UNIVERSITY OF NEW SOUTH WALES
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

A comparison of address translation mechanisms for virtually-addressed caches in embedded systems

Harvey Tuch

Bachelor of Engineering (Computer Engineering)

November 2002

Supervisor: Prof. Gernot Heiser

Abstract

Virtual memory is a runtime abstraction that allows for more secure, robust and easier to develop software. Virtual memory may be used on embedded systems to reduce costs of development and improve the reliability of systems. However, virtual memory has both time and energy overheads. Virtually-addressed caches offer a means of potentially reducing these.

In this thesis, one component of virtual memory, address translation, is explored on virtually-addressed caches, in the context of embedded systems. A detailed survey of the literature is presented, and an experimental approach is utilised to quantitatively compare the options available for address translation on virtually-addressed caches, with the goal of characterising the mechanism that provides the lowest energy and time overheads.

Acknowledgments

I would like to thank my family, Jeff, Sandy, Laurie and Sharni, and express my gratitude for their endless dedication and support for me throughout my education. I'd also like to thank John Lai for the provision of intelligent conversation at all times during the past 4 years, and in particular the generous amount of time spent on proof-reading and feedback for this thesis.

I'd like to thank Adam Wiggins for the many hours of supervision, feedback, and discussion he provided — I doubt I would have remained as focussed without this. The large amount of feedback provided by my supervisor Gernot Heiser was also of great help and made a large difference to the structure of this thesis. I'd like to express thanks to Chris Szmajda for suggesting this topic through his discussion paper and the critical and insightful feedback he provided.

I would finally like to thank the others who proof-read drafts of this thesis: Luke Macpherson and Shane Stephens.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

For a program to execute, it is necessary to fetch instructions from, and load from and store data to memory. The location in memory that a datum is stored in is referred to as its *physical address*. In a simple system, as indicated in Figure 1.1, the CPU will generate the physical address and this will be directly used to access memory.

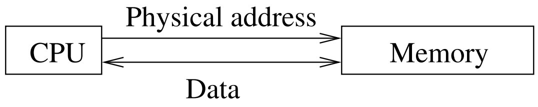

Figure 1.1: Addressing of memory directly by CPU.

This approach is limited, as a program that requires access to more data than is storable in main memory is required to explicitly transfer the data between main memory and secondary storage. If several programs execute on the CPU, there are no restrictions on the data to which each program has access, and so one program may read the data of another, or an unstable or malicious program may not only damage itself but also alter the contents of memory of other programs executing on the system.

It is for these reasons, amongst others, that *virtual memory* is used in many systems. Virtual memory is an abstraction provided to the program executing on the CPU, in which the program executes in a *virtual address space*. The CPU generates *virtual addresses*, and hardware mechanisms provide *address translation* to generate the physical address required for addressing memory, and *memory protection* to restrict access to only the locations the running program, or *process*, has the required access rights (see Figure 1.2). If no valid virtual-to-physical address mapping exists, or if the process does not have access rights to the virtual address generated by the CPU, a *page fault exception* is generated. This exception is so named as rather than mapping individual virtual addresses to physical addresses, it is typical to map a block of contiguous virtual addresses, called *pages*, to another block of contiguous physical addresses, called *frames* (see Figure 1.3).

Virtual memory presents several benefits, such as allowing programs to transpar-

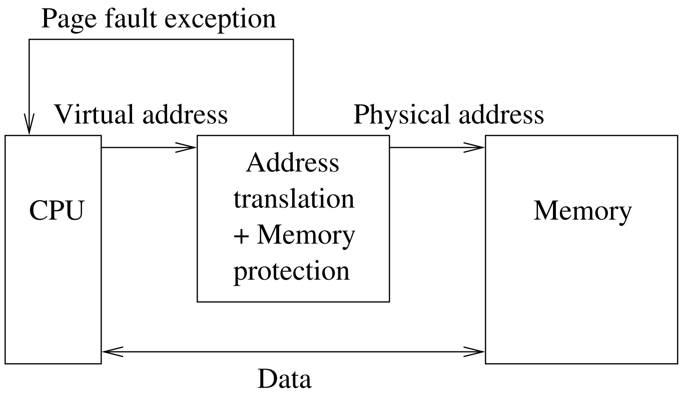

Figure 1.2: Virtual memory.

ently access data in an address space larger than that provided by main memory, protection of memory so that a program may only access data to which it has had access rights granted and so that the text segment may by protected from corruption, and the elimination of the reliance of addressing inside a program on the location of data in physical memory. This allows for more secure, robust and easier to develop software.

An embedded system consists of an integrated microprocessor, memory and I/O devices, together with application-specific software. Embedded systems constitute the majority of computer systems in use today, and are an important component of many electronic devices from mobile phones to micro-satellites. Embedded systems benefit from virtual memory by the increased ease of development, freeing the programmer from concerns about conflicts between processes and improving the reliability of the system [47]. The improved security, while an advantage, is a lower priority to the above benefits, as many embedded systems have fixed the programs that may execute on them and do not exist in the multi-user environment that traditional systems with virtual memory do. Embedded systems, in the form of *personal digital assistants* (PDAs) are, however, increasingly being used in networked environments and downloading and executing untrusted code, making the security that virtual memory can provide of increasing utility and relevance.

Virtual memory, however, is not without its costs. The provision of this abstraction presents both time and energy overheads. These can be significant, with a 10–20% performance overhead on standard workloads [19] and even 40% overhead in some extreme cases. The energy consumption of the associative caches used to provide address translation on standard architectures is high — on the StrongARM SA-110 it was found to be 17% of the total processor power budget [29]. On embedded systems, energy consumption is a priority, with many embedded systems running on batteries or other limited power sources.

