identified the cache as the most power consuming block in the implementation. The conclusion that an energy efficient processor architecture should specify an energy efficient cache architecture was used as a basis for much of the work reported here.

Chapter 3, *'Baseline HORN architecture'* presented the processor architecture which has formed the basis of this work and the changes it has undergone during the project. These changes, and the subsequent changes in compiler and functional simulator, have provided opportunities for evaluating a number of architectures at a detailed level.

In order to compare architectural features a suitable metric needed to be established. Chapter 4, *'Metrics and benchmarks'* divided processor applications into three classes and presented suitable metrics for each. Based on this it was decided to analyze the HORN architecture as a conventional microprocessor, even though this excluded some applications which the architecture was defined to address. The benchmark suite was chosen accordingly.

In view of the identification of on-chip RAM, notably the cache, as the most power consuming block, Chapter 5, *'Energy consumption in caches'* established how energy consumption in caches scales with the traditional cache parameters such as size, line size and degree of associativity. A number of 'newer' cache architectures were also analyzed. The effect the cache parameters had on the cache timing was also analyzed and the conclusion was drawn that the line size and degree of associativity should be kept low to reduce energy consumption. It was shown that to minimize the cycle time of the cache and thus improve the cycle time of the processor, the number of cache lines should be kept as low as possible as should the degree of associativity. There was therefore a trade-off to be made between the cycle time and energy consumption of the cache, as shorter cache lines implied lower energy consumption but longer cycle time. Similarly, although a high degree of associativity was found to yield a better hit-rate, it also produced a higher energy consumption.

As the HORN architecture breaks some of the dogmas associated with RISC architectures and introduces novel features such as dual-instruction branches, memory mapped registers and variable-size instructions; Chapter 6, *'Dual instruction branch'*, Chapter 7, *'Register file architectures'* and Chapter 8, *'Instruction fetching'* have investigated the effect of these architectural choices on performance and for energy efficiency.

Chapter 6 evaluated the effect of splitting the actions of a branch instruction into two: a 'go' instruction which sets up the target and a 'leap'-instruction which evaluates the condition and specifies the branch shadow. This structure was proposed to reduce or eliminate the branch penalty and improve the hit-rate in the instruction cache.

The chapter showed that if the early specification of the target for the branch is exploited to prefetch from the target into a shadow pipeline, the performance can be improved. However, due to the increased number of cache accesses, the scheme is not energy efficient in a single instruction issue implementation.

The chapter also examined the performance and energy efficiency of the scheme in a dual issue configuration. In such an implementation, the performance advantage of the two-instruction branch is so significant that the scheme is more energy efficient than a conventional single-instruction branch, despite a higher energy consumption

Chapter 7, *'Register file architectures'* evaluated the effect different register file architectures have on performance and energy efficiency. It also examined the introduction of the special operand queue, which was intended to reduce the need to use the limited number of registers to hold temporary values. This was shown to have a positive effect on both performance and energy efficiency.

Furthermore, the chapter compared the performance and energy efficiency of three schemes:

- a scheme where registers are mapped to memory through a pointer,
- a scheme which implemented register windows through separate instructions
- a register window scheme used in the SPARC architecture.

The results suggest that the first two schemes are less performance- and energy-efficient than the SPARC scheme due to the increased instruction count. However, the first two schemes ensure a more constant performance, which might be essential for some applications, as the register-file in these schemes cannot overflow. Of the first two schemes, it was demonstrated that the memory mapped scheme was more energy efficient than the conceptually simpler spill/fill scheme, due to lower instruction count and fewer accesses to the data cache. This, despite the larger and hence more energy consuming register ‘file’.

Chapter 8, *‘Instruction fetching’* addressed the issue of variable-size instructions. It was shown that reducing the average size of instructions has a positive effect on instruction cache performance, but can reduce the overall performance of the processor if the issue of instructions straddling cache lines is not addressed. The chapter proposed three instruction cache architectures:

- The Alignment architecture
- The Dual Cache Line (DCL) architecture
- The eXtra-line architecture

These can almost eliminate the problem, and the instruction format therefore affects the performance positively due to an improved hit rate in the instruction cache. Furthermore the suggested architectures have the effect of significantly reducing the number of references and thus the energy consumption in the cache. Consequently the variable-size instruction format is considered both performance and energy efficient. Furthermore, the ratio of the power consumption in the cache to the total power consumption of the entire system is reduced, implying that further improvements in energy efficiency should be obtained by tuning other parts of the architecture.

It was demonstrated that of the three architectures the DCL-architecture produced the lowest the number of accesses to the cache, but the energy consumption of individual requests and the cycle time of the eXtra-line architecture is expected to be lower and the energy efficiency consequently better.

Chapter 9 discussed the optimal cache configuration for an energy efficient implementation of the HORN architecture. Numerous cache configurations were simulated and the power consumption, performance and energy efficiency for each configuration was computed. The results showed that the most energy efficient cache configurations are smaller than the best performing configurations, but even more significantly, they have shorter cache lines both in the instruction and in the data cache. Furthermore, the best performing configurations are direct mapped for all the benchmarks, while the most energy-efficient configurations for some of the benchmarks are skewed-associative. As Chapter 8 showed how the number of references to the instruction cache was reduced significantly (88% for long 64-byte cache lines) Chapter 9 assessed an architecture where both the instruction and data streams were fed from one cache without most of the performance penalty of the conventional unified cache. However this architecture did not perform as well as the separate caches nor was it as energy efficient.

10.2 Assessment of work

The work reported in this thesis has been theoretical. No hardware has been implemented nor have any low-level transistor models been developed. The results are therefore based upon a number of extrapolations from other processor designs such as the ARM3 and upon numerous simulators developed either specifically for this project or for commercially available products. The validity of these extrapolations and especially of the use of multiple extrapolations in the same expression may be questioned; only an implementation of the suggested architectures can provide a definite answer.

Most previous research into energy efficient computer architecture has explored subsystems, notably caches. This thesis has considered a whole processor system and has highlighted the tension between performance and energy efficiency at that level. The project has successfully identified a number of features which future computer architectures concerned with energy efficiency can exploit:

- Two instruction branch structures such as the ones described for the HORN architecture may improve performance but degrade the energy efficiency. If instructions are not prefetched into a shadow pipeline, the single instruction branch used in most RISCs performs better and is more energy efficient.
- The use of a queue for storage of temporary results allows the semantic content of instructions to be coded in less space. Instruction sizes are reduced without increasing the total number of instructions in a program. This increases the performance of the instruction cache and thus the energy efficiency of the entire processor system.
- The DCL and eXtra-line cache architectures reduce the energy consumption in caches significantly, without affecting the performance negatively. They also provide effective solutions to assembling instructions which straddle cache lines. The combination of the variable-size instructions and these cache architectures thus provides a feature that future RISC architectures should exploit.
- The register file schemes which have been proposed for the HORN architecture have been shown *not* to perform as well as the established overlapping register window architecture used in SPARC, due to the increased instruction count. However, the work has shown that a compiler can exploit a variable-size overlap of register windows and thus yield a better utilization of the available on-chip storage, reducing the number of register window overflows. The performance and energy efficiency would consequently

improve. This could be investigated further, but it should be emphasized that the specification of the size of the overlap should be in the call /return instructions rather than in a separate instruction.

- The results do not point to any ‘golden’ cache configurations. The most energy efficient cache configurations have a higher degree of associativity and they are often smaller than the best performing configurations. The graphs in chapter 9 have shown that there may be a significant variation in energy efficiency even within a narrow band of performance.

10.3 Conclusions

This work has shown that the energy efficiency of a microprocessor is affected by early decisions in the specification of a processor architecture such as the instruction set. It is therefore clear that future processors concerned with energy efficiency should optimize for this metric in all the stages of the specification and implementation processes. ‘Performance against power consumption’ graphs with constant energy-efficiency levels shown are useful when carrying out such optimizations as they clearly show how architectural changes affect all three measures.

Optimizing for energy efficiency may result in a similar architecture as when optimizing for performance but while optimizing for performance tends to increase power consumption, optimizing for energy efficiency will tend to keep the power consumption down. Given a power budget below that required for optimal performance, the energy-efficiency metric should point to a configuration/architecture which is not the best performing, but where the amount of computation per energy unit is highest.

As implementation technologies are expected to keep improving and allow processors to be clocked with ever increasing frequencies and hand-held and portable equipment is expected to operate longer, battery technologies will be put under ever increasing pressure. Optimizing for energy efficiency throughout the process of specifying and implementing a microprocessor will ensure the best compromise between high performance and long battery life. This work has successfully proposed a number of ways to improve the energy efficiency of a processor and has identified schemes which may increase the performance but which have a negative effect on the energy efficiency.

10.4 Suggestions for future work

There is still considerable scope for research into energy efficient computer architectures and much work can be done to develop further the ideas presented here. Furthermore the following areas are suggested as fruitful topics for future investigation:

- **Block buffers in the data cache:** The simulations in Chapter 9 assumed a write-through strategy in the block buffer of the data cache. Changing the write-through strategy to copy-back will reduce the energy consumption in the data cache further due to a reduced number of writes to the cache memory. In order to avoid a negative effect on performance the dirty block buffer should be copied to a new 'writeback-buffer' while a new line is fetched from the cache memory into the block buffer. The contents of the writeback-buffer could then be written back to the cache in the following cycle. The performance level of the write-through scheme is thus retained.
- **The data path**: This work has shown that some improvement in energy consumption can be obtained if cache lines are fetched from and written to external memory in a Gray-coded order, rather than the traditional sequential order. This can be explored further if a compiler can be developed which can lay-out code and data in such a way that the number of bit-transi-

tions on the data bus is minimized. Careful allocation of opcodes might also reduce the number of bit-transitions. These changes would not only reduce the power consumption in the I/O system, but also on the data-path within the processor. Without further work it is not clear if this is feasible.

- **The pipeline**: This work has not determined whether pipelining is energy efficient. Increasing the pipeline length might allow an increase in the clock frequency of the processor and thereby the peak-performance. However, given that the penalty from branches and register dependencies might increase with a longer pipeline and the energy consumption per instruction thus increase, the effect on the energy efficiency is not clear.
- If the effect of increasing pipeline length is to decrease energy efficiency, replicating the structures suggested in Chapter 8 should be considered. This will allow prefetching of instructions as described in Chapter 6 without increasing the energy consumption. This would eliminate the branch penalty completely and thus improve both performance and energy efficiency.
- **Unified cache**: The architecture evaluated in section 9.3 can be expanded to serve data references through a block buffer at the same time as instructions as served from the eXtra-line. This will reduce the number of references to the cache further and hence yield higher energy-efficiency measures. The work should establish if a unified configuration can yield energy-efficiency levels comparable with those found for separate caches.

Finally, many of the results presented in this thesis have been based upon simulations and extrapolations. Implementation of the cache architectures proposed in Chapter 8, would verify the extrapolations and allow more precise models to be written for the use of future work.

It is the author's hope that the work reported in this thesis will help designers of future microprocessors to improve the energy efficiency of their products.

This is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.

Winston Churchill

References

[Argade] P.V Argade et al., “Hobbit: A High-Performance, Low Power Microprocessor”, Proceedings of CompCon ‘94 pp. 88-95, IEEE Computer Society Press, San Francisco, March 1994

[Bensch] B.J. Benschneider et.al, “A 300-MHz 64-bit Quad-Issue CMOS RISC Microprocessor”, IEEE Journal of Solid State Circuits, Vol. 30 No 11 November 1995

[Biggs] T. Biggs et al., “A 1 Watt 68040-Compatible Microprocessor”, Proceedings of IEEE Symposium on Low Power Electronics, San Diego, 1994

[Bird] P.L.Bird, U.W.Pleban, N.P Topham and Henrik Scheuer: “Semantics Driven Computer Architecture”, Parallel Computing’91, Ed. D.Evans, G. Joubert and H.Liddell, North Holland, 1992

[Bird2] “The Interaction of Compilation Technology and Computer Architecture”, Editor D. Lilja and P.L.Bird, Kluwer Academic Publisher

[Bodin] F. Bodin, A. Seznec, “Skewed associativity enhances performance predictability”, IRISA, Campus Universitaire de Beaulieu, 35042 Rennes, France; Internal Publication No. 909

[Bunda] J. Bunda, D. Fussell, W. C. Athas, “Energy-Efficient Instruction Set Architecture for CMOS Microprocessors”, Proceedings of the 28th Annual Hawaii International Conference on System Sciences, 1995, pp 298-305

[Burd] T.D.Burd and R.W. Brodersen, “Energy Efficient CMOS Microprocessor Design”, Proceedings of the 28th Annual Hawaii International Conference on System Science, 1995, vol.1. page 288-297, IEEE Computer Society Press.

[Burd2] Tom Burd, “Low-Power CMOS Library Design Methodology”, M.Sc Thesis; Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1993

[BurdPeters] T.Burd and et.al, “A Power Analysis of a Microprocessor: A Study of the MIPS R3000 Architecture”. Technical report, University of California, Berkeley, available on the internet: http://infopad.eecs.brekeley.edu/~burd/gpp/r3000/total.html

[Case] S.Case, “Low-End PA7100LC Adds Dual Integer ALUs”, Microprocessor Report, November 18, 1992

[Case2] B. Case, “IBM Delivers First PowerPC MicroProcessor”, Microprocessor Report, Volume 6, Number 14, October 28, 1992

[Child] Mark Child et. al. “First Looks, 200MHz Pentium PCs: Ultimate speed” PC Magazine, August 1996, Vol. 5 Issue 8.

[Culbert] M. Culbert, “Low Power Hardware for a High Performance PDA”, Proceedings of CompCon’94 pp. 144-147, IEEE Computer Society Press, San Francisco, March 1994

[Conte] T.M. Conte, K.N. Menezes, P.M. Mills, B.A. Patel, “Optimization of Instruction Fetch Mechanisms for High Issue Rates”, Proceedings of ISCA ‘95 pp 333-344.

[DEC21064] DECchip 21064-AA Microprocessor, “Hardware Reference Manual”, Order Number: EC-N0079-72, Digital Equipment Corporation, Maynard, Massachusetts, USA.

[DeRosa] J.A. DeRosa and H.M. Levy, “An Evaluation of Branch Architectures”, ACM 0084-795/87/0600-0010, 1987

[Farquhar] E.Farquhar and P. Bunce, “The MIPS Programmer’s Handbook”, Morgan Kaufmann, San Francisco, CA, 1994, ISBN 1-55860-297-6

[Fleet] P. Fleet, “The SH Microprocessor: 16-Bit Fixed Length Instruction Set Provides Better Power and Die Size”, Proceedings of IEEE, Compcon’93

[Furber] S.B.Furber, “ARM Systems Architecture”, Chapter 7: “The Thumb Instruction Set”, Addison-Wesley

[Furber2] S.B. Furber et al., “AMULET1: A Micropipelined ARM”, Proceedings of CompCon’94, IEEE Computer Society Press, San Francisco, March 1994

[Garside] Personal communication with Dr. J. D. Garside, Manchester University, Department of Computer Science, 10th of October, 1995.

[Garside2] J.D Garside et al., “The AMULET2e Cache System”, Proceedings: Async’96, Aizu-Wakamatsu, March, 1996

[Gerosa] G.Gerosa et.al, “A 2.2W, 80MHz Superscalar RISC Microprocessor”, IEEE Journal of Solid-State circuits, vol. 29, December 1994

[Hilditch] S. Hilditch and S. Furber: “Hash Cache: Fully-Associative Cache Performance from a 4-way Associative cache”, 'Internal unpublished report', Manchester University Department of Computer Science

[Hill] Mark D. Hill et al. “Design decisions in SPUR”, Computer, pages 8-22 November 1986

[Hitachi] Hitachi IC Memory Data Book, 1989

[HORNV3] SGS-Thomson Microelectronics, Chameleon Architecture Manual, 3rd Edition, March 1994; INMOS Document Number: 72-TRN-253-02, Available under Non-Disclosure Agreement

[HORNV5] SGS-Thomson Microelectronics, Chameleon Architecture Manual, 5rd Edition, October 1994; INMOS Document Number: 72-TRN-253-04, Available under Non-Disclosure Agreement

[IBM] IBM Corporation. “IBM RT PC Hardware Technical Reference, Volume 1.” IBM, September 1986, SV21-8024

[Johnson] Mike Johnson, “Superscalar Microprocessor design”, Prentice Hall Inc., 1991, ISBN 0-13-875634-1.

[Juan] T.Juan et.al., ”The Difference-bit Cache”, Proceedings of 23rd International Symposium on Computer Architecture, Pennsylvania, 1996, pp 114-120.