Much of this overhead is typically associated with the *translation-lookaside buffer* (TLB). The TLB is an associative cache that, given a virtual address, provides the corresponding physical address. The TLB only holds a subset of the total virtual-to-physical

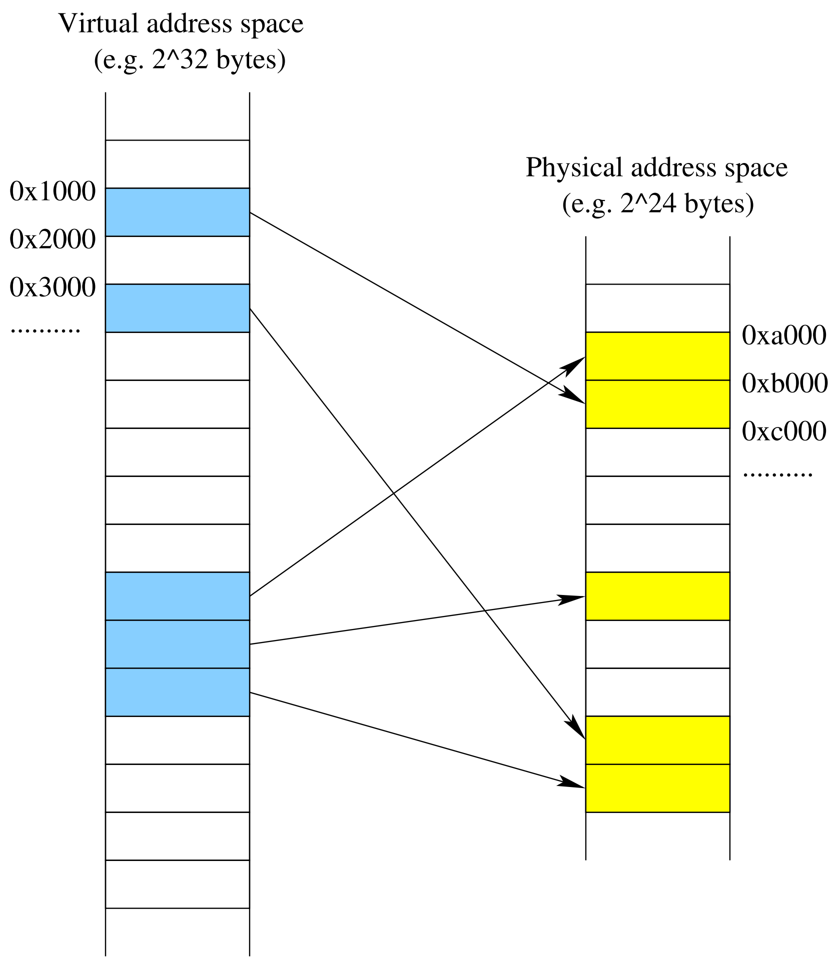

Figure 1.3: Mappings from pages in the virtual address space to frames in the physical address space.

mappings, which are held in memory in a data structure called a *page table*. The TLB caches these to reduce the large overhead that would be imposed if the page table had to be consulted on every memory access. The TLB also typically stores protection information and may generate the page fault exception if either a valid virtual-to-physical mapping does not exist, or if the process does not have access rights to the data at the requested virtual address.

The significant problems with a conventional TLB are summarised below:

- The TLB is only a cache of the virtual-to-physical mappings, and so is only able to store a small number of these at a time. If a mapping is not present, it is necessary to perform a *TLB refill* from the page table that contains these mappings. When the TLB is not able to cover the address space being currently used by a program, the *working set* of pages, *TLB thrashing* will set in, where the TLB refills necessary for the program's memory accesses will be so frequent that they noticeably degrade performance. This is the source of the TLB time overhead. Since the TLB is on the cache hit path, and larger TLB sizes imply higher access times, the TLB does not scale to larger sizes as this would increase CPU cycle time. This puts a limit on the practical address space utilisation of an application.

- The TLB is typically a fully-associative memory, consulted on every instruction reference, and load and store operation. Each of these TLB lookups requires energy, and the overhead of this can be significant, especially on embedded systems where the memory subsystem may be the greatest consumer of power.

- Rather than mapping each individual virtual address to its corresponding physical address, the TLB stores mappings at *page* granularity. This granularity is not necessarily the same granularity that is desirable for memory protection, with ideal memory protection granularity ranging from segment down to word size. In addition, the TLB's already limited coverage is further restricted when processes share data between virtual address spaces, with a separate entry for the shared page for each of the address spaces required in many TLBs.

It is possible to avoid some of these problems associated with the TLB by the use of *virtually-addressed caches*. Since memory access time is usually one or more orders of magnitude slower than the processor's clock cycle time, it is usual to have one or more levels of higher speed but more costly, and consequently lower capacity, *caches* between memory and the processor (see Figure 1.4). Most of the memory accesses are handled by the cache, and only in the, hopefully infrequent, case where the requested data are not in the cache is an access to memory necessary. While most systems address the cache with a physical address, it is also possible to place the cache before the address translation takes place and address the cache with the virtual address from the CPU. Such caches are known as virtually-addressed caches. Virtually-addressed caches do not require address translation to be performed prior to the cache access, and allow the TLB to either be eliminated or only used on the infrequent cache miss.

In this thesis, I explore the options available for address translation when a virtually-addressed cache is used. Related work in the literature has presented each of these

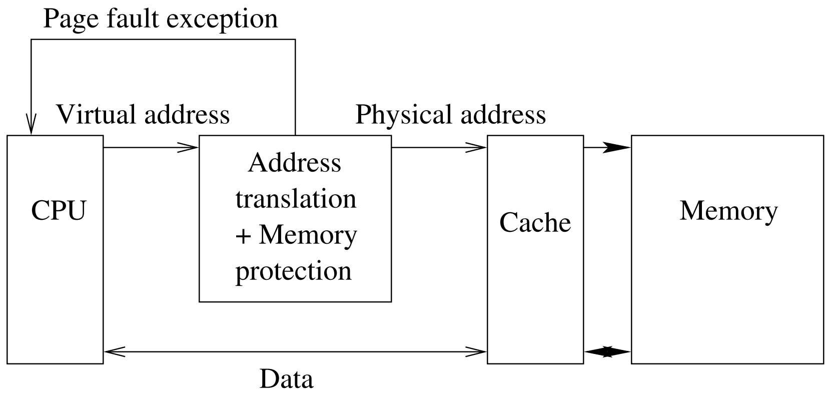

Figure 1.4: Caching.

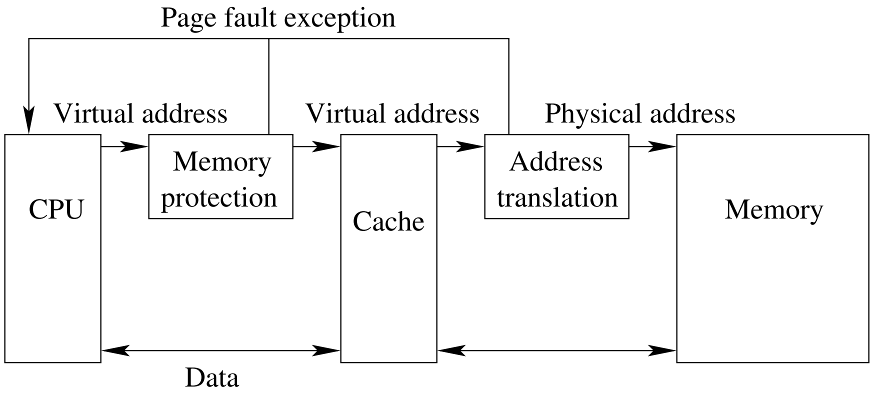

Figure 1.5: A virtually-addressed cache.

mechanisms alone, or in comparison to a standard TLB, or compared them in the context of high-end multiprocessor systems. The work done in this thesis differs, in that it:

- quantitatively examines both time and energy overheads of address translation under each of these mechanisms. In past work in the literature only a qualitative assessment of the energy savings of virtually-addressed caches has been presented;

- uses a memory hierarchy with configuration and parameters typical of that of the high-performance low-power embedded processors with virtual memory that are currently commercially available, and uses workloads representative of the applications that are typically run on such systems;

- investigates some of the issues involved in the provision of memory protection with virtually-addressed caches comparable to systems with a conventional TLB.

A literature survey is presented in Chapter 2. This begins by providing an overview of the memory hierarchy and an overview of caching and address translation to provide context for this thesis. This is followed by a discussion of cache models from which timing and energy consumption can be found for a given cache and TLB configuration. The address translation options available for a virtually-addressed cache are considered next. Then the benefits and disadvantages of virtually-addressed caches are explored. Finally a description of several processors that have utilised a virtually-addressed cache is provided, together with a description of how address translation and memory protection have been achieved on these architectures.

In Chapter 3 the experimental method utilised is detailed. The objectives of the experiments are outlined, and then the simulation environment, benchmarks and metrics used to achieve these are presented. The baseline configuration and experiment is described as well as the parameters varied for the sensitivity analysis.

Chapter 4 presents the results from the simulations conducted and an analysis of these results.

Finally, Chapter 5 presents the conclusions from the above work.

Chapter 2

Background and Related Work

2.1 Overview

In this chapter the background necessary to understand the rationale for this study and the experimental approach taken is presented. The chapter begins with a discussion of the memory hierarchy, providing context for the following discussion on caching and virtual memory.

The basic operation of caches is first outlined. The components of the memory subsystem that are of interest when considering virtual memory overheads are the data and instruction caches and TLBs (usually fully-associative caches). Then several salient cache parameters are explained. This includes a brief overview of cache addressing, from which the operation of virtually-addressed caches can be understood.

In order that a quantitative assessment be made of time and energy consumption of cache operation, it is necessary to have a model for a cache that, given the relevant cache parameters, is able to calculate these costs. While there are many different ways to model a cache, an analytical method from related work in the literature is reviewed here. This model is presented after the cache parameters above.

Next, the address translation alternatives on virtually-addressed caches are enumerated, and the related work in the literature is reviewed. The benefits of and considerations when using virtually-addressed caches are discussed here, providing justification for the premise and contribution of this thesis, namely that a virtual memory system utilising virtually-addressed caches may be advantageous in embedded systems and, consequently, that a quantitative assessment of the address translation options on virtually-addressed caches, in this context, is worthwhile.

Finally, several example processors that have used virtually-addressed caches in part of their memory hierarchy are detailed, with the aim of understanding how virtually-addressed caches affect and are affected by the other components of a processor's architecture. The means by which the considerations involved in the use of virtually-addressed caches, introduced in the section previous, may be dealt with are also included here.

2.2 Memory hierarchy

There are many different memory technologies for the storage and retrieval of data. Each of these is characterised by parameters such as access time (latency), throughput (bandwidth), energy consumption, and area (cost). From the application perspective it is desirable to provide a large amount of memory with low access time, low energy consumption, high bandwidth and low cost. However, this is not possible as technologies with, for example low access times, tend to be more expensive than those with high access times.