[Katevenis] M.G.H Katevenis, “Reduced Instruction Set Computer Architectures for VLSI”. The MIT Press, Cambridge, Massachusetts, 1985

[Kohavi] Z. Kohavi, “Switching and Finite Automata Theory”, New York, McGraw-Hill, 1970

[Lev] L.A.Lev et al., “A 64-b Microprocessor with Multimedia Support”, IEEE Journal of Solid-State Circuits. vol. 30, No. 11, November 1995

[Mahon] M.J. Mahon et.al., “Hewlett-Packard Precision architecture” Hewlett-Pacard Journal, No. 37, p. 4-22, August 1986

[Mead] C. Mead and L. Conway, “Introduction to VLSI systems”, Addison Westley, 1980

[MHill] Personal communication with Mark B. Hill, PACT, at HORN-meeting, 21st and 22nd of March, 1996 in Karlsruhe

[MRP1092] “Hobbit Enables Personal Communicators”, Microprocessor Report, October 28, 1992

[Mulder] H. Mulder and M.J. Flynn, “Processor Architecture and Data Buffering”, IEEE Transaction on Computers Vol. 41 No 10, October 1992.

[Okada] T. Okada et.al, “A PA-RISC Microprocessor PA/50L For Low-Cost Systems”, Proceedings of IEEE CompCon’94, pp 47-52, IEEE Computer Society Press, San Francisco, March 1994

[OMIMAP] Manchester University with contributions from Advanced RISC Machines, “Low Power Technologies - Preliminary Report”, January 1992, OMI/MAP P5386 Deliverable 4.2.1.

[Patt] D.A Patterson, J. L. Hennesey, “Computer Architecture A Quantitative Approach”, Morgan Kaufman Publishers Inc., 1990

[R4300i] World Wide Web page for the R4300i processor: http://www.mips.com/r4300i/Prod_Overview.book.html#prefetching

[Robin] P.R. Robinson, “Mastering the 68000 Microprocessor”, TAB Books Inc, No 1886, 1985, ISBN 0-8306-1886-4

[RS6000] “The PowerPC Architecture: A Specification for a New Family of RISC Processors” Second Edition, Morgan Kaufmann Publishers, ISBN 1-55860-316-6

[Ruegg] J.Ruegg, Sozobon Limited, 1991. Public domain software available from the author.

[RYork] Personal communication with Mr. R. York, ARM Ltd. Friday, April 5th 1996.

[RYork2] Richard York, “Branch Prediction Strategies for Low Power Microprocessor Design”, M.Sc-thesis. Department of Computer Science, Manchester University, 1994

[Sato] T. Sato et al., “Power and Performance Simulator: ESP and its Application for 100MIPS/W Class RISC Design”, IEEE Symposium on Low Power Electronics, San Diego, 1994

[Segar] S.Segar et al, “Embedded Control Problems, Thumb, and the ARM7TDMI”, IEEE Micro October 1995, page 22 - 30.

[Seznec] A. Seznec, “A case for two-way skewed-associative caches”, Proceedings of 20th International Symposium on Computer Architecture, San Diego, 1993

[Seznec2] A Seznec, “Decoupled Sectored Caches: Conciliating low tag implementation cost and low miss ration”, Proceedings of 21st International Symposium on Computer Architecture, Chicago, 1994

[Shade] SPARC Performance Analysis Tools - Shade User’s Manual, 4th Edition, Sun Microsystems Laboratories Inc, 1992

[Singh] J.P. Singh, W.D. Weber and A. Gupta, "Splash: Stanford Parallel Applications for Shared Memory", Technical Report, Computer Systems Laboratory, Stanford University, 1991

[Slater] M.Slater, "AT&T Sampling Low-Power "Hobbit" processor", Microprocessor Report, Volume 6 Number 2, February 12, 1992

[SPEC89] SPEC, "SPEC Benchmark Suite Release 1.0," October 2, 1989

[SPEC91] SPEC, "The SPEC Benchmark Suite". SPEC Newsletter 3, p3-4, 1991.

[Sprack] L. Spracklen, "Z80 and 8080 Assembly Language Programming", Hayden Book Company INC, ISBN: 0-8104-5167-0

[Su] C.L. Su, A. M. Despain, "Cache Designs for Energy Efficiency", Proceedings of the 28th Annual Hawaii International Conference on System Sciences, 1995

[Suessmith] B.W. Suessmith and George Paap III, "PowerPC 603 Microprocessor Power Management", Communications of the ACM, June 1994, pp 43-46

[Thakker] S.S. Thakker, "A High Performance Virtual Memory Management Unit for a supermini computer", Ph.D Thesis, Department of Computer Science, Manchester University, April 1982

[Tiwari] V. Tiwari et al. "Compilation Techniques for Low Energy: An Overview", IEEE Symposium on Low Power Electronics, San Diego, 1994

[TOSHIBA] Information about Toshiba's Dynamic RAM products can be found on the internet: http://www.toshiba.com/taec/components/mem1.html

[TOSHIBA2] Information about Toshiba's Static RAM products can be found on the internet: http://www.toshiba.com/taec/components/mem4.html

[Transputer] INMOS, "The Transputer Databook", Second Edition, 1989; INMOS document number: 72-TRN-203-01.

[Uhlig] R.Uhlig, et al., “Instruction Fetching: Coping with Code Bloat”, Proceedings of 22th International Symposium on Computer Architecture, Santa Margherita, Ligure, Italy, 1995, pp 345-355

[VLSI] VLSI Technology INC., “1-micron cell compiler library, Rev. 2.0 April 1991”, sections: ‘CRAM02’ and ‘VDPRAM300’

[VLSI2] VLSI Technology INC, VSC350 Library, April 1991.

[Wang] H. Wang, T Sun and Q Yang, “CAT - Cache Address Tags - A Technique for Reducing Area Cost of On Chip Caches”, Proceedings of 22th International Symposium on Computer Architecture, Santa Margherita, Ligure, 1995

[Weaver] D.L Weaver and T. Germond, “Sparc Architecture Manual”, Prentice Hall, 1994, ISBN: 0-138-250-014

[Weicker] R.P. Weicker, “Dhrystone: A Synthetic Systems Programming Benchmark”. Communications of the ACM 27(10): 1013-1030, October 1984

[Weste] N. H. E. Weste & K. Esthraghian, “Principles of CMOS VLSI Design, A system perspective”, 2nd Edition, Addison Westley, 1985

[Wharton] J.H Wharton, “Intel Unveils Radical New CPU family”, Microprocessor Report, Volume 2 Number 4, April 1988

[Williams] T. Williams, N.Patkar, G.Sheen, “SPARC64: A 64-b 64-Active-Instruction Out-of-Order-Execution MCM Processor”, IEEE Journal of Solid-State Circuits. vol. 30, NO. 11, November 1995

[Wilton] S.J.E Wilton and N.P. Jouppi; “An Enchanced Access and Cycle Time Model for On-Chip Caches”, Research Report 93/5, Digital Equipment Coperation, Western Research Laboratory, Palo Alto, California, USA

[Yeung] N. Yeung, B. Zivkov, G. Ezer, “Unified Datapath: An Innovative Approach to the Design of a Low-Cost, Low-Power, High Performance Microprocessor”, Proceedings of CompCon 1994, IEEE Computer Society Press, San Francisco, March 1994

[Zivkov] B. Zivkov, B. Ferguson and M. Gupta, “R4200: A High-Performance MIPS Microprocessor for Portables”. Proceedings of CompCon’94 page 18-25, IEEE Computer Society Press, San Francisco, March 1994

[Z80] “Z80 Microprocessor Family User’s Manual”, Zilog Inc.

Appendix A Energy Efficiency versus power consumption

A.1 Separate caches

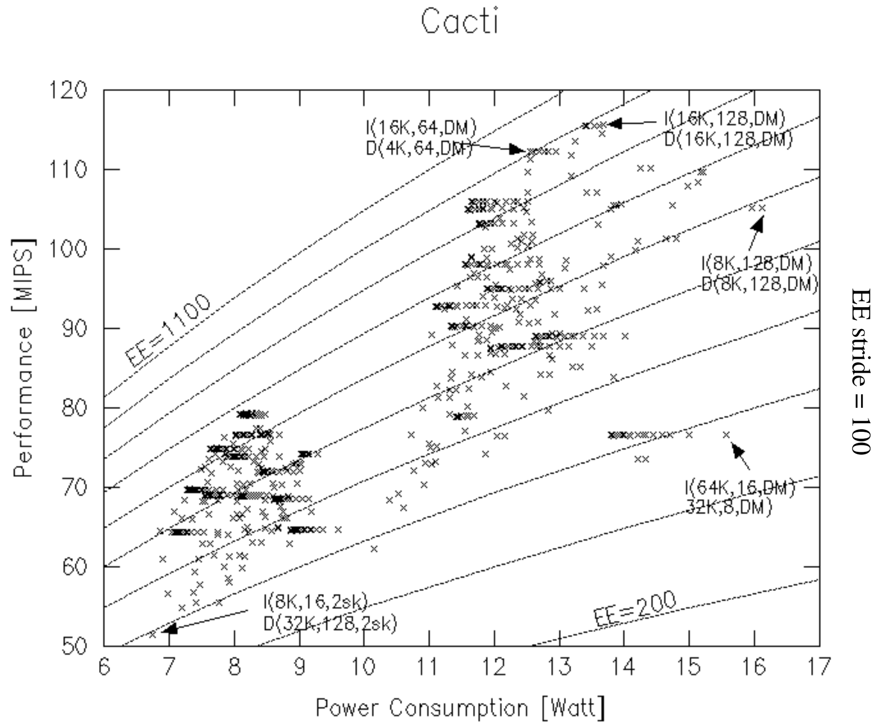


Figure A.1 EE and performance versus power consumption, cacti

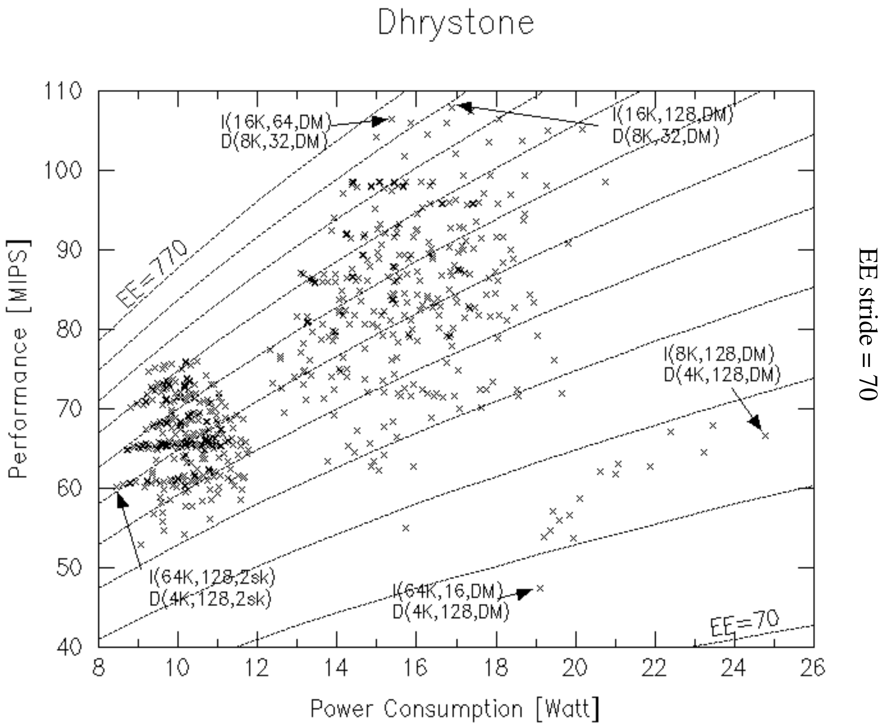


Figure A.2 EE and performance versus power consumption, dhrystone

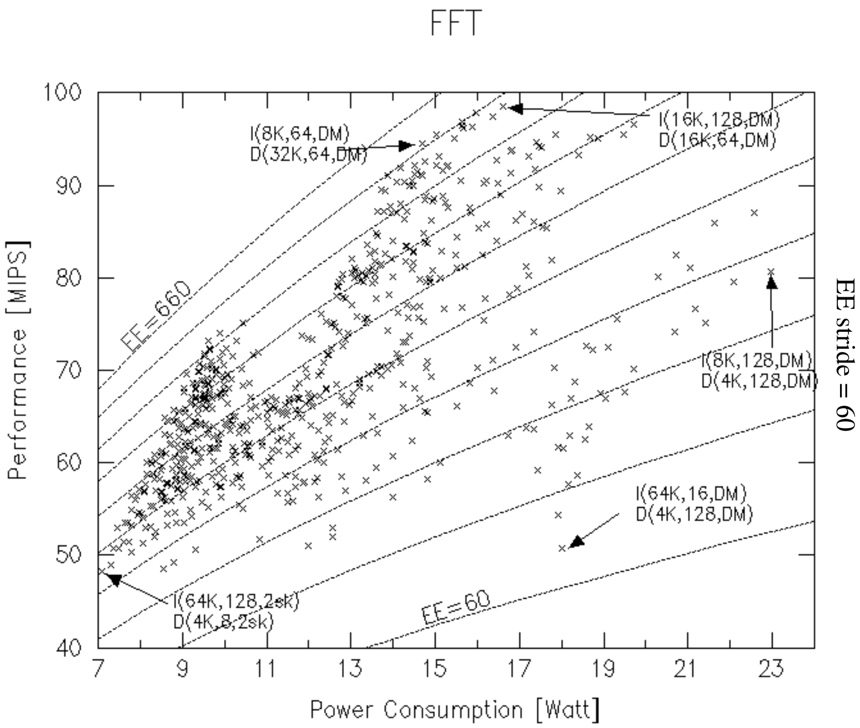


Figure A.3 EE and performance versus power consumption, fft

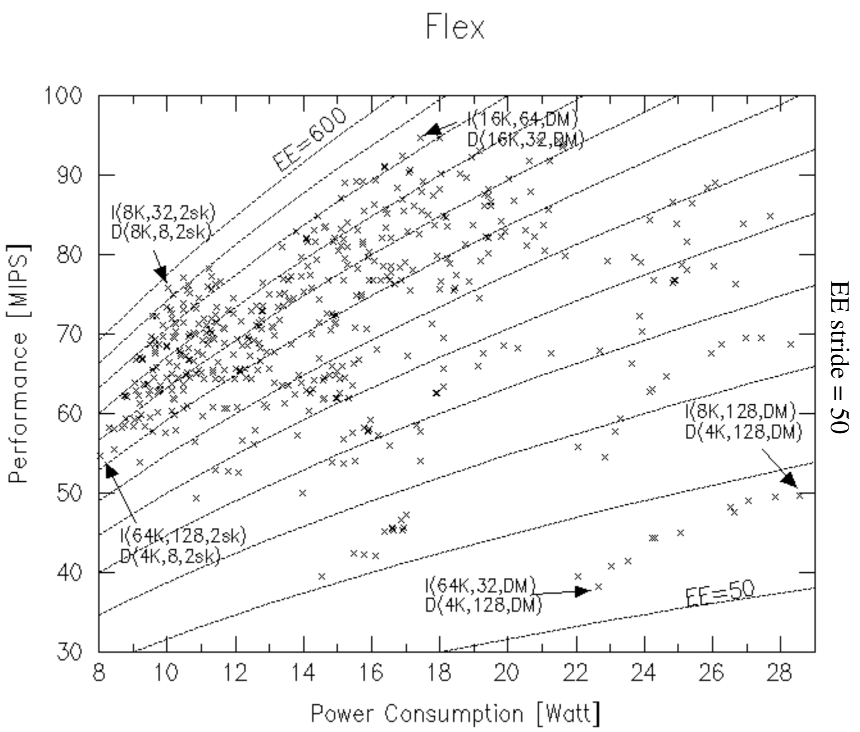


Figure A.4 EE and performance versus power consumption, flex

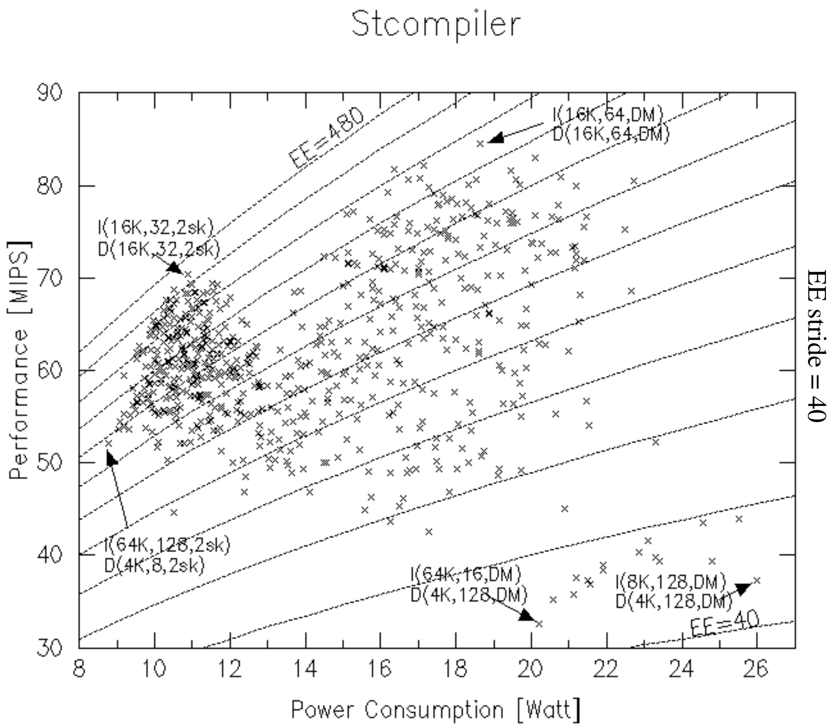


Figure A.5 EE and performance versus power consumption, stcompiler

A.2 Unified cache

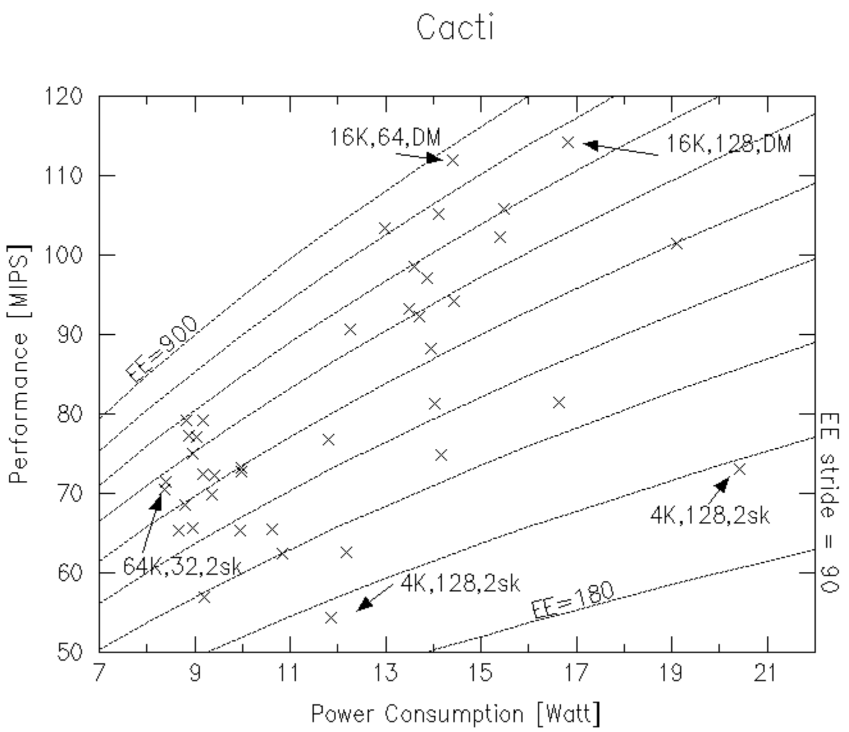


Figure A.6 EE and performance versus power consumption, cacti

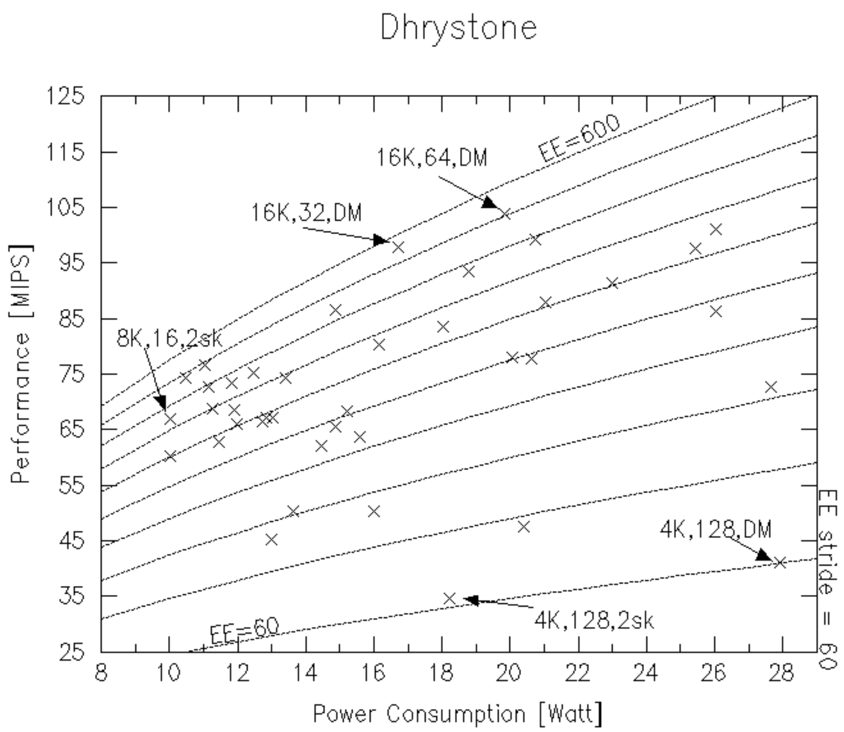


Figure A.7 EE and performance versus power consumption, dhrystone

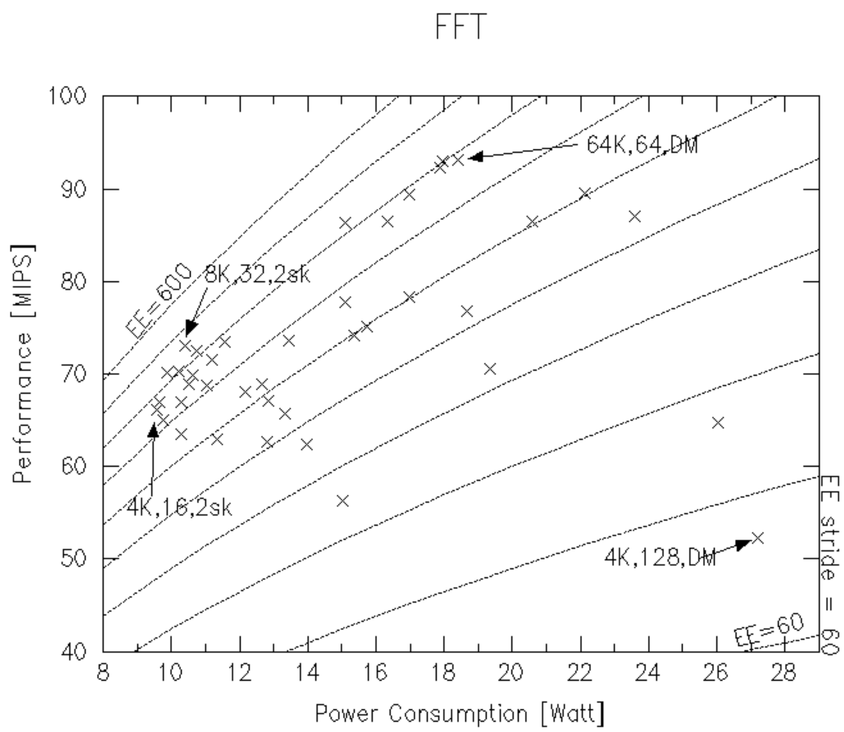


Figure A.8 EE and performance versus power consumption, fft

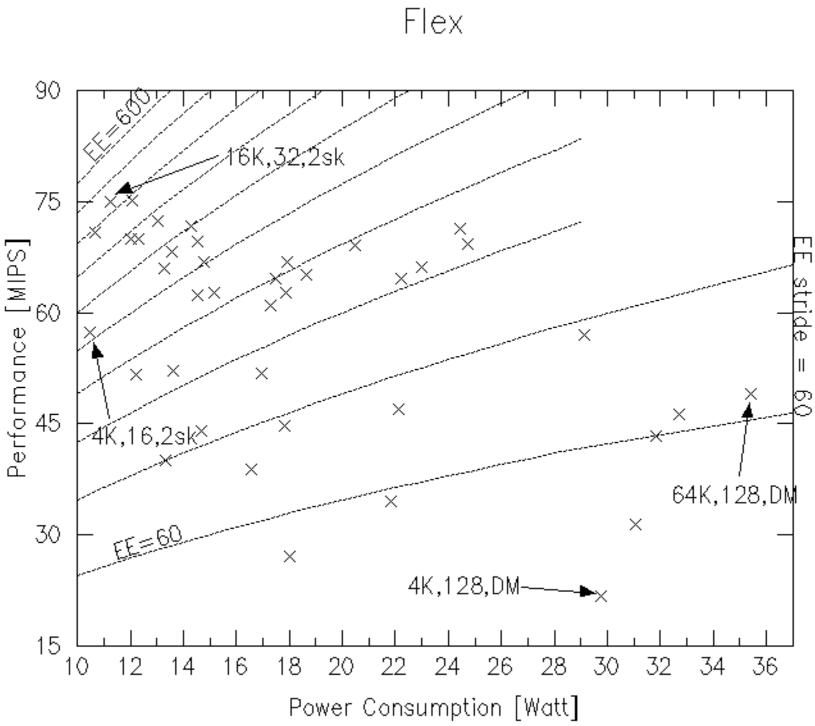


Figure A.9 EE and performance versus power consumption, flex

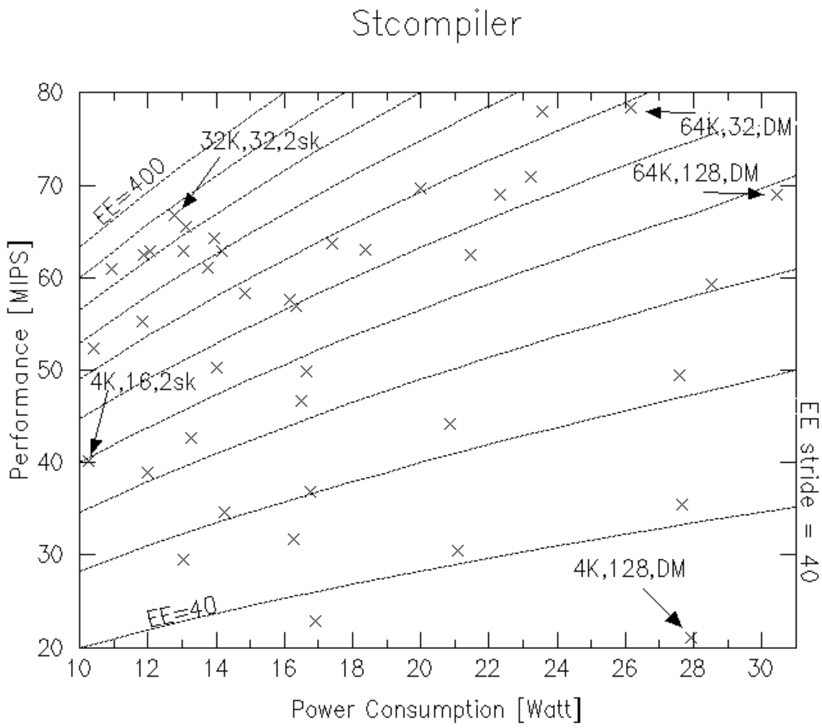


Figure A.10 EE and performance versus power consumption, stcompiler

Appendix B Energy Efficiency versus cache line size

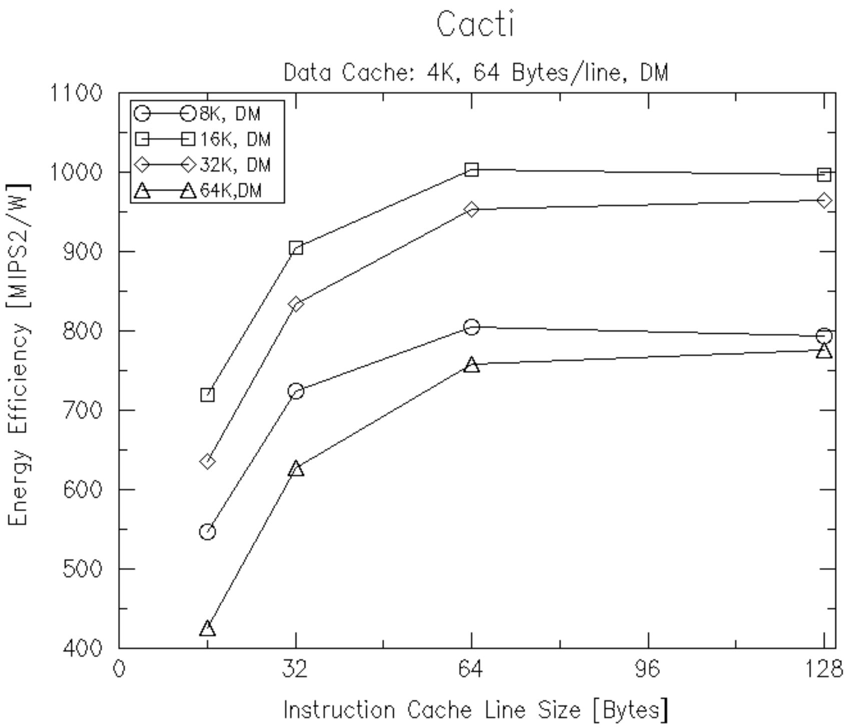


Figure B.1 Energy Efficiency versus instruction cache line size, cacti

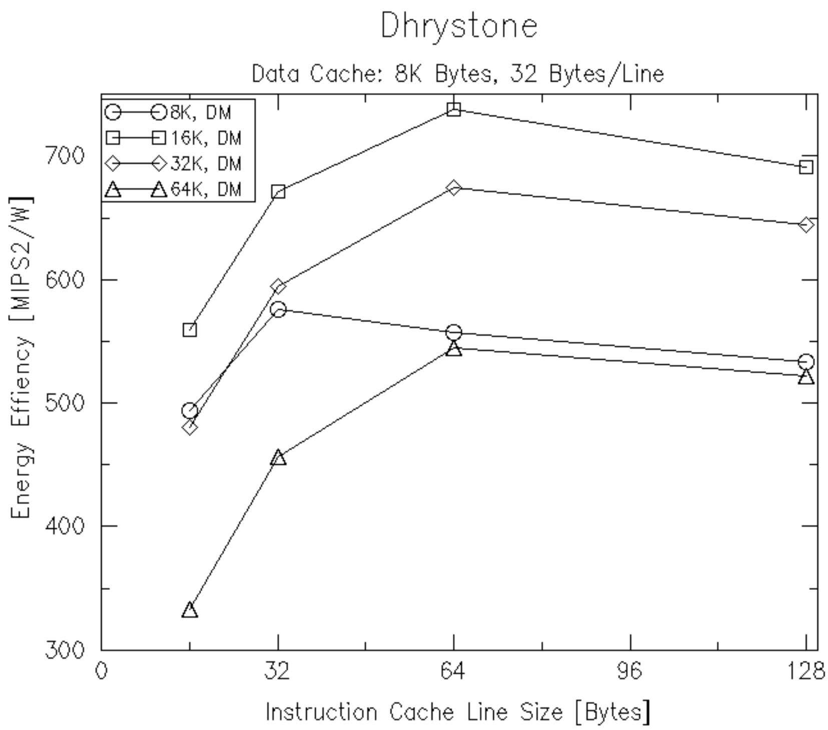


Figure B.2 Energy Efficiency versus instruction cache line size, dhrystone

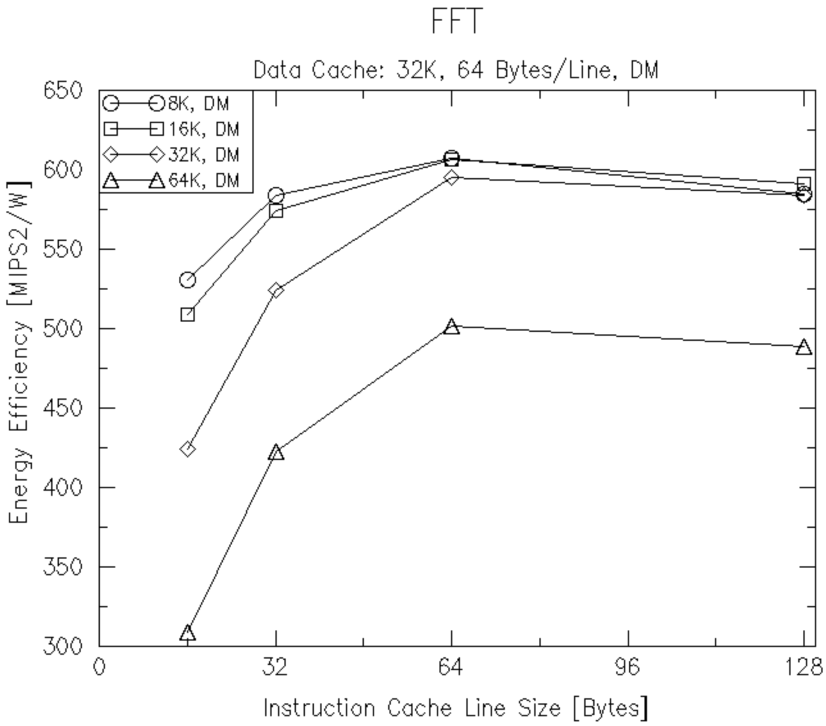
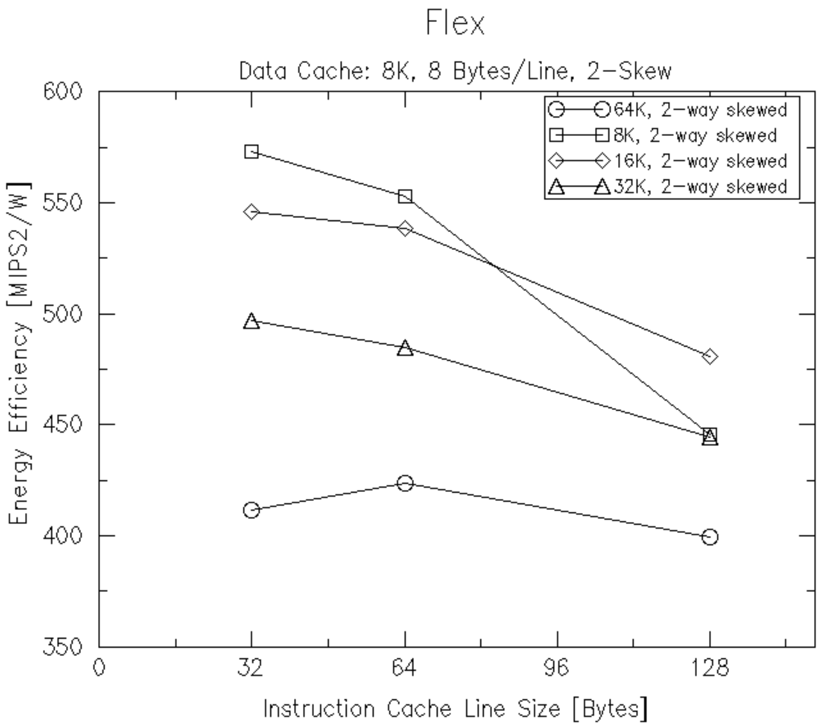