The solution to this problem is the organisation of these memories in a *memory hierarchy* [44]. An illustration of the memory hierarchy is provided in Figure 2.1. As the hierarchy is descended, access times and energy consumption for accesses increase, and cost decrease, providing for greater capacities. Compilers, operating systems and hardware provide an abstraction of the memory hierarchy implementation, presenting a consistent interface to the application programmer. The difference in times and costs between levels may be many orders of magnitude.

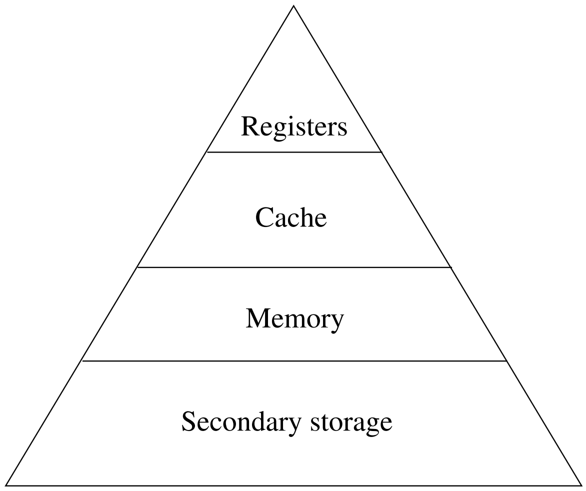

Figure 2.1: The memory hierarchy

The memory hierarchy is intended to provide memory of the capacity and the cost of the lowest level with the access time, energy consumption and throughput of the highest level. This is achievable as a result of the *principle of locality* [44]. Memory accesses tend to exhibit two types of locality. An access to a memory location is likely to be followed by an access to a nearby memory location — this is known as *spatial locality*. Also, a memory access is likely to be to a recently accessed memory location — this is known as *temporal locality*. As a result of the principle of locality it is possible to have smaller memories at higher levels in the hierarchy which are sufficient for most memory accesses and to only reference lower levels in the infrequent case in which the data required are not present at a given level.

2.3 Caches

A cache can be defined as "any storage managed to take advantage of locality of access" [31]. In the context of this thesis, however, a cache is a small high-speed memory that is placed between the processor core and main memory. Modern architectures have one or more levels of cache which provide lower average access times and energy consumption for the data and instructions in main memory.

In the following two sections an overview is provided of the way in which a cache may be organised, its basic operation, as well as the timing and energy consumption model selected for this thesis.

2.3.1 Cache operation

The basic operation of and design choices for a cache are now reviewed below, in order to provide context for the discussion of virtually-addressed caches later in this chapter.

In addition to the below, Smith [40] gives cache fetch algorithm, line size, replacement algorithm, main memory update algorithm, multiprogramming, multicache consistency, cache size, multilevel organisation, and cache bandwidth as the design alternatives to be considered for caches.

2.3.1.1 Cache size and associativity

A cache can, in general, be considered to consist of a number of same-sized sets (see Figure 2.2) with each set consisting of a number of entries, each with the fields described in Table 2.1.

The fields in each entry may be grouped into the tag and data fields. The tag is used in the lookup to select the entry, and so the additional fields to the address, such as the *address space identifier* (ASID) and *valid bit*, may be considered as additional higher order bits concatenated to the address. On many systems each process is provided with its own virtual address space, with an ASID used to identify a processes' virtual address space. An ASID is generally used to tag the TLB rather than the cache, but will be considered later as part of the cache tag in the memory protection experiments together with the, also optional, protection bits in the cache entry.

A cache lookup begins with an address being presented to the cache input, as indicated in Figure 2.2. The index bits in the address are used to select a set, and then the tags of the entries in the set are compared with the tag bits in the address. The data fields are returned by the cache if a matching tag is found, a cache *hit*. If no matching tag is found the cache lookup is said to have been a *miss*, and the line data needs to be fetched from memory and loaded into the cache.

There are three different types of cache miss [31]:

- *Compulsory misses* occur when the cache is empty, also called *cold*, and new entries are bought into the cache without displacing any existing entries in the cache. This might occur at processor startup or after the cache is flushed.

- *Conflict misses* occur when a new entry displaces an existing entry in a set due to the competition between entries within a set. These would not occur if the cache were fully-associative.
- *Capacity misses* occur when a new entry displaces an existing entry in the cache due to the size of the cache not being sufficient, i.e. even if the cache were fully-associative the miss would still occur.

Note that this description of cache operation is a logical one, and the actual implementation and how it is achieved in practice is described in Section 2.3.2.

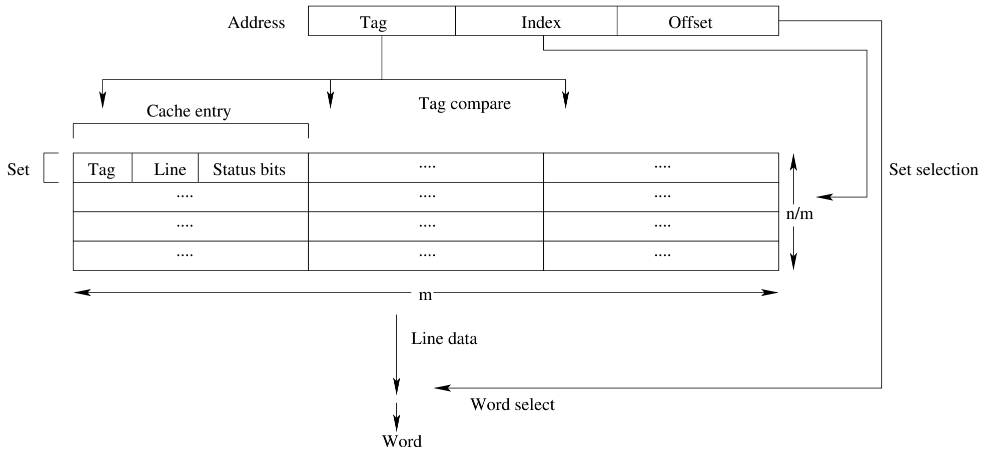

Figure 2.2: Cache associativity

Field	Size (bits)	Description
Tag		
ASID	k	Address space identifier (optional)
Address	VA $- \log_2 n/m - \log_2 l/8$	Address tag
Valid	1	Cache line is valid
Data		
Dirty	1	Cache line is dirty
Protection	p	Protections bits (optional)
Line	l	Cache line data

Table 2.1: Cache entry fields, where VA is the number of bits in the virtual address.