Figure B.3 Energy Efficiency versus instruction cache line size, fft



Note, measurements for 16 byte lines are missing due to software problems

Figure B.4 Energy Efficiency versus instruction cache line size, flex

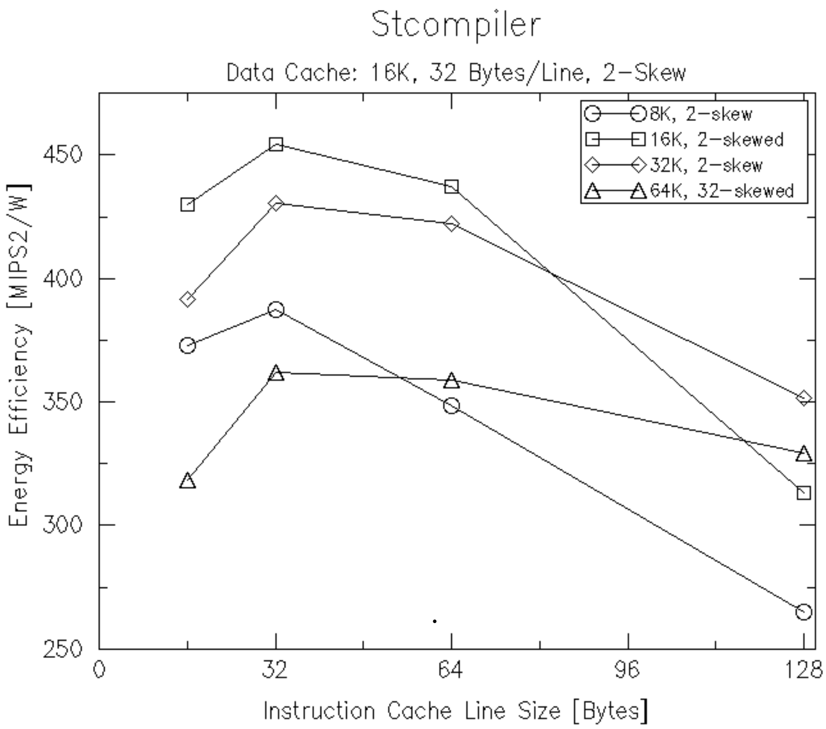


Figure B.5 Energy Efficiency versus instruction cache line size, stcompiler

Appendix C Simulation results

This appendix presents the results of the simulations described in section 9.2. Due to the large number of configurations and hence results to report only the results for one of the benchmarks, espresso, will be shown here. The results for the remaining benchmarks are available from the author.

Table C.1 Cache configuration measurements, espresso

Instruction cache configuration [total size, line size, associativity]	Data cache configuration [total size, line size, associativity]	System Power [Watt]	Performance [MIPS]	Energy Eff. [MIPS^2/W]	T_{Cycle} [ns]	P_{ram}/P_{total} [%]	P_{proc}/P_{total} [%]	P_{cache}/P_{total} [%]
8K,16,DM	4K,8,DM	11.9	70.6	418.7	7.8	5.9	62.8	31.3
8K,16,DM	4K,16,DM	12.6	71.9	409.9	7.8	7.1	59.2	33.7
8K,16,DM	4K,32,DM	13.9	69.0	341.8	7.8	9.4	53.6	37.0
8K,16,DM	4K,64,DM	16.6	60.4	219.6	7.8	13.5	44.9	41.6
8K,16,DM	4K,128,DM	21.8	45.2	93.7	7.8	18.2	34.3	47.5
8K,16,DM	8K,8,DM	12.1	76.8	487.7	8.3	4.5	57.9	37.6
8K,16,DM	8K,16,DM	13.2	80.2	485.4	7.8	5.2	56.4	38.4
8K,16,DM	8K,32,DM	14.3	77.5	419.3	7.8	6.7	52.2	41.2
8K,16,DM	8K,64,DM	16.7	70.5	297.4	7.8	9.6	44.7	45.7
8K,16,DM	8K,128,DM	21.7	58.3	156.6	7.8	13.5	34.5	52.0
8K,16,DM	16K,8,DM	12.2	73.3	442.7	9.3	3.8	51.6	44.6
8K,16,DM	16K,16,DM	13.2	79.2	474.6	8.6	3.9	51.5	44.6
8K,16,DM	16K,32,DM	14.8	83.7	472.8	8.0	4.1	49.2	46.7
8K,16,DM	16K,64,DM	16.9	84.9	425.6	7.8	4.3	44.1	51.6
8K,16,DM	16K,128,DM	21.1	79.9	302.3	7.8	5.7	35.4	58.9
8K,16,DM	32K,8,DM	13.0	68.3	357.4	10.4	3.0	43.0	54.0
8K,16,DM	32K,16,DM	13.9	74.6	399.4	9.4	3.1	44.3	52.6
8K,16,DM	32K,32,DM	15.0	78.4	409.2	8.9	3.1	43.6	53.3
8K,16,DM	32K,64,DM	16.8	81.3	393.0	8.6	2.8	40.4	56.8
8K,16,DM	32K,128,DM	19.4	80.3	332.4	8.7	2.4	34.6	63.0
8K,16,2-skew	4K,8,2-skew	9.2	72.6	571.4	9.8	3.6	64.7	31.8
8K,16,2-skew	4K,16,2-skew	9.7	74.9	576.3	9.8	3.5	61.3	35.2
8K,16,2-skew	4K,32,2-skew	10.7	74.3	517.2	9.8	4.1	55.9	40.1
8K,16,2-skew	4K,64,2-skew	12.1	71.1	417.1	10.1	4.5	47.5	47.9
8K,16,2-skew	4K,128,2-skew	14.3	55.2	213.1	11.6	8.3	35.2	56.5
8K,16,2-skew	8K,8,2-skew	9.4	73.7	578.9	10.0	3.1	62.4	34.5
8K,16,2-skew	8K,16,2-skew	10.0	76.2	582.3	9.8	3.0	59.8	37.1
8K,16,2-skew	8K,32,2-skew	10.6	75.5	537.5	10.0	2.9	55.1	41.9
8K,16,2-skew	8K,64,2-skew	11.8	72.0	440.8	10.4	3.0	47.6	49.4
8K,16,2-skew	8K,128,2-skew	13.2	63.3	302.5	11.8	3.1	37.4	59.5

Table C.1 Cache configuration measurements, espresso

Instruction cache configuration [total size, line size, associativity]	Data cache configuration [total size, line size, associativity]	System Power [Watt]	Performance [MIPS]	Energy Eff. [MIPS^2/W]	T_{Cycle} [ns]	P_{ram}/P_{total} [%]	P_{proc}/P_{total} [%]	P_{cache}/P_{total} [%]
8K,16,2-skew	16K,8,2-skew	9.1	69.6	532.8	11.0	2.5	58.4	39.1
8K,16,2-skew	16K,16,2-skew	9.8	73.5	550.6	10.4	2.4	57.0	40.7
8K,16,2-skew	16K,32,2-skew	10.2	71.6	502.1	10.7	2.3	53.2	44.5
8K,16,2-skew	16K,64,2-skew	11.2	69.4	429.7	11.1	1.9	46.7	51.4
8K,16,2-skew	16K,128,2-skew	12.7	63.4	316.5	12.2	1.8	37.7	60.6
8K,16,2-skew	32K,8,2-skew	9.3	64.0	441.3	12.1	2.2	52.1	45.7
8K,16,2-skew	32K,16,2-skew	9.7	67.1	462.5	11.5	2.1	52.0	45.9
8K,16,2-skew	32K,32,2-skew	10.3	67.9	445.2	11.4	2.0	49.4	48.6
8K,16,2-skew	32K,64,2-skew	11.0	64.2	376.2	12.1	1.9	44.1	54.1
8K,16,2-skew	32K,128,2-skew	12.2	59.0	284.4	13.2	1.5	36.1	62.4
8K,32,DM	4K,8,DM	12.3	74.8	453.7	7.7	7.0	61.7	31.2
8K,32,DM	4K,16,DM	13.9	81.3	475.8	7.2	7.9	58.3	33.8
8K,32,DM	4K,32,DM	15.4	77.9	395.1	7.2	10.2	52.7	37.2
8K,32,DM	4K,64,DM	18.3	67.8	250.8	7.2	14.0	44.2	41.8
8K,32,DM	4K,128,DM	23.9	50.2	105.5	7.2	18.5	33.8	47.7
8K,32,DM	8K,8,DM	12.4	82.0	541.8	8.3	5.4	56.3	38.3
8K,32,DM	8K,16,DM	13.6	85.9	540.5	7.8	6.2	54.8	39.1
8K,32,DM	8K,32,DM	15.9	88.5	492.8	7.2	7.5	50.9	41.6
8K,32,DM	8K,64,DM	18.6	80.0	345.0	7.2	10.3	43.6	46.1
8K,32,DM	8K,128,DM	24.0	65.5	178.6	7.2	13.9	33.7	52.4
8K,32,DM	16K,8,DM	12.5	78.3	489.6	9.3	4.6	50.0	45.4
8K,32,DM	16K,16,DM	13.7	84.9	527.9	8.6	4.8	49.8	45.4
8K,32,DM	16K,32,DM	15.4	90.0	527.3	8.0	5.0	47.5	47.6
8K,32,DM	16K,64,DM	18.0	92.7	477.0	7.6	5.2	42.5	52.3
8K,32,DM	16K,128,DM	22.7	87.7	338.2	7.5	6.5	34.0	59.5
8K,32,DM	32K,8,DM	13.5	72.8	392.0	10.4	3.7	41.4	54.9
8K,32,DM	32K,16,DM	14.5	79.8	440.9	9.4	3.8	42.7	53.5
8K,32,DM	32K,32,DM	15.6	84.2	453.7	8.9	3.8	41.9	54.3
8K,32,DM	32K,64,DM	17.6	87.3	434.8	8.6	3.4	38.7	57.9
8K,32,DM	32K,128,DM	20.4	86.3	366.2	8.7	3.0	33.0	64.0
8K,32,2-skew	4K,8,2-skew	9.3	75.0	604.2	10.0	4.2	62.9	32.9
8K,32,2-skew	4K,16,2-skew	9.8	77.5	609.8	10.0	4.1	59.5	36.4
8K,32,2-skew	4K,32,2-skew	10.8	76.8	545.9	10.0	4.7	54.1	41.2
8K,32,2-skew	4K,64,2-skew	12.5	74.7	446.1	10.1	5.0	46.0	49.0
8K,32,2-skew	4K,128,2-skew	14.9	57.2	219.9	11.6	8.8	33.9	57.3
8K,32,2-skew	8K,8,2-skew	9.7	77.7	625.1	10.0	3.6	60.6	35.7
8K,32,2-skew	8K,16,2-skew	10.1	78.9	616.8	10.0	3.6	58.0	38.3
8K,32,2-skew	8K,32,2-skew	11.0	79.7	579.7	10.0	3.4	53.4	43.2

Table C.1 Cache configuration measurements, espresso

Instruction cache configuration [total size, line size, associativity]	Data cache configuration [total size, line size, associativity]	System Power [Watt]	Performance [MIPS]	Energy Eff. [$MIPS^2/W$]	T_{Cycle} [ns]	P_{ram}/P_{total} [%]	P_{proc}/P_{total} [%]	P_{cache}/P_{total} [%]
8K,32,2-skew	8K,64,2-skew	12.2	75.6	470.2	10.4	3.5	46.0	50.5
8K,32,2-skew	8K,128,2-skew	13.7	66.5	322.0	11.8	3.5	36.0	60.5
8K,32,2-skew	16K,8,2-skew	9.4	73.4	574.2	11.0	3.0	56.6	40.3
8K,32,2-skew	16K,16,2-skew	10.1	77.5	593.4	10.4	2.9	55.2	41.9
8K,32,2-skew	16K,32,2-skew	10.6	75.5	540.3	10.7	2.7	51.5	45.8
8K,32,2-skew	16K,64,2-skew	11.6	73.1	460.1	11.1	2.3	45.1	52.6
8K,32,2-skew	16K,128,2-skew	13.2	66.8	337.6	12.2	2.1	36.3	61.6
8K,32,2-skew	32K,8,2-skew	9.6	67.3	473.2	12.1	2.6	50.5	46.9
8K,32,2-skew	32K,16,2-skew	10.1	70.6	496.2	11.5	2.5	50.3	47.1
8K,32,2-skew	32K,32,2-skew	10.7	71.5	477.2	11.4	2.4	47.8	49.8
8K,32,2-skew	32K,64,2-skew	11.4	67.6	402.3	12.1	2.2	42.5	55.2
8K,32,2-skew	32K,128,2-skew	12.7	62.0	302.5	13.2	1.8	34.8	63.4
8K,64,DM	4K,8,DM	13.0	76.3	447.0	7.7	8.3	58.4	33.2
8K,64,DM	4K,16,DM	15.0	83.8	468.8	7.0	9.3	55.5	35.2
8K,64,DM	4K,32,DM	16.6	80.7	393.6	7.0	11.3	50.4	38.3
8K,64,DM	4K,64,DM	19.6	70.1	251.3	7.0	15.0	42.7	42.4
8K,64,DM	4K,128,DM	25.2	51.9	106.8	7.0	19.2	33.1	47.7
8K,64,DM	8K,8,DM	13.2	83.6	531.4	8.3	6.8	53.1	40.1
8K,64,DM	8K,16,DM	14.4	87.7	532.5	7.8	7.4	51.7	40.9
8K,64,DM	8K,32,DM	16.8	90.5	488.7	7.2	8.5	48.3	43.2
8K,64,DM	8K,64,DM	19.9	83.2	347.3	7.0	11.4	41.8	46.8
8K,64,DM	8K,128,DM	25.5	67.8	180.6	7.0	14.8	32.7	52.5
8K,64,DM	16K,8,DM	13.3	79.8	480.0	9.3	5.9	47.3	46.8
8K,64,DM	16K,16,DM	14.5	86.7	519.2	8.6	6.1	47.0	46.9
8K,64,DM	16K,32,DM	16.3	92.1	521.3	8.0	6.2	44.8	49.0
8K,64,DM	16K,64,DM	19.0	95.0	474.3	7.6	6.3	40.3	53.4
8K,64,DM	16K,128,DM	23.8	89.6	338.2	7.5	7.3	32.5	60.1
8K,64,DM	32K,8,DM	14.2	74.1	385.3	10.4	4.9	39.4	55.7
8K,64,DM	32K,16,DM	15.2	81.4	434.2	9.4	5.0	40.5	54.5
8K,64,DM	32K,32,DM	16.5	86.0	448.3	8.9	5.0	39.7	55.2
8K,64,DM	32K,64,DM	18.5	89.2	430.9	8.6	4.5	36.8	58.7
8K,64,DM	32K,128,DM	21.3	88.2	364.4	8.7	4.0	31.5	64.6
8K,64,2-skew	4K,8,2-skew	9.4	74.0	579.5	10.4	4.7	59.2	36.2
8K,64,2-skew	4K,16,2-skew	10.0	76.2	582.7	10.4	4.6	56.0	39.4
8K,64,2-skew	4K,32,2-skew	10.9	75.6	522.7	10.4	5.0	51.1	43.9
8K,64,2-skew	4K,64,2-skew	12.8	74.1	429.1	10.4	5.6	43.7	50.7
8K,64,2-skew	4K,128,2-skew	15.4	58.2	219.7	11.6	9.0	32.7	58.3
8K,64,2-skew	8K,8,2-skew	9.8	76.5	597.6	10.4	4.1	57.0	38.8