A *fully-associative* (FA) cache of n entries is composed of only one set with n members. A *direct-mapped* (DM) cache of n entries consists of n sets each with one member. A cache of n entries in which there are n/m sets each containing m members is referred to as an m-way *set-associative* (SA) cache.

While a larger cache size will reduce the number of capacity misses and a higher associativity will reduce the number of conflict misses, larger cache and higher associativities will also consume more energy, have a higher access time, and have a larger area requirement. Figure 2.3 demonstrates this relationship for access times. A balance between these costs and the miss costs is sought when selecting cache parameters, and will depend on workload behaviour.

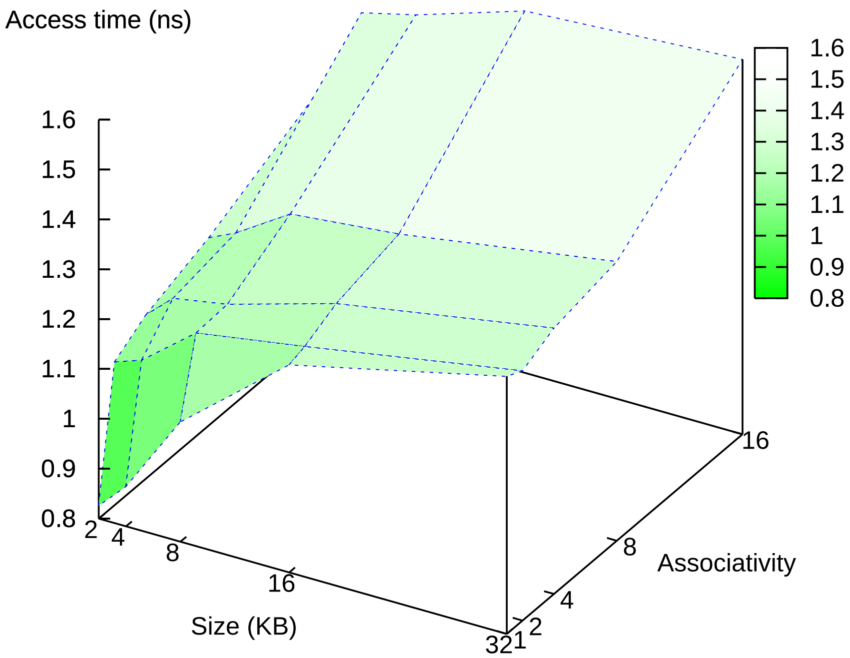

Figure 2.3: Cache associativity and size vs. access time at 0.18 microns and 32 byte line size from CACTI 3.0.

2.3.1.2 Addressing

Virtual memory provides a mapping from a set of addresses, the *virtual address space*, to another set of addresses, the *physical address space* [31]. This supports the memory hierarchy abstraction between the main memory and secondary storage layers, by allowing applications to refer to a larger address space than would be possible if only the physical address space was available. The operating system and hardware transfer fixed-size *pages* of data between secondary storage and main memory on demand. Other abstractions are also provided by virtual memory, such as memory protection and distributed shared memory.

The address in Figure 2.2 may be either the virtual address produced by the processor or the physical address used to access memory. The process of obtaining a physical address given a virtual address is referred to as *address translation*. On a memory access, address translation is performed through a data structure called the *page table*.

As the page table is also stored in memory, this would at the minimum double the effective access time and so the page table entries (PTEs) that specify translations are typically cached in a high speed associative memory called the translation-lookaside buffer (TLB). A more detailed explanation of how this may be achieved is presented in Section 2.4.

The options for cache addressing are [8]:

- Physically indexed, physically tagged (P/P) cache

  The address presented to a P/P cache is physical. The virtual address from the processor may be translated either prior to or in parallel with the cache lookup.

  In Figure 2.4(a), the TLB first translates the VPN component of the address before cache lookup. This adds extra latency to cache lookup, although can be hidden by pipelining [8].

  To reduce this latency, in Figure 2.4(b), the *virtual page number* (VPN), i.e. the higher order bits of the virtual address that specify the page, is translated in parallel with the cache indexing independently using the displacement component of the address. Since there are many physical addresses that will share a displacement, the indexed physical tags are compared with the results of the TLB translation, the *physical frame number* (PFN), and if a match is found a cache hit occurs. As only the displacement is used in the cache indexing, the cache will only have one page size of addressable locations. That is, the cache size will be fixed at the page size multiplied by the associativity of the cache. This effectively puts a bound on the maximum P/P cache size with parallel translation, as an increase in size will require an increase in associativity, which results in a higher access time as indicated in Figure 2.3, as well as energy consumption.

  P/P caches are generally used at the bottom of the cache hierarchy.

- Physically indexed, virtually tagged (P/V) cache

  The set is selected using lower order bits of the address, which correspond to the physical address, however the tag comparison is performed using the virtual address.

  This is an unusual addressing scheme, however an example of where a P/V cache has been used is the MIPS R6000, where translation is achieved through a *TLB slice* [45]. Cache lines are both virtually and physically tagged, but physically indexed. Rather than restricting the index to only the physical bits in the displacement, as a P/P cache does, the TLB slice translates the lower order bits (in

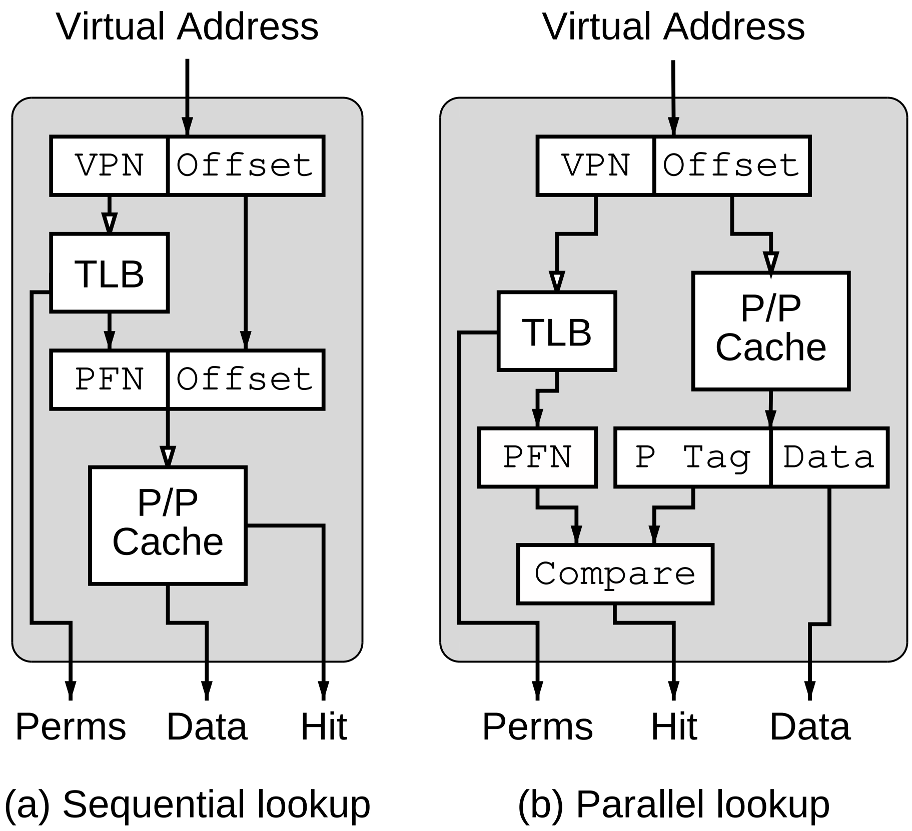

Figure 2.4: P/P cache lookup

the case of the R6000 3 bits) of the VPN. This removes restrictions on the cache size, and because the TLB slice is direct-mapped and small, it can perform the translation prior to cache indexing with low latency. The TLB slice requires significantly less area than a conventional TLB, consisting of only 16 4-bit entries on the R6000.

Once indexing has occurred, the virtual tag is compared with the requested virtual address and a hit occurs if they match. The physical tag is required for miss handling. Miss handling is complicated as a failed comparison may indicate either a TLB slice miss or a cache miss. Due to the high miss rate, Taylor et al [45] suggest the TLB slice be used in a multilevel cache hierarchy, with a V/V cache, described below, at the first level, shielding the P/V cache on the second level in order to reduce the TLB miss rate as they demonstrate in simulation.

- Virtually indexed, physically tagged (V/P) cache

  V/P caches also avoid the restrictions on cache size presented by parallel TLB

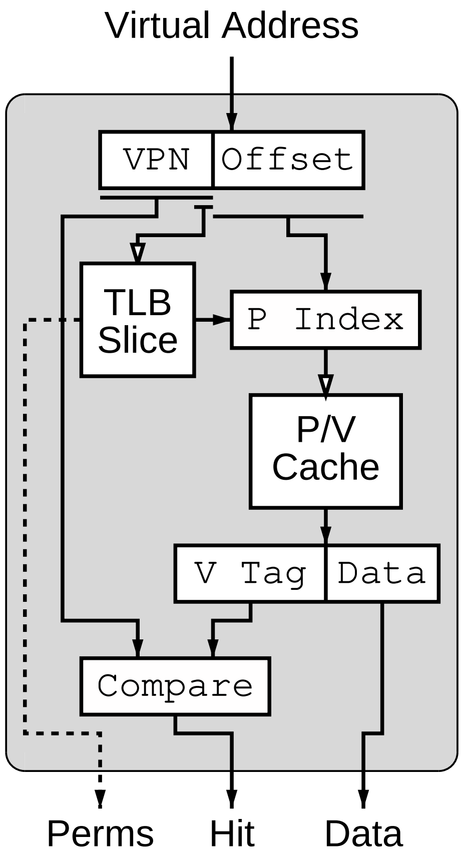

Figure 2.5: P/V cache lookup

lookup in P/P caches, by using some lower-order bits of the untranslated VPN, called the *superset bits*, during cache indexing (see Figure 2.6).

There may exist in a V/P cache more than one entry with the same physical tag. Such entries, in which multiple virtual addresses map to the same physical address, are known as *synonyms*.