Table C.1 Cache configuration measurements, espresso

Instruction cache configuration [total size, line size, associativity]	Data cache configuration [total size, line size, associativity]	System Power [Watt]	Performance [MIPS]	Energy Eff. [$MIPS^2/W$]	T_{Cycle} [ns]	P_{ram}/P_{total} [%]	P_{proc}/P_{total} [%]	P_{cache}/P_{total} [%]
8K,64,2-skew	8K,16,2-skew	10.2	77.6	588.4	10.4	4.1	54.6	41.2
8K,64,2-skew	8K,32,2-skew	11.1	78.3	553.5	10.4	3.9	50.4	45.7
8K,64,2-skew	8K,64,2-skew	12.8	77.4	468.1	10.4	4.0	43.6	52.3
8K,64,2-skew	8K,128,2-skew	14.4	67.9	321.0	11.8	3.9	34.4	61.7
8K,64,2-skew	16K,8,2-skew	10.0	75.2	567.5	11.0	3.7	53.4	43.0
8K,64,2-skew	16K,16,2-skew	10.7	79.4	587.3	10.4	3.5	52.1	44.4
8K,64,2-skew	16K,32,2-skew	11.2	77.3	535.8	10.7	3.3	48.6	48.0
8K,64,2-skew	16K,64,2-skew	12.2	74.8	457.5	11.1	2.8	42.8	54.3
8K,64,2-skew	16K,128,2-skew	13.8	68.3	337.4	12.2	2.5	34.6	62.8
8K,64,2-skew	32K,8,2-skew	10.1	68.9	468.7	12.1	3.2	47.8	49.0
8K,64,2-skew	32K,16,2-skew	10.6	72.3	491.6	11.5	3.1	47.6	49.3
8K,64,2-skew	32K,32,2-skew	11.3	73.1	473.6	11.4	2.9	45.3	51.8
8K,64,2-skew	32K,64,2-skew	12.0	69.2	400.5	12.1	2.8	40.4	56.8
8K,64,2-skew	32K,128,2-skew	13.3	63.3	302.1	13.2	2.2	33.2	64.5
8K,128,DM	4K,8,DM	13.7	75.9	419.7	7.7	10.1	55.5	34.4
8K,128,DM	4K,16,DM	15.7	83.4	442.1	7.0	10.9	52.8	36.3
8K,128,DM	4K,32,DM	17.5	81.4	379.2	6.9	12.5	48.3	39.1
8K,128,DM	4K,64,DM	20.4	70.8	245.6	6.9	15.7	41.4	42.9
8K,128,DM	4K,128,DM	25.9	52.5	106.3	6.9	19.4	32.6	48.0
8K,128,DM	8K,8,DM	13.9	82.9	494.9	8.3	8.6	50.4	41.0
8K,128,DM	8K,16,DM	15.2	87.0	498.0	7.8	9.2	49.1	41.7
8K,128,DM	8K,32,DM	17.6	90.0	461.3	7.2	10.1	46.1	43.8
8K,128,DM	8K,64,DM	20.7	82.9	332.3	7.0	12.5	40.3	47.2
8K,128,DM	8K,128,DM	26.4	68.6	178.3	6.9	15.2	32.0	52.7
8K,128,DM	16K,8,DM	13.9	78.9	447.1	9.3	7.8	45.0	47.2
8K,128,DM	16K,16,DM	15.2	85.8	484.5	8.6	8.0	44.7	47.3
8K,128,DM	16K,32,DM	17.1	91.3	488.9	8.0	8.0	42.8	49.3
8K,128,DM	16K,64,DM	19.8	94.3	448.4	7.6	8.0	38.6	53.4
8K,128,DM	16K,128,DM	24.5	89.1	323.7	7.5	8.5	31.6	60.0
8K,128,DM	32K,8,DM	14.8	73.1	360.8	10.4	6.6	37.8	55.6
8K,128,DM	32K,16,DM	15.9	80.5	406.8	9.4	6.8	38.8	54.5
8K,128,DM	32K,32,DM	17.2	85.1	421.1	8.9	6.8	38.0	55.2
8K,128,DM	32K,64,DM	19.2	88.3	406.5	8.6	6.1	35.4	58.5
8K,128,DM	32K,128,DM	22.0	87.3	346.1	8.7	5.4	30.5	64.2
8K,128,2-skew	4K,8,2-skew	8.9	65.9	489.4	11.8	6.1	55.8	38.1
8K,128,2-skew	4K,16,2-skew	9.3	67.6	489.8	11.8	6.0	53.0	41.0
8K,128,2-skew	4K,32,2-skew	10.2	67.0	441.6	11.8	6.2	48.6	45.1
8K,128,2-skew	4K,64,2-skew	11.8	65.7	365.0	11.8	6.6	41.9	51.5

Table C.1 Cache configuration measurements, espresso

Instruction cache configuration [total size, line size, associativity]	Data cache configuration [total size, line size, associativity]	System Power [Watt]	Performance [MIPS]	Energy Eff. [MIPS^2/W]	T_{Cycle} [ns]	P_{ram}/P_{total} [%]	P_{proc}/P_{total} [%]	P_{cache}/P_{total} [%]
8K,128,2-skew	4K,128,2-skew	15.6	56.9	207.4	11.8	9.7	31.8	58.5
8K,128,2-skew	8K,8,2-skew	9.2	67.9	502.0	11.8	5.6	53.9	40.5
8K,128,2-skew	8K,16,2-skew	9.6	68.7	494.0	11.8	5.5	51.8	42.7
8K,128,2-skew	8K,32,2-skew	10.3	69.3	465.5	11.8	5.2	47.9	46.9
8K,128,2-skew	8K,64,2-skew	11.8	68.5	396.5	11.8	5.1	41.8	53.1
8K,128,2-skew	8K,128,2-skew	14.8	67.6	308.6	11.8	4.8	33.4	61.9
8K,128,2-skew	16K,8,2-skew	9.8	70.0	501.2	11.8	4.9	50.6	44.4
8K,128,2-skew	16K,16,2-skew	10.0	70.2	492.5	11.8	4.9	49.4	45.7
8K,128,2-skew	16K,32,2-skew	10.7	70.4	464.0	11.8	4.6	46.4	49.1
8K,128,2-skew	16K,64,2-skew	12.1	70.4	411.0	11.8	4.1	41.1	54.8
8K,128,2-skew	16K,128,2-skew	14.3	68.0	323.7	12.2	3.5	33.5	63.0
8K,128,2-skew	32K,8,2-skew	10.6	68.6	443.7	12.1	4.5	45.6	49.9
8K,128,2-skew	32K,16,2-skew	10.9	70.4	455.1	11.8	4.4	45.5	50.1
8K,128,2-skew	32K,32,2-skew	11.4	70.5	434.9	11.8	4.2	43.3	52.5
8K,128,2-skew	32K,64,2-skew	12.4	68.9	381.9	12.1	3.9	38.9	57.2
8K,128,2-skew	32K,128,2-skew	13.7	63.0	290.1	13.2	3.2	32.2	64.6
16K,16,DM	4K,8,DM	11.7	68.8	404.4	8.6	4.8	58.0	37.2
16K,16,DM	4K,16,DM	12.4	70.0	394.4	8.6	6.1	54.7	39.2
16K,16,DM	4K,32,DM	13.7	67.0	328.2	8.6	8.6	49.7	41.7
16K,16,DM	4K,64,DM	16.2	58.3	210.1	8.6	12.9	42.1	45.0
16K,16,DM	4K,128,DM	20.9	43.0	88.7	8.6	18.0	32.6	49.4
16K,16,DM	8K,8,DM	12.7	78.7	487.1	8.6	3.4	53.5	43.2
16K,16,DM	8K,16,DM	13.1	78.1	465.3	8.6	4.2	51.8	44.0
16K,16,DM	8K,32,DM	14.1	75.3	401.8	8.6	5.7	48.1	46.2
16K,16,DM	8K,64,DM	16.4	68.3	284.6	8.6	8.9	41.5	49.6
16K,16,DM	8K,128,DM	20.9	55.9	149.0	8.6	13.1	32.5	54.4
16K,16,DM	16K,8,DM	13.1	77.3	454.9	9.3	2.6	47.7	49.6
16K,16,DM	16K,16,DM	14.3	83.8	490.5	8.6	2.8	47.5	49.7
16K,16,DM	16K,32,DM	15.0	83.5	464.1	8.6	3.0	45.2	51.7
16K,16,DM	16K,64,DM	16.7	82.6	407.7	8.6	3.4	40.6	56.0
16K,16,DM	16K,128,DM	20.7	77.4	289.8	8.6	5.0	32.9	62.1
16K,16,DM	32K,8,DM	14.1	71.8	367.3	10.4	2.0	39.9	58.1
16K,16,DM	32K,16,DM	15.0	78.8	412.4	9.4	2.0	41.0	56.9
16K,16,DM	32K,32,DM	16.2	83.0	425.0	8.9	2.1	40.4	57.6
16K,16,DM	32K,64,DM	18.2	86.2	408.6	8.6	1.9	37.4	60.7
16K,16,DM	32K,128,DM	20.9	85.2	346.6	8.7	1.6	32.1	66.3
16K,16,2-skew	4K,8,2-skew	9.0	71.2	561.9	10.4	2.5	61.9	35.6
16K,16,2-skew	4K,16,2-skew	9.5	73.4	564.6	10.4	2.5	58.6	38.9

Table C.1 Cache configuration measurements, espresso

Instruction cache configuration [total size, line size, associativity]	Data cache configuration [total size, line size, associativity]	System Power [Watt]	Performance [MIPS]	Energy Eff. [MIPS^2/W]	T_{Cycle} [ns]	P_{ram}/P_{total} [%]	P_{proc}/P_{total} [%]	P_{cache}/P_{total} [%]
16K,16,2-skew	4K,32,2-skew	10.4	72.8	506.4	10.4	3.1	53.5	43.4
16K,16,2-skew	4K,64,2-skew	12.2	71.5	416.8	10.4	3.9	45.6	50.4
16K,16,2-skew	4K,128,2-skew	14.8	56.9	218.4	11.6	7.8	34.1	58.2
16K,16,2-skew	8K,8,2-skew	9.4	73.6	579.0	10.4	2.0	59.7	38.3
16K,16,2-skew	8K,16,2-skew	9.8	74.6	569.7	10.4	2.0	57.2	40.8
16K,16,2-skew	8K,32,2-skew	10.6	75.3	535.5	10.4	1.9	52.8	45.3
16K,16,2-skew	8K,64,2-skew	12.2	74.6	454.9	10.4	2.2	45.7	52.1
16K,16,2-skew	8K,128,2-skew	13.8	65.5	311.3	11.8	2.5	36.0	61.5
16K,16,2-skew	16K,8,2-skew	9.5	72.3	549.9	11.0	1.5	56.0	42.6
16K,16,2-skew	16K,16,2-skew	10.2	76.3	568.2	10.4	1.4	54.6	44.0
16K,16,2-skew	16K,32,2-skew	10.7	74.3	518.5	10.7	1.4	51.0	47.7
16K,16,2-skew	16K,64,2-skew	11.7	71.9	442.0	11.1	1.2	44.8	54.0
16K,16,2-skew	16K,128,2-skew	13.2	65.7	326.1	12.2	1.1	36.2	62.7
16K,16,2-skew	32K,8,2-skew	9.7	66.2	453.8	12.1	1.3	50.0	48.7
16K,16,2-skew	32K,16,2-skew	10.2	69.5	475.6	11.5	1.2	49.9	48.9
16K,16,2-skew	32K,32,2-skew	10.8	70.3	457.9	11.4	1.2	47.4	51.4
16K,16,2-skew	32K,64,2-skew	11.4	66.5	387.3	12.1	1.1	42.3	56.6
16K,16,2-skew	32K,128,2-skew	12.7	60.9	291.9	13.2	0.9	34.7	64.4
16K,32,DM	4K,8,DM	12.4	76.1	468.3	8.0	5.6	58.9	35.5
16K,32,DM	4K,16,DM	13.2	77.7	458.1	8.0	6.9	55.4	37.7
16K,32,DM	4K,32,DM	14.6	74.2	378.5	8.0	9.4	50.1	40.6
16K,32,DM	4K,64,DM	17.3	64.2	238.2	8.0	13.6	42.1	44.3
16K,32,DM	4K,128,DM	22.4	47.1	98.8	8.0	18.6	32.5	49.0
16K,32,DM	8K,8,DM	13.0	85.7	564.5	8.3	4.0	53.7	42.3
16K,32,DM	8K,16,DM	14.0	87.7	550.7	8.0	4.9	52.2	42.9
16K,32,DM	8K,32,DM	15.1	84.4	471.7	8.0	6.5	48.3	45.3
16K,32,DM	8K,64,DM	17.6	76.1	328.5	8.0	9.6	41.4	48.9
16K,32,DM	8K,128,DM	22.6	61.7	168.3	8.0	13.7	32.2	54.1
16K,32,DM	16K,8,DM	13.2	81.9	509.9	9.3	3.2	47.7	49.1
16K,32,DM	16K,16,DM	14.3	89.0	551.9	8.6	3.4	47.4	49.2
16K,32,DM	16K,32,DM	16.2	94.5	553.2	8.0	3.6	45.1	51.2
16K,32,DM	16K,64,DM	18.1	93.5	482.9	8.0	4.0	40.3	55.7
16K,32,DM	16K,128,DM	22.5	87.3	337.8	8.0	5.5	32.4	62.1
16K,32,DM	32K,8,DM	14.2	76.1	407.9	10.4	2.5	39.5	58.0
16K,32,DM	32K,16,DM	15.2	83.6	460.2	9.4	2.6	40.6	56.8
16K,32,DM	32K,32,DM	16.4	88.4	475.4	8.9	2.6	39.9	57.6
16K,32,DM	32K,64,DM	18.4	91.7	455.5	8.6	2.3	36.8	60.9
16K,32,DM	32K,128,DM	21.4	90.6	383.8	8.7	2.0	31.4	66.6