- Virtually indexed, virtually tagged (V/V) cache

  V/V caches, also referred to as *virtually-addressed caches* and *virtual address caches*, require no translation to be performed prior to cache lookup (see Figure 2.7). Indexing and tag matching are performed with the virtual address. Translation is necessary on a cache miss however, and on several architectures a TLB is consulted in parallel with the cache lookup to also provide memory protection. V/V caches suffer from the synonym problem described above, as well as other problems described in Section 2.6. The motivation for the use of V/V caches is

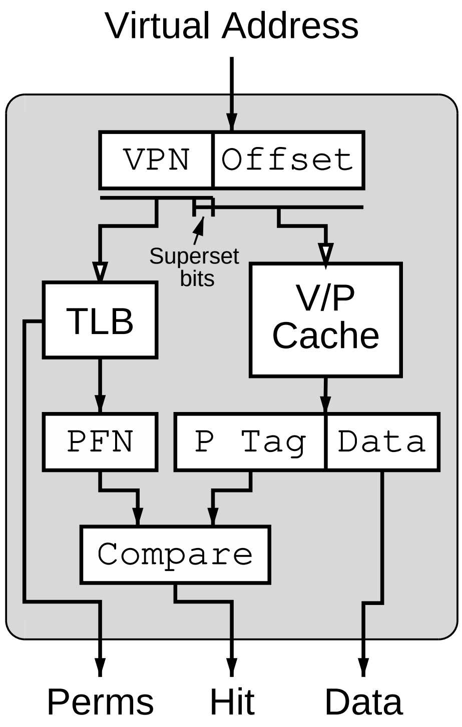

Figure 2.6: V/P cache lookup

the possibility of the elimination of the TLB from the critical path on a cache hit, resulting in a faster cache access and lower energy consumption, and avoiding other problems associated with a conventional TLB. Section 2.5 describes these advantages in greater depth.

The other cache design parameters considered in this section will be with respect to V/V caches.

2.3.1.3 Write policy

When a cache line is modified, its contents may be written immediately to memory, a policy called *write-through*, or have the updating of memory deferred until the cache entry is replaced, a policy referred to as *write-back*.

Write-through increases the memory bus traffic, and hence results in greater energy consumption and latency. The latency may be hidden by the use of *write buffers*, which buffer the written data until the memory bus is available for further writes, allowing the process to continue execution. One potential advantage of the extra memory bus traffic generated by write-through caches is that it may be used to simplify cache consistency on multiprocessor systems [40].

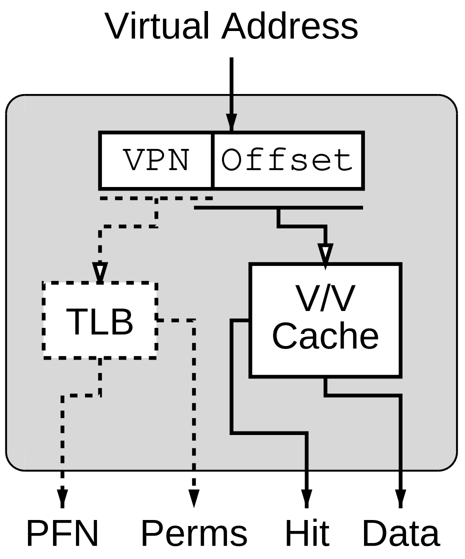

Figure 2.7: V/V cache lookup

When using a virtually-addressed cache, a write-back cache will allow the translation for the line that is to be written back to also be deferred until replacement, at the expense of increased miss penalty. When replacement occurs, translation needs to occur anyway for the new line. Most architectures that have used virtually-addressed caches have used write-back caching. Some architectures, for example the StrongARM [3], avoid this additional translation by storing the physical address for the line in the cache. For this reason, the rest of this thesis will consider write-back caches when a choice needs to be made.

2.3.1.4 Split vs. unified

By providing separate caches for data and instructions, a number of benefits can be gained over a unified cache [40]:

- the cache bandwidth increases as it is possible to access the data and instruction caches simultaneously;
- the separate caches can more easily be placed physically closer to the appropriate stage in the pipeline;
- some architectures, for example the MIPS R8000 and Alpha 21264, take advantage of the instruction cache being read-only, by using a virtually-addressed cache for the instruction cache, without needing to provide provisions for both single and multiprocessor cache consistency.

A major disadvantage of this design is that the miss ratio may increase, if the cache memory is inefficiently split. Smith [40] found that this may be significant but workload dependent.

Other disadvantages include poor performance for self-modifying code and *just-in-time* (JIT) compilers, which modify the instruction code during application execution, hence requiring the instruction cache to be synchronised with the contents of memory and the data cache whenever a write is made to the instruction code to maintain cache consistency.

2.3.2 Cache models

In order that a quantitative assessment can be made of the relevant cache characteristics, such as time, power and area, and their relationship to the cache parameters, it is necessary to use an appropriate cache model. Such a model could vary from a detailed implementation taking into account specific information about the technology process used to implement the cache, simulated under SPICE [36], to an analytical expression that approximates the relationship between the characteristics and parameters.

The *simulation program with integrated circuit emphasis* (SPICE) is a circuit simulator that may be used to model the analog components, e.g. resistors, capacitors and transistors [36], that are used to implement a cache on silicon. Each of these components is represented by a model that may consider details that are highly dependent on the implementation and fabrication process. For example, the MOSFET model includes parameters such as oxide thickness, process transconductance and substrate doping [36], all process dependent. This level of detail for the modelling purposes in this thesis is undesirable, as results will be coupled to a particular process. In addition, the issue of simulation time is of concern. If a highly detailed model is used, it will require the actual data and addresses present at each cycle to be simulated. This applies to any low-level simulation, not just SPICE. The need for this can be seen from the dynamic power dissipation of a CMOS inverter [36], a basic building block of digital circuits such as caches:

$$P_D = fCV_{DD}^2$$

where P_D is the dynamic power dissipation of the inverter, C is the sum of the intrinsic capacitance of the inverter and the capacitance by which the inverter is loaded, V_{DD} is the supply voltage, and f is the frequency at which the input is switched. The data and address lines, and all others that depend on these, will switch at different rates, data and address streams dependent. Since SPICE simulation is very time consuming, it would be impractical to simulate long data and instruction streams of realistic workloads that often are in the order of billions of references long.