Table C.1 Cache configuration measurements, espresso

Instruction cache configuration [total size, line size, associativity]	Data cache configuration [total size, line size, associativity]	System Power [Watt]	Performance [MIPS]	Energy Eff. [MIPS^2/W]	T_{Cycle} [ns]	P_{ram}/P_{total} [%]	P_{proc}/P_{total} [%]	P_{cache}/P_{total} [%]
16K,32,2-skew	4K,8,2-skew	8.8	72.5	595.8	10.7	2.8	61.5	35.7
16K,32,2-skew	4K,16,2-skew	9.3	74.8	598.6	10.7	2.8	58.2	39.1
16K,32,2-skew	4K,32,2-skew	10.3	74.1	534.6	10.7	3.4	52.8	43.7
16K,32,2-skew	4K,64,2-skew	12.1	72.7	436.9	10.7	4.3	44.9	50.9
16K,32,2-skew	4K,128,2-skew	15.2	58.6	226.8	11.6	8.2	33.2	58.6
16K,32,2-skew	8K,8,2-skew	9.2	75.0	614.6	10.7	2.3	59.3	38.4
16K,32,2-skew	8K,16,2-skew	9.6	76.1	604.4	10.7	2.3	56.7	41.0
16K,32,2-skew	8K,32,2-skew	10.4	76.8	566.7	10.7	2.2	52.1	45.7
16K,32,2-skew	8K,64,2-skew	12.1	75.9	476.1	10.7	2.6	44.9	52.6
16K,32,2-skew	8K,128,2-skew	14.1	68.6	334.2	11.8	2.7	35.2	62.1
16K,32,2-skew	16K,8,2-skew	9.6	75.9	601.0	11.0	1.7	55.4	42.9
16K,32,2-skew	16K,16,2-skew	10.1	77.9	603.0	10.7	1.7	54.0	44.4
16K,32,2-skew	16K,32,2-skew	10.8	78.1	564.9	10.7	1.6	50.3	48.1
16K,32,2-skew	16K,64,2-skew	11.9	75.5	478.8	11.1	1.4	44.0	54.6
16K,32,2-skew	16K,128,2-skew	13.5	68.9	351.0	12.2	1.2	35.4	63.4
16K,32,2-skew	32K,8,2-skew	9.8	69.5	493.3	12.1	1.5	49.4	49.1
16K,32,2-skew	32K,16,2-skew	10.3	72.9	517.0	11.5	1.4	49.2	49.4
16K,32,2-skew	32K,32,2-skew	10.9	73.8	496.9	11.4	1.3	46.7	52.0
16K,32,2-skew	32K,64,2-skew	11.6	69.8	418.8	12.1	1.3	41.5	57.2
16K,32,2-skew	32K,128,2-skew	13.0	63.8	313.5	13.2	1.0	33.9	65.0
16K,64,DM	4K,8,DM	13.3	79.5	474.4	7.7	6.5	57.1	36.3
16K,64,DM	4K,16,DM	14.3	81.8	469.8	7.6	7.8	53.7	38.5
16K,64,DM	4K,32,DM	15.7	78.3	389.9	7.6	10.1	48.7	41.2
16K,64,DM	4K,64,DM	18.6	67.6	246.0	7.6	14.3	41.2	44.5
16K,64,DM	4K,128,DM	23.9	49.5	102.7	7.6	19.0	32.0	48.9
16K,64,DM	8K,8,DM	13.5	87.5	565.6	8.3	4.9	51.6	43.5
16K,64,DM	8K,16,DM	14.9	91.9	567.5	7.8	5.6	50.2	44.1
16K,64,DM	8K,32,DM	16.4	89.6	490.1	7.6	7.3	46.7	46.0
16K,64,DM	8K,64,DM	19.0	80.5	341.4	7.6	10.4	40.3	49.4
16K,64,DM	8K,128,DM	24.2	65.1	174.9	7.6	14.3	31.6	54.1
16K,64,DM	16K,8,DM	13.7	83.6	510.8	9.3	4.1	45.8	50.1
16K,64,DM	16K,16,DM	14.9	91.0	554.1	8.6	4.3	45.5	50.2
16K,64,DM	16K,32,DM	16.8	96.8	557.2	8.0	4.5	43.4	52.2
16K,64,DM	16K,64,DM	19.7	99.9	507.0	7.6	4.8	38.9	56.3
16K,64,DM	16K,128,DM	24.4	93.1	355.0	7.6	6.2	31.4	62.4
16K,64,DM	32K,8,DM	14.7	77.6	409.0	10.4	3.3	38.0	58.7
16K,64,DM	32K,16,DM	15.8	85.4	462.2	9.4	3.4	39.1	57.5
16K,64,DM	32K,32,DM	17.1	90.4	478.5	8.9	3.4	38.3	58.3

Table C.1 Cache configuration measurements, espresso

Instruction cache configuration [total size, line size, associativity]	Data cache configuration [total size, line size, associativity]	System Power [Watt]	Performance [MIPS]	Energy Eff. [MIPS^2/W]	T_{Cycle} [ns]	P_{ram}/P_{total} [%]	P_{proc}/P_{total} [%]	P_{cache}/P_{total} [%]
16K,64,DM	32K,64,DM	19.2	93.8	459.1	8.6	3.1	35.4	61.5
16K,64,DM	32K,128,DM	22.2	92.7	387.7	8.7	2.7	30.3	67.1
16K,64,2-skew	4K,8,2-skew	8.9	72.0	579.3	11.1	3.1	58.6	38.3
16K,64,2-skew	4K,16,2-skew	9.4	74.0	579.6	11.1	3.0	55.5	41.5
16K,64,2-skew	4K,32,2-skew	10.4	73.3	518.4	11.1	3.6	50.5	45.9
16K,64,2-skew	4K,64,2-skew	12.2	71.8	423.8	11.1	4.3	43.1	52.6
16K,64,2-skew	4K,128,2-skew	15.6	59.6	227.3	11.6	8.3	32.2	59.4
16K,64,2-skew	8K,8,2-skew	9.3	74.3	595.0	11.1	2.6	56.5	41.0
16K,64,2-skew	8K,16,2-skew	9.7	75.2	584.2	11.1	2.6	54.1	43.4
16K,64,2-skew	8K,32,2-skew	10.5	75.9	547.8	11.1	2.4	49.8	47.8
16K,64,2-skew	8K,64,2-skew	12.2	75.0	461.8	11.1	2.8	43.0	54.2
16K,64,2-skew	8K,128,2-skew	14.6	69.9	334.8	11.8	3.0	33.9	63.1
16K,64,2-skew	16K,8,2-skew	9.9	76.6	592.6	11.1	2.0	52.9	45.1
16K,64,2-skew	16K,16,2-skew	10.2	76.9	581.6	11.1	2.0	51.5	46.5
16K,64,2-skew	16K,32,2-skew	10.9	77.1	545.5	11.1	1.9	48.1	50.1
16K,64,2-skew	16K,64,2-skew	12.4	77.1	479.1	11.1	1.7	42.2	56.1
16K,64,2-skew	16K,128,2-skew	14.1	70.4	352.6	12.2	1.5	34.0	64.4
16K,64,2-skew	32K,8,2-skew	10.2	71.0	492.2	12.1	1.9	47.2	50.9
16K,64,2-skew	32K,16,2-skew	10.8	74.5	516.0	11.5	1.8	47.0	51.2
16K,64,2-skew	32K,32,2-skew	11.4	75.4	496.6	11.4	1.7	44.7	53.6
16K,64,2-skew	32K,64,2-skew	12.1	71.3	419.5	12.1	1.6	39.8	58.6
16K,64,2-skew	32K,128,2-skew	13.5	65.2	315.0	13.2	1.3	32.7	66.0
16K,128,DM	4K,8,DM	13.8	79.5	457.0	7.7	7.8	55.0	37.2
16K,128,DM	4K,16,DM	14.9	82.7	458.2	7.5	8.8	51.9	39.3
16K,128,DM	4K,32,DM	16.4	79.1	382.3	7.5	11.0	47.2	41.8
16K,128,DM	4K,64,DM	19.2	68.3	243.6	7.5	14.7	40.3	45.0
16K,128,DM	4K,128,DM	24.4	50.1	102.9	7.5	19.1	31.7	49.2
16K,128,DM	8K,8,DM	14.1	87.4	542.2	8.3	6.2	49.7	44.1
16K,128,DM	8K,16,DM	15.4	91.8	545.5	7.8	6.9	48.4	44.7
16K,128,DM	8K,32,DM	17.1	90.6	478.9	7.5	8.4	45.1	46.5
16K,128,DM	8K,64,DM	19.7	81.4	336.4	7.5	11.0	39.3	49.7
16K,128,DM	8K,128,DM	24.9	65.9	174.5	7.5	14.6	31.1	54.3
16K,128,DM	16K,8,DM	14.2	83.4	489.9	9.3	5.4	44.2	50.4
16K,128,DM	16K,16,DM	15.5	90.8	532.0	8.6	5.6	43.9	50.6
16K,128,DM	16K,32,DM	17.4	96.7	536.7	8.0	5.7	41.9	52.4
16K,128,DM	16K,64,DM	20.3	99.9	490.9	7.6	5.9	37.7	56.4
16K,128,DM	16K,128,DM	25.2	94.0	350.2	7.5	7.0	30.7	62.4
16K,128,DM	32K,8,DM	15.2	77.3	393.7	10.4	4.5	36.9	58.6

Table C.1 Cache configuration measurements, espresso

Instruction cache configuration [total size, line size, associativity]	Data cache configuration [total size, line size, associativity]	System Power [Watt]	Performance [MIPS]	Energy Eff. [$MIPS^2/W$]	T_{Cycle} [ns]	P_{ram}/P_{total} [%]	P_{proc}/P_{total} [%]	P_{cache}/P_{total} [%]
16K,128,DM	32K,16,DM	16.3	85.2	445.0	9.4	4.6	37.9	57.6
16K,128,DM	32K,32,DM	17.6	90.2	461.4	8.9	4.6	37.1	58.3
16K,128,DM	32K,64,DM	19.7	93.6	444.0	8.6	4.2	34.4	61.4
16K,128,DM	32K,128,DM	22.7	92.5	376.6	8.7	3.6	29.5	66.8
16K,128,2-skew	4K,8,2-skew	8.5	66.7	526.3	12.2	3.4	56.6	40.0
16K,128,2-skew	4K,16,2-skew	8.9	68.6	526.6	12.2	3.4	53.5	43.1
16K,128,2-skew	4K,32,2-skew	9.8	68.0	471.9	12.2	3.9	48.9	47.2
16K,128,2-skew	4K,64,2-skew	11.5	66.5	386.3	12.2	4.7	41.8	53.5
16K,128,2-skew	4K,128,2-skew	15.3	57.2	214.3	12.2	8.8	31.3	59.9
16K,128,2-skew	8K,8,2-skew	8.8	68.9	540.6	12.2	2.9	54.5	42.6
16K,128,2-skew	8K,16,2-skew	9.2	69.8	531.3	12.2	2.9	52.2	44.9
16K,128,2-skew	8K,32,2-skew	9.9	70.4	498.9	12.2	2.7	48.2	49.1
16K,128,2-skew	8K,64,2-skew	11.5	69.5	421.4	12.2	3.0	41.7	55.2
16K,128,2-skew	8K,128,2-skew	14.5	68.6	324.5	12.2	3.1	33.0	63.9
16K,128,2-skew	16K,8,2-skew	9.4	71.1	539.6	12.2	2.3	51.1	46.7
16K,128,2-skew	16K,16,2-skew	9.6	71.4	529.7	12.2	2.2	49.8	48.0
16K,128,2-skew	16K,32,2-skew	10.3	71.5	497.4	12.2	2.1	46.5	51.4
16K,128,2-skew	16K,64,2-skew	11.7	71.5	437.8	12.2	1.9	40.9	57.2
16K,128,2-skew	16K,128,2-skew	14.4	71.5	353.7	12.2	1.6	33.1	65.2
16K,128,2-skew	32K,8,2-skew	10.5	71.4	486.9	12.2	2.0	45.8	52.2
16K,128,2-skew	32K,16,2-skew	10.5	71.6	487.2	12.2	2.0	45.5	52.5
16K,128,2-skew	32K,32,2-skew	11.1	71.7	464.6	12.2	1.9	43.3	54.8
16K,128,2-skew	32K,64,2-skew	12.4	71.7	415.9	12.2	1.7	38.7	59.6
16K,128,2-skew	32K,128,2-skew	13.8	66.1	315.9	13.2	1.4	31.9	66.8
32K,16,DM	4K,8,DM	12.2	68.0	380.1	9.4	3.3	50.8	45.9
32K,16,DM	4K,16,DM	12.8	68.9	370.1	9.4	4.6	48.1	47.3
32K,16,DM	4K,32,DM	14.0	65.7	308.5	9.4	7.2	44.1	48.7
32K,16,DM	4K,64,DM	16.2	56.7	198.0	9.4	11.8	38.0	50.2
32K,16,DM	4K,128,DM	20.4	41.2	83.5	9.4	17.4	30.3	52.3
32K,16,DM	8K,8,DM	13.3	77.7	454.5	9.4	1.8	46.5	51.7
32K,16,DM	8K,16,DM	13.7	77.0	434.4	9.4	2.6	45.2	52.2
32K,16,DM	8K,32,DM	14.6	74.1	376.4	9.4	4.3	42.3	53.4
32K,16,DM	8K,64,DM	16.6	66.8	267.9	9.4	7.7	37.1	55.2
32K,16,DM	8K,128,DM	20.8	54.0	140.5	9.4	12.3	29.7	58.0
32K,16,DM	16K,8,DM	14.8	81.9	453.7	9.4	1.2	41.7	57.1
32K,16,DM	16K,16,DM	14.9	82.7	457.9	9.4	1.3	41.3	57.3
32K,16,DM	16K,32,DM	15.6	82.4	434.1	9.4	1.6	39.5	58.9
32K,16,DM	16K,64,DM	17.3	81.4	383.2	9.4	2.1	35.7	62.2

Table C.1 Cache configuration measurements, espresso

Instruction cache configuration [total size, line size, associativity]	Data cache configuration [total size, line size, associativity]	System Power [Watt]	Performance [MIPS]	Energy Eff. [MIPS^2/W]	T_{Cycle} [ns]	P_{ram}/P_{total} [%]	P_{proc}/P_{total} [%]	P_{cache}/P_{total} [%]
32K,16,DM	16K,128,DM	21.1	76.0	274.2	9.4	4.0	29.3	66.6
32K,16,DM	32K,8,DM	15.9	77.1	373.4	10.4	0.7	35.2	64.1
32K,16,DM	32K,16,DM	17.1	85.0	421.8	9.4	0.8	36.0	63.2
32K,16,DM	32K,32,DM	17.5	85.1	414.8	9.4	0.8	35.3	63.9
32K,16,DM	32K,64,DM	18.8	85.2	385.7	9.4	0.7	32.8	66.5
32K,16,DM	32K,128,DM	21.8	85.1	333.1	9.4	0.7	28.4	71.0
32K,16,2-skew	4K,8,2-skew	8.9	66.0	489.5	11.5	2.0	56.9	41.1
32K,16,2-skew	4K,16,2-skew	9.4	67.8	490.6	11.5	1.9	54.1	44.0
32K,16,2-skew	4K,32,2-skew	10.2	67.1	442.0	11.5	2.6	49.6	47.8
32K,16,2-skew	4K,64,2-skew	11.8	65.9	366.8	11.5	3.5	42.7	53.8
32K,16,2-skew	4K,128,2-skew	15.5	57.5	213.8	11.6	7.3	32.6	60.1
32K,16,2-skew	8K,8,2-skew	9.2	68.0	502.4	11.5	1.5	55.0	43.5
32K,16,2-skew	8K,16,2-skew	9.6	68.8	494.1	11.5	1.5	52.8	45.7
32K,16,2-skew	8K,32,2-skew	10.3	69.4	465.6	11.5	1.5	48.9	49.6
32K,16,2-skew	8K,64,2-skew	11.8	68.7	398.7	11.5	1.9	42.7	55.5
32K,16,2-skew	8K,128,2-skew	14.5	66.3	302.9	11.8	2.2	34.1	63.7
32K,16,2-skew	16K,8,2-skew	9.8	70.0	501.0	11.5	1.0	51.7	47.3
32K,16,2-skew	16K,16,2-skew	10.0	70.3	492.4	11.5	1.0	50.5	48.5
32K,16,2-skew	16K,32,2-skew	10.7	70.4	464.0	11.5	0.9	47.3	51.7
32K,16,2-skew	16K,64,2-skew	12.1	70.4	411.0	11.5	0.9	41.9	57.2
32K,16,2-skew	16K,128,2-skew	14.0	66.6	316.9	12.2	0.8	34.2	64.9
32K,16,2-skew	32K,8,2-skew	10.4	67.1	434.2	12.1	0.9	46.6	52.5
32K,16,2-skew	32K,16,2-skew	10.9	70.4	455.0	11.5	0.9	46.4	52.7
32K,16,2-skew	32K,32,2-skew	11.4	70.5	434.9	11.5	0.8	44.2	55.0
32K,16,2-skew	32K,64,2-skew	12.2	67.4	373.8	12.1	0.8	39.7	59.5
32K,16,2-skew	32K,128,2-skew	13.4	61.7	283.6	13.2	0.6	32.9	66.4
32K,32,DM	4K,8,DM	12.0	73.7	452.8	8.9	3.8	54.6	41.6
32K,32,DM	4K,16,DM	12.8	75.0	441.0	8.9	5.2	51.4	43.4
32K,32,DM	4K,32,DM	14.1	71.4	362.8	8.9	8.0	46.6	45.4
32K,32,DM	4K,64,DM	16.6	61.3	226.6	8.9	12.8	39.5	47.7
32K,32,DM	4K,128,DM	21.2	44.4	92.6	8.9	18.4	30.8	50.8
32K,32,DM	8K,8,DM	13.2	85.5	555.4	8.9	2.1	49.7	48.1
32K,32,DM	8K,16,DM	13.6	84.8	528.8	8.9	3.0	48.2	48.8
32K,32,DM	8K,32,DM	14.7	81.4	452.1	8.9	4.9	44.7	50.5
32K,32,DM	8K,64,DM	17.0	73.0	313.4	8.9	8.5	38.6	52.9
32K,32,DM	8K,128,DM	21.6	58.6	158.9	8.9	13.2	30.3	56.5
32K,32,DM	16K,8,DM	14.2	87.3	536.4	9.3	1.4	44.1	54.5
32K,32,DM	16K,16,DM	15.0	91.7	562.2	8.9	1.6	43.8	54.7

Table C.1 Cache configuration measurements, espresso

Instruction cache configuration [total size, line size, associativity]	Data cache configuration [total size, line size, associativity]	System Power [Watt]	Performance [MIPS]	Energy Eff. [MIPS^2/W]	T_{Cycle} [ns]	P_{ram}/P_{total} [%]	P_{proc}/P_{total} [%]	P_{cache}/P_{total} [%]
32K,32,DM	16K,32,DM	15.8	91.4	530.4	8.9	1.9	41.6	56.5
32K,32,DM	16K,64,DM	17.6	90.3	462.8	8.9	2.4	37.2	60.4
32K,32,DM	16K,128,DM	21.8	83.9	322.9	8.9	4.5	30.0	65.5
32K,32,DM	32K,8,DM	15.3	81.0	428.9	10.4	0.9	36.6	62.5
32K,32,DM	32K,16,DM	16.4	89.4	486.2	9.4	0.9	37.6	61.5
32K,32,DM	32K,32,DM	17.8	94.8	505.1	8.9	0.9	36.8	62.3
32K,32,DM	32K,64,DM	19.3	94.8	466.6	8.9	0.8	34.0	65.2
32K,32,DM	32K,128,DM	22.6	94.8	398.3	8.9	0.8	29.0	70.2
32K,32,2-skew	4K,8,2-skew	8.7	69.7	556.3	11.4	2.1	58.5	39.4
32K,32,2-skew	4K,16,2-skew	9.2	71.7	556.6	11.4	2.1	55.4	42.5
32K,32,2-skew	4K,32,2-skew	10.1	71.0	498.4	11.4	2.8	50.5	46.7
32K,32,2-skew	4K,64,2-skew	11.8	69.6	409.1	11.4	3.6	43.1	53.2
32K,32,2-skew	4K,128,2-skew	15.6	59.3	225.4	11.6	7.8	32.3	59.9
32K,32,2-skew	8K,8,2-skew	9.1	71.9	571.3	11.4	1.6	56.4	42.0
32K,32,2-skew	8K,16,2-skew	9.5	72.8	561.0	11.4	1.6	54.0	44.3
32K,32,2-skew	8K,32,2-skew	10.3	73.5	526.3	11.4	1.6	49.8	48.6
32K,32,2-skew	8K,64,2-skew	11.9	72.6	444.2	11.4	2.0	43.1	54.9
32K,32,2-skew	8K,128,2-skew	14.6	69.5	331.5	11.8	2.3	34.0	63.7
32K,32,2-skew	16K,8,2-skew	9.7	74.2	569.2	11.4	1.1	52.9	46.1
32K,32,2-skew	16K,16,2-skew	9.9	74.5	558.6	11.4	1.1	51.5	47.4
32K,32,2-skew	16K,32,2-skew	10.6	74.6	524.2	11.4	1.0	48.1	50.9
32K,32,2-skew	16K,64,2-skew	12.1	74.6	460.8	11.4	0.9	42.3	56.8
32K,32,2-skew	16K,128,2-skew	14.0	69.8	348.1	12.2	0.9	34.2	65.0
32K,32,2-skew	32K,8,2-skew	10.2	70.4	485.0	12.1	0.9	47.3	51.8
32K,32,2-skew	32K,16,2-skew	10.8	73.9	508.2	11.5	0.9	47.1	52.0
32K,32,2-skew	32K,32,2-skew	11.4	74.8	489.3	11.4	0.9	44.7	54.4
32K,32,2-skew	32K,64,2-skew	12.1	70.8	413.7	12.1	0.8	39.9	59.2
32K,32,2-skew	32K,128,2-skew	13.4	64.7	310.8	13.2	0.7	32.8	66.5
32K,64,DM	4K,8,DM	12.4	77.9	488.6	8.6	4.0	54.7	41.3
32K,64,DM	4K,16,DM	13.2	79.3	476.1	8.6	5.4	51.4	43.3
32K,64,DM	4K,32,DM	14.6	75.5	391.2	8.6	8.0	46.6	45.4
32K,64,DM	4K,64,DM	17.2	64.7	243.2	8.6	12.7	39.5	47.9
32K,64,DM	4K,128,DM	22.0	46.7	99.1	8.6	17.9	30.9	51.2
32K,64,DM	8K,8,DM	13.7	90.7	601.3	8.6	2.4	49.7	47.9
32K,64,DM	8K,16,DM	14.1	89.9	572.3	8.6	3.2	48.1	48.7
32K,64,DM	8K,32,DM	15.2	86.3	488.5	8.6	5.0	44.6	50.4
32K,64,DM	8K,64,DM	17.6	77.1	337.0	8.6	8.5	38.5	53.0
32K,64,DM	8K,128,DM	22.4	61.7	169.5	8.6	13.0	30.3	56.8

Table C.1 Cache configuration measurements, espresso

Instruction cache configuration [total size, line size, associativity]	Data cache configuration [total size, line size, associativity]	System Power [Watt]	Performance [MIPS]	Energy Eff. [MIPS^2/W]	T_{Cycle} [ns]	P_{ram}/P_{total} [%]	P_{proc}/P_{total} [%]	P_{cache}/P_{total} [%]
32K,64,DM	16K,8,DM	14.3	89.3	559.3	9.3	1.6	43.9	54.4
32K,64,DM	16K,16,DM	15.6	97.4	608.6	8.6	1.8	43.6	54.7
32K,64,DM	16K,32,DM	16.4	97.1	573.9	8.6	2.1	41.4	56.5
32K,64,DM	16K,64,DM	18.4	95.8	499.4	8.6	2.6	36.9	60.5
32K,64,DM	16K,128,DM	22.8	88.9	346.6	8.6	4.5	29.8	65.7
32K,64,DM	32K,8,DM	15.4	82.9	446.0	10.4	1.1	36.4	62.6
32K,64,DM	32K,16,DM	16.5	91.5	506.1	9.4	1.1	37.3	61.6
32K,64,DM	32K,32,DM	17.9	97.1	525.9	8.9	1.1	36.5	62.3
32K,64,DM	32K,64,DM	20.2	100.7	503.3	8.6	1.0	33.7	65.3
32K,64,DM	32K,128,DM	23.4	99.6	423.8	8.7	0.9	28.7	70.4
32K,64,2-skew	4K,8,2-skew	8.5	67.5	535.6	12.1	2.4	56.9	40.7
32K,64,2-skew	4K,16,2-skew	9.0	69.3	535.6	12.1	2.4	53.8	43.8
32K,64,2-skew	4K,32,2-skew	9.8	68.7	479.3	12.1	3.0	49.1	47.8
32K,64,2-skew	4K,64,2-skew	11.5	67.2	392.2	12.1	4.0	41.9	54.1
32K,64,2-skew	4K,128,2-skew	15.3	57.9	218.8	12.1	8.2	31.5	60.3
32K,64,2-skew	8K,8,2-skew	8.8	69.6	549.9	12.1	1.9	54.8	43.3
32K,64,2-skew	8K,16,2-skew	9.2	70.5	540.2	12.1	1.9	52.5	45.6
32K,64,2-skew	8K,32,2-skew	10.0	71.2	507.1	12.1	1.8	48.4	49.8
32K,64,2-skew	8K,64,2-skew	11.5	70.3	428.0	12.1	2.3	41.9	55.9
32K,64,2-skew	8K,128,2-skew	14.6	69.3	329.2	12.1	2.5	33.1	64.4
32K,64,2-skew	16K,8,2-skew	9.4	71.9	548.5	12.1	1.3	51.4	47.4
32K,64,2-skew	16K,16,2-skew	9.7	72.1	538.4	12.1	1.3	50.0	48.7
32K,64,2-skew	16K,32,2-skew	10.3	72.3	505.4	12.1	1.2	46.8	52.0
32K,64,2-skew	16K,64,2-skew	11.8	72.3	444.6	12.1	1.1	41.1	57.8
32K,64,2-skew	16K,128,2-skew	14.4	71.5	355.4	12.2	1.0	33.3	65.8
32K,64,2-skew	32K,8,2-skew	10.5	72.1	494.6	12.1	1.1	46.0	52.9
32K,64,2-skew	32K,16,2-skew	10.6	72.3	495.1	12.1	1.1	45.8	53.1
32K,64,2-skew	32K,32,2-skew	11.1	72.4	471.9	12.1	1.1	43.5	55.4
32K,64,2-skew	32K,64,2-skew	12.4	72.5	422.2	12.1	1.0	38.9	60.2
32K,64,2-skew	32K,128,2-skew	13.8	66.2	317.2	13.2	0.8	31.9	67.3
32K,128,DM	4K,8,DM	12.3	77.7	491.6	8.7	4.3	54.7	40.9
32K,128,DM	4K,16,DM	13.1	79.1	478.7	8.7	5.7	51.4	42.9
32K,128,DM	4K,32,DM	14.4	75.3	392.8	8.7	8.4	46.5	45.1
32K,128,DM	4K,64,DM	17.1	64.5	243.5	8.7	13.0	39.4	47.6
32K,128,DM	4K,128,DM	21.8	46.5	99.0	8.7	18.2	30.8	51.0
32K,128,DM	8K,8,DM	13.5	90.5	606.1	8.7	2.8	49.7	47.6
32K,128,DM	8K,16,DM	14.0	89.7	576.5	8.7	3.6	48.1	48.3
32K,128,DM	8K,32,DM	15.1	86.1	491.5	8.7	5.3	44.5	50.1

Table C.1 Cache configuration measurements, espresso

Instruction cache configuration [total size, line size, associativity]	Data cache configuration [total size, line size, associativity]	System Power [Watt]	Performance [MIPS]	Energy Eff. [MIPS^2/W]	T_{Cycle} [ns]	P_{ram}/P_{total} [%]	P_{proc}/P_{total} [%]	P_{cache}/P_{total} [%]
32K,128,DM	8K,64,DM	17.5	76.9	338.1	8.7	8.8	38.4	52.8
32K,128,DM	8K,128,DM	22.3	61.5	169.7	8.7	13.3	30.1	56.6
32K,128,DM	16K,8,DM	14.3	90.2	570.2	9.3	1.9	43.9	54.2
32K,128,DM	16K,16,DM	15.4	97.3	613.5	8.7	2.1	43.5	54.4
32K,128,DM	16K,32,DM	16.3	97.0	578.1	8.7	2.4	41.3	56.3
32K,128,DM	16K,64,DM	18.2	95.7	502.5	8.7	2.9	36.8	60.3
32K,128,DM	16K,128,DM	22.6	88.7	347.8	8.7	4.8	29.6	65.6
32K,128,DM	32K,8,DM	15.4	83.8	454.1	10.4	1.4	36.3	62.4
32K,128,DM	32K,16,DM	16.6	92.5	515.5	9.4	1.4	37.2	61.4
32K,128,DM	32K,32,DM	18.0	98.1	535.7	8.9	1.4	36.4	62.2
32K,128,DM	32K,64,DM	20.0	100.7	506.6	8.7	1.3	33.6	65.1
32K,128,DM	32K,128,DM	23.5	100.6	431.0	8.7	1.1	28.6	70.3
32K,128,2-skew	4K,8,2-skew	7.9	62.8	497.5	13.2	2.5	55.7	41.8
32K,128,2-skew	4K,16,2-skew	8.4	64.3	495.2	13.2	2.5	52.8	44.8
32K,128,2-skew	4K,32,2-skew	9.2	63.7	443.5	13.2	3.1	48.2	48.7
32K,128,2-skew	4K,64,2-skew	10.7	62.3	362.8	13.2	4.0	41.2	54.8
32K,128,2-skew	4K,128,2-skew	14.2	53.6	201.7	13.2	8.1	31.0	60.9
32K,128,2-skew	8K,8,2-skew	8.2	64.6	508.7	13.2	1.9	53.7	44.3
32K,128,2-skew	8K,16,2-skew	8.6	65.4	498.6	13.2	2.0	51.5	46.5
32K,128,2-skew	8K,32,2-skew	9.3	65.9	467.4	13.2	1.9	47.5	50.6
32K,128,2-skew	8K,64,2-skew	10.7	65.1	394.8	13.2	2.3	41.1	56.6
32K,128,2-skew	8K,128,2-skew	13.6	64.2	303.8	13.2	2.5	32.5	64.9
32K,128,2-skew	16K,8,2-skew	8.8	66.5	505.4	13.2	1.4	50.4	48.3
32K,128,2-skew	16K,16,2-skew	9.0	66.8	495.7	13.2	1.4	49.1	49.5
32K,128,2-skew	16K,32,2-skew	9.6	66.9	465.3	13.2	1.3	45.9	52.8
32K,128,2-skew	16K,64,2-skew	10.9	66.9	409.4	13.2	1.2	40.4	58.4
32K,128,2-skew	16K,128,2-skew	13.5	66.8	330.8	13.2	1.0	32.7	66.3
32K,128,2-skew	32K,8,2-skew	9.8	66.8	455.8	13.2	1.2	45.1	53.7
32K,128,2-skew	32K,16,2-skew	9.8	66.9	455.9	13.2	1.2	44.9	53.9
32K,128,2-skew	32K,32,2-skew	10.3	67.0	434.5	13.2	1.2	42.7	56.1
32K,128,2-skew	32K,64,2-skew	11.6	67.0	388.9	13.2	1.0	38.2	60.8
32K,128,2-skew	32K,128,2-skew	14.0	67.0	320.1	13.2	0.9	31.4	67.7
64K,16,DM	4K,8,DM	13.2	60.9	280.0	10.8	2.7	40.7	56.6
64K,16,DM	4K,16,DM	13.9	61.5	273.2	10.8	3.8	38.8	57.4
64K,16,DM	4K,32,DM	14.8	58.6	232.2	10.8	6.2	36.4	57.4
64K,16,DM	4K,64,DM	16.5	50.5	154.8	10.8	10.7	32.7	56.6
64K,16,DM	4K,128,DM	19.6	36.6	68.3	10.8	16.6	27.5	55.9
64K,16,DM	8K,8,DM	14.5	69.0	328.0	10.8	1.4	37.1	61.5

Table C.1 Cache configuration measurements, espresso

Instruction cache configuration [total size, line size, associativity]	Data cache configuration [total size, line size, associativity]	System Power [Watt]	Performance [MIPS]	Energy Eff. [MIPS^2/W]	T_{Cycle} [ns]	P_{ram}/P_{total} [%]	P_{proc}/P_{total} [%]	P_{cache}/P_{total} [%]
64K,16,DM	8K,16,DM	14.8	68.4	315.3	10.8	2.1	36.3	61.6
64K,16,DM	8K,32,DM	15.6	65.8	278.1	10.8	3.6	34.6	61.8
64K,16,DM	8K,64,DM	17.1	59.3	205.0	10.8	6.8	31.4	61.8
64K,16,DM	8K,128,DM	20.4	47.9	112.6	10.8	11.5	26.4	62.0
64K,16,DM	16K,8,DM	16.0	72.5	329.1	10.8	0.9	33.7	65.4
64K,16,DM	16K,16,DM	16.1	73.1	331.6	10.8	1.0	33.4	65.6
64K,16,DM	16K,32,DM	16.7	72.8	317.0	10.8	1.3	32.2	66.5
64K,16,DM	16K,64,DM	18.1	71.9	284.9	10.8	1.7	29.7	68.6
64K,16,DM	16K,128,DM	21.3	67.1	211.6	10.8	3.6	25.3	71.2
64K,16,DM	32K,8,DM	18.5	74.8	303.3	10.8	0.5	29.2	70.3
64K,16,DM	32K,16,DM	18.1	75.0	310.7	10.8	0.5	29.7	69.8
64K,16,DM	32K,32,DM	18.4	75.1	306.4	10.8	0.5	29.2	70.3
64K,16,DM	32K,64,DM	19.6	75.2	288.2	10.8	0.5	27.5	72.0
64K,16,DM	32K,128,DM	22.2	75.1	254.2	10.8	0.5	24.2	75.3
64K,16,2-skew	4K,8,2-skew	9.2	60.0	390.2	12.8	1.6	49.1	49.3
64K,16,2-skew	4K,16,2-skew	9.7	61.5	390.2	12.8	1.5	46.9	51.6
64K,16,2-skew	4K,32,2-skew	10.4	60.9	355.6	12.8	2.1	43.5	54.4
64K,16,2-skew	4K,64,2-skew	11.9	59.8	300.2	12.8	3.0	38.2	58.9
64K,16,2-skew	4K,128,2-skew	15.1	52.4	181.9	12.8	6.7	30.1	63.2
64K,16,2-skew	8K,8,2-skew	9.6	61.7	399.1	12.8	1.1	47.5	51.3
64K,16,2-skew	8K,16,2-skew	9.9	62.4	393.0	12.8	1.2	45.8	53.0
64K,16,2-skew	8K,32,2-skew	10.6	62.8	372.7	12.8	1.1	42.9	56.0
64K,16,2-skew	8K,64,2-skew	11.9	62.2	324.2	12.8	1.5	38.0	60.5
64K,16,2-skew	8K,128,2-skew	14.6	61.4	257.5	12.8	1.9	31.0	67.1
64K,16,2-skew	16K,8,2-skew	10.1	63.4	397.9	12.8	0.7	44.9	54.4
64K,16,2-skew	16K,16,2-skew	10.3	63.6	391.8	12.8	0.7	44.0	55.3
64K,16,2-skew	16K,32,2-skew	10.9	63.7	371.8	12.8	0.7	41.6	57.8
64K,16,2-skew	16K,64,2-skew	12.2	63.7	333.6	12.8	0.7	37.3	62.1
64K,16,2-skew	16K,128,2-skew	14.6	63.7	277.2	12.8	0.6	31.1	68.3
64K,16,2-skew	32K,8,2-skew	11.1	63.6	365.2	12.8	0.6	41.0	58.4
64K,16,2-skew	32K,16,2-skew	11.1	63.8	365.4	12.8	0.6	40.8	58.6
64K,16,2-skew	32K,32,2-skew	11.6	63.8	351.0	12.8	0.6	39.1	60.3
64K,16,2-skew	32K,64,2-skew	12.8	63.9	319.2	12.8	0.5	35.5	63.9
64K,16,2-skew	32K,128,2-skew	14.7	62.1	261.7	13.2	0.5	30.0	69.6
64K,32,DM	4K,8,DM	12.7	68.5	370.8	9.8	3.2	46.9	49.9
64K,32,DM	4K,16,DM	13.4	69.5	361.2	9.8	4.5	44.4	51.2
64K,32,DM	4K,32,DM	14.5	66.1	301.4	9.8	7.1	40.9	52.0
64K,32,DM	4K,64,DM	16.6	56.7	193.5	9.8	11.8	35.7	52.5

Table C.1 Cache configuration measurements, espresso

Instruction cache configuration [total size, line size, associativity]	Data cache configuration [total size, line size, associativity]	System Power [Watt]	Performance [MIPS]	Energy Eff. [$MIPS^2/W$]	T_{Cycle} [ns]	P_{ram}/P_{total} [%]	P_{proc}/P_{total} [%]	P_{cache}/P_{total} [%]
64K,32,DM	4K,128,DM	20.5	40.9	81.6	9.8	17.6	28.9	53.4
64K,32,DM	8K,8,DM	14.0	78.9	445.6	9.8	1.7	42.6	55.8
64K,32,DM	8K,16,DM	14.3	78.2	426.1	9.8	2.5	41.5	56.1
64K,32,DM	8K,32,DM	15.2	75.0	369.4	9.8	4.2	39.0	56.9
64K,32,DM	8K,64,DM	17.2	67.3	262.9	9.8	7.7	34.5	57.8
64K,32,DM	8K,128,DM	21.1	54.0	137.7	9.8	12.4	28.1	59.5
64K,32,DM	16K,8,DM	15.6	83.4	446.9	9.8	1.0	38.2	60.8
64K,32,DM	16K,16,DM	15.7	84.2	451.2	9.8	1.2	37.8	61.0
64K,32,DM	16K,32,DM	16.4	83.9	428.3	9.8	1.5	36.2	62.3
64K,32,DM	16K,64,DM	18.1	82.8	378.9	9.8	2.0	32.8	65.1
64K,32,DM	16K,128,DM	21.8	77.0	271.7	9.8	4.0	27.2	68.8
64K,32,DM	32K,8,DM	17.4	81.6	383.8	10.4	0.6	32.3	67.1
64K,32,DM	32K,16,DM	18.0	86.7	417.7	9.8	0.6	33.0	66.3
64K,32,DM	32K,32,DM	18.3	86.8	411.0	9.8	0.6	32.4	67.0
64K,32,DM	32K,64,DM	19.7	86.9	382.9	9.8	0.6	30.2	69.2
64K,32,DM	32K,128,DM	22.7	86.8	331.9	9.8	0.6	26.1	73.3
64K,32,2-skew	4K,8,2-skew	8.7	63.5	465.5	12.7	1.8	53.1	45.1
64K,32,2-skew	4K,16,2-skew	9.1	65.0	464.6	12.7	1.7	50.5	47.8
64K,32,2-skew	4K,32,2-skew	9.9	64.4	418.9	12.7	2.4	46.4	51.2
64K,32,2-skew	4K,64,2-skew	11.5	63.1	347.9	12.7	3.2	40.1	56.7
64K,32,2-skew	4K,128,2-skew	14.9	54.6	199.5	12.7	7.2	30.8	62.0
64K,32,2-skew	8K,8,2-skew	9.0	65.3	476.0	12.7	1.3	51.3	47.4
64K,32,2-skew	8K,16,2-skew	9.3	66.0	467.5	12.7	1.3	49.3	49.4
64K,32,2-skew	8K,32,2-skew	10.1	66.5	440.3	12.7	1.3	45.7	53.0
64K,32,2-skew	8K,64,2-skew	11.5	65.7	375.9	12.7	1.7	40.0	58.3
64K,32,2-skew	8K,128,2-skew	14.3	64.9	294.0	12.7	2.0	32.1	65.9
64K,32,2-skew	16K,8,2-skew	9.5	67.2	473.5	12.7	0.8	48.2	51.0
64K,32,2-skew	16K,16,2-skew	9.8	67.4	465.2	12.7	0.8	47.1	52.1
64K,32,2-skew	16K,32,2-skew	10.4	67.5	438.6	12.7	0.8	44.2	55.0
64K,32,2-skew	16K,64,2-skew	11.7	67.5	389.1	12.7	0.7	39.2	60.1
64K,32,2-skew	16K,128,2-skew	14.3	67.4	317.9	12.7	0.7	32.1	67.2
64K,32,2-skew	32K,8,2-skew	10.6	67.4	430.1	12.7	0.7	43.5	55.8
64K,32,2-skew	32K,16,2-skew	10.6	67.5	430.3	12.7	0.7	43.4	56.0
64K,32,2-skew	32K,32,2-skew	11.1	67.6	411.4	12.7	0.7	41.4	58.0
64K,32,2-skew	32K,64,2-skew	12.3	67.7	370.6	12.7	0.6	37.2	62.2
64K,32,2-skew	32K,128,2-skew	14.3	65.0	295.7	13.2	0.5	30.9	68.6
64K,64,DM	4K,8,DM	12.5	72.2	416.9	9.5	3.4	49.1	47.6
64K,64,DM	4K,16,DM	13.2	73.3	405.5	9.5	4.7	46.3	49.0

Table C.1 Cache configuration measurements, espresso

Instruction cache configuration [total size, line size, associativity]	Data cache configuration [total size, line size, associativity]	System Power [Watt]	Performance [MIPS]	Energy Eff. [$MIPS^2/W$]	T_{Cycle} [ns]	P_{ram}/P_{total} [%]	P_{proc}/P_{total} [%]	P_{cache}/P_{total} [%]
64K,64,DM	4K,32,DM	14.4	69.7	336.2	9.5	7.3	42.5	50.2
64K,64,DM	4K,64,DM	16.7	59.6	212.8	9.5	12.0	36.7	51.3
64K,64,DM	4K,128,DM	20.9	42.9	88.4	9.5	17.5	29.4	53.0
64K,64,DM	8K,8,DM	13.8	83.3	503.6	9.5	1.9	44.5	53.6
64K,64,DM	8K,16,DM	14.2	82.6	480.5	9.5	2.7	43.2	54.1
64K,64,DM	8K,32,DM	15.2	79.2	414.1	9.5	4.4	40.5	55.2
64K,64,DM	8K,64,DM	17.3	70.9	290.9	9.5	7.9	35.5	56.6
64K,64,DM	8K,128,DM	21.4	56.6	149.4	9.5	12.5	28.6	58.9
64K,64,DM	16K,8,DM	15.4	88.2	503.6	9.5	1.2	39.7	59.1
64K,64,DM	16K,16,DM	15.6	89.1	508.6	9.5	1.3	39.3	59.3
64K,64,DM	16K,32,DM	16.4	88.8	481.5	9.5	1.6	37.5	60.8
64K,64,DM	16K,64,DM	18.1	87.6	423.0	9.5	2.2	33.8	64.0
64K,64,DM	16K,128,DM	22.1	81.3	299.2	9.5	4.1	27.8	68.1
64K,64,DM	32K,8,DM	16.8	83.6	415.4	10.4	0.7	33.3	66.0
64K,64,DM	32K,16,DM	18.0	91.8	467.9	9.5	0.8	34.1	65.2
64K,64,DM	32K,32,DM	18.4	91.9	460.0	9.5	0.8	33.4	65.9
64K,64,DM	32K,64,DM	19.8	92.0	426.9	9.5	0.7	31.0	68.3
64K,64,DM	32K,128,DM	23.0	91.9	367.4	9.5	0.6	26.7	72.7
64K,64,2-skew	4K,8,2-skew	8.3	63.0	474.6	13.1	1.9	53.3	44.8
64K,64,2-skew	4K,16,2-skew	8.8	64.5	472.9	13.1	1.9	50.6	47.5
64K,64,2-skew	4K,32,2-skew	9.6	63.9	425.2	13.1	2.6	46.4	51.0
64K,64,2-skew	4K,64,2-skew	11.1	62.5	350.5	13.1	3.5	40.0	56.5
64K,64,2-skew	4K,128,2-skew	14.6	53.9	199.0	13.1	7.6	30.5	61.9
64K,64,2-skew	8K,8,2-skew	8.7	64.8	485.2	13.1	1.4	51.5	47.1
64K,64,2-skew	8K,16,2-skew	9.0	65.5	476.2	13.1	1.5	49.4	49.1
64K,64,2-skew	8K,32,2-skew	9.7	66.0	447.9	13.1	1.4	45.7	52.9
64K,64,2-skew	8K,64,2-skew	11.2	65.2	380.8	13.1	1.9	39.9	58.3
64K,64,2-skew	8K,128,2-skew	14.0	64.3	295.6	13.1	2.2	31.8	66.0
64K,64,2-skew	16K,8,2-skew	9.2	66.7	482.6	13.1	0.9	48.3	50.8
64K,64,2-skew	16K,16,2-skew	9.4	66.9	474.0	13.1	0.9	47.1	51.9
64K,64,2-skew	16K,32,2-skew	10.1	67.0	446.2	13.1	0.9	44.2	54.9
64K,64,2-skew	16K,64,2-skew	11.4	67.0	394.7	13.1	0.8	39.1	60.1
64K,64,2-skew	16K,128,2-skew	14.0	67.0	321.2	13.1	0.7	31.9	67.4
64K,64,2-skew	32K,8,2-skew	10.2	66.9	437.3	13.1	0.8	43.5	55.7
64K,64,2-skew	32K,16,2-skew	10.3	67.1	437.5	13.1	0.8	43.3	55.9
64K,64,2-skew	32K,32,2-skew	10.8	67.2	417.9	13.1	0.8	41.3	58.0
64K,64,2-skew	32K,64,2-skew	12.0	67.2	375.6	13.1	0.7	37.0	62.3
64K,64,2-skew	32K,128,2-skew	14.4	66.6	308.2	13.2	0.6	30.7	68.7

Table C.1 Cache configuration measurements, espresso

Instruction cache configuration [total size, line size, associativity]	Data cache configuration [total size, line size, associativity]	System Power [Watt]	Performance [MIPS]	Energy Eff. [$MIPS^2/W$]	T_{Cycle} [ns]	P_{ram}/P_{total} [%]	P_{proc}/P_{total} [%]	P_{cache}/P_{total} [%]
64K,128,DM	4K,8,DM	11.5	69.9	423.5	9.9	3.9	51.0	45.2
64K,128,DM	4K,16,DM	12.2	70.9	410.7	9.9	5.3	48.0	46.7
64K,128,DM	4K,32,DM	13.4	67.4	337.9	9.9	8.1	43.7	48.2
64K,128,DM	4K,64,DM	15.7	57.6	211.0	9.9	12.9	37.3	49.8
64K,128,DM	4K,128,DM	19.9	41.4	86.3	9.9	18.5	29.5	52.0
64K,128,DM	8K,8,DM	12.7	80.7	513.7	9.9	2.2	46.3	51.5
64K,128,DM	8K,16,DM	13.1	80.0	488.9	9.9	3.1	44.9	52.0
64K,128,DM	8K,32,DM	14.1	76.7	418.8	9.9	4.9	41.8	53.3
64K,128,DM	8K,64,DM	16.2	68.6	290.6	9.9	8.6	36.3	55.1
64K,128,DM	8K,128,DM	20.3	54.7	147.2	9.9	13.3	28.9	57.8
64K,128,DM	16K,8,DM	14.3	85.5	512.7	9.9	1.4	41.2	57.3
64K,128,DM	16K,16,DM	14.4	86.4	517.7	9.9	1.6	40.8	57.6
64K,128,DM	16K,32,DM	15.1	86.0	488.8	9.9	1.9	38.8	59.2
64K,128,DM	16K,64,DM	16.9	84.9	427.0	9.9	2.5	34.8	62.7
64K,128,DM	16K,128,DM	20.8	78.7	298.5	9.9	4.5	28.3	67.2
64K,128,DM	32K,8,DM	16.3	84.7	439.1	10.4	0.9	34.3	64.8
64K,128,DM	32K,16,DM	16.7	89.0	474.0	9.9	0.9	35.2	63.9
64K,128,DM	32K,32,DM	17.1	89.2	465.6	9.9	0.9	34.4	64.6
64K,128,DM	32K,64,DM	18.5	89.2	430.7	9.9	0.9	31.8	67.3
64K,128,DM	32K,128,DM	21.6	89.2	368.6	9.9	0.8	27.3	72.0
64K,128,2-skew	4K,8,2-skew	7.6	58.7	452.5	14.3	2.0	53.5	44.5
64K,128,2-skew	4K,16,2-skew	8.0	59.9	448.5	14.3	2.0	50.8	47.2
64K,128,2-skew	4K,32,2-skew	8.7	59.3	402.6	14.3	2.6	46.6	50.8
64K,128,2-skew	4K,64,2-skew	10.2	58.0	330.8	14.3	3.5	40.0	56.5
64K,128,2-skew	4K,128,2-skew	13.4	50.0	186.1	14.3	7.5	30.4	62.2
64K,128,2-skew	8K,8,2-skew	7.9	60.2	460.5	14.3	1.5	51.7	46.8
64K,128,2-skew	8K,16,2-skew	8.2	60.8	450.6	14.3	1.6	49.6	48.8
64K,128,2-skew	8K,32,2-skew	8.9	61.2	422.6	14.3	1.5	45.9	52.6
64K,128,2-skew	8K,64,2-skew	10.2	60.5	358.5	14.3	1.9	39.9	58.2
64K,128,2-skew	8K,128,2-skew	12.8	59.6	277.2	14.3	2.2	31.8	66.0
64K,128,2-skew	16K,8,2-skew	8.4	61.8	455.7	14.3	1.0	48.5	50.4
64K,128,2-skew	16K,16,2-skew	8.6	62.0	447.0	14.3	1.0	47.4	51.6
64K,128,2-skew	16K,32,2-skew	9.2	62.1	420.1	14.3	0.9	44.4	54.7
64K,128,2-skew	16K,64,2-skew	10.4	62.1	370.9	14.3	0.9	39.2	59.9
64K,128,2-skew	16K,128,2-skew	12.8	62.0	301.0	14.3	0.8	31.9	67.3
64K,128,2-skew	32K,8,2-skew	9.3	62.0	412.0	14.3	0.9	43.6	55.5
64K,128,2-skew	32K,16,2-skew	9.4	62.1	411.9	14.3	0.9	43.5	55.7

Table C.1 Cache configuration measurements, espresso

Instruction cache configuration [total size, line size, associativity]	Data cache configuration [total size, line size, associativity]	System Power [Watt]	Performance [MIPS]	Energy Eff. [$MIPS^2/W$]	T_{Cycle} [ns]	P_{ram}/P_{total} [%]	P_{proc}/P_{total} [%]	P_{cache}/P_{total} [%]
64K,128,2-skew	32K,32,2-skew	9.8	62.2	393.0	14.3	0.8	41.4	57.8
64K,128,2-skew	32K,64,2-skew	11.0	62.2	352.6	14.3	0.8	37.1	62.2
64K,128,2-skew	32K,128,2-skew	13.3	62.2	291.4	14.3	0.6	30.6	68.7