Therefore a model is desired that captures the essential relationships between cache parameters and characteristics, making appropriate approximations to eliminate the data and address stream dependency. It is the relative access times, for example, that are important, not the absolute timing which would require a detailed model of the implementation.

CACTI [33, 37, 50] is an integrated model that, given cache size, line size, associativity, feature size, number of read and write ports, tag size, output width and the number of banks, produces optimised access and cycle times, power consumption, aspect ratio and area for a cache. Such an integrated model is desirable over the alternative of using separate models for time, power and area as a common set of assumptions is made in the single model for each of these [33]. The analytical models on which CACTI are based are now briefly described to present an appreciation for the depth at which modelling occurs at.

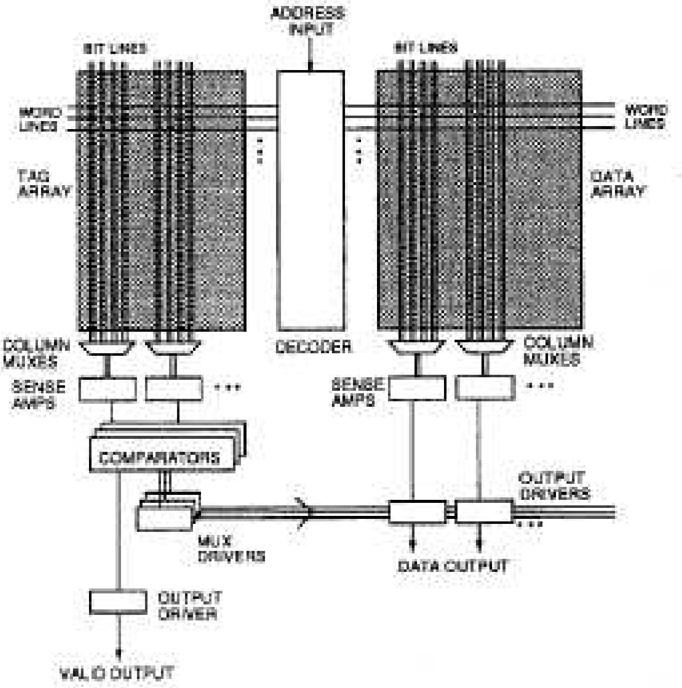

Figure 2.8: Cache structure for the set-associative model in CACTI from [50].

The cache structure that is modelled in the set-associative cache model is shown in Figure 2.8. The decoder drives a single word line in the tag and data arrays, causing the contents of that lines to appear at the outputs of the sense amplifiers. This is the cache indexing described above. The comparators allow for comparison of the tags from each set with the address, and if a match is found, *valid output* is driven high and the appropriate multiplexer selects the data block from the set that the tag was found [50]. CACTI models the decoder, word lines, bit lines, sense amplifiers, comparators, multiplexor drivers and output drivers in this process [50]. The access time will be given by:

$$t_{access} = \max(t_{data\ path}, t_{tag\ path}) + t_{output\ driver}$$

CACTI 2 [33] improved on this model by improving tag path delay and making the tag and data paths more balanced, as well as adding support for process and supply voltage scaling and multiported caches.

CACTI 3 [37] includes a model of the area and geometry of the cache, and improves the power modelling by reducing the static power consumption of the sense amplifiers when in the precharge state. This was achieved by turning turning off part of the sense amplifier when not needed.

CACTI has an optimisation function which adjusts the cache layout and geometry to meet the desired criteria. By default, CACTI optimises for power and time with approximately equal weighting, while area efficiency and aspect ratio contribute approximately at least an order of magnitude less [37].

CACTI produces two timing results, one for access time and one for cycle time. Cycle time is lower than the access time as it is wave-pipelined.

The fully-associative cache model has only a single path, with a fully-associative tag decoder driving a single word-line in the data array.

The authors of CACTI have also verified the model against an HSPICE cache model and found a maximum error of 6% for CACTI 1. The CACTI 3 area model had a maximum error of 10%.

In this thesis, CACTI 3 is used for modelling the caches and TLBs. The CACTI model does have some limitations. It is unsuitable for directly modelling highly-associative low-power CAM-tag based caches [55]. The use of this model in this thesis will consequently restrict the consideration of cache set-associativities to less than or equal to 16-way.

2.4 Address translation

The alternatives for the placement of address translation, as described by Wood [51], are shown in Figure 2.9. Translation at *A* corresponds to a TLB placed before the cache, as may be the case with the physically-addressed caches described in the last section. Translation at *B* corresponds to a TLB performing address translation in parallel to the cache lookup. This may be used with the P/P, V/P and V/V caches described above. Only with V/V caches are options *C* and *D* available. At *C*, virtual addresses are used for cache lookup, and only on a miss, when a physical address is needed to address the memory bus, does translation take place. The difference between *C* and *D* is non-existent on a uniprocessor system, however, on a multiprocessor system, the memory bus is physically addressed with option *C* and virtually addressed with option *D*.

In this section the page table data structures that the TLB caches are described and then the different types of TLB that may be used to cache address translation at each of the different translation options, *A, B, C* and *D* are described. These are the address translation options for virtually-addressed caches that are compared in the experimental part of this thesis, and so a detailed description of some of the past work, including experimental results, in the literature is presented here.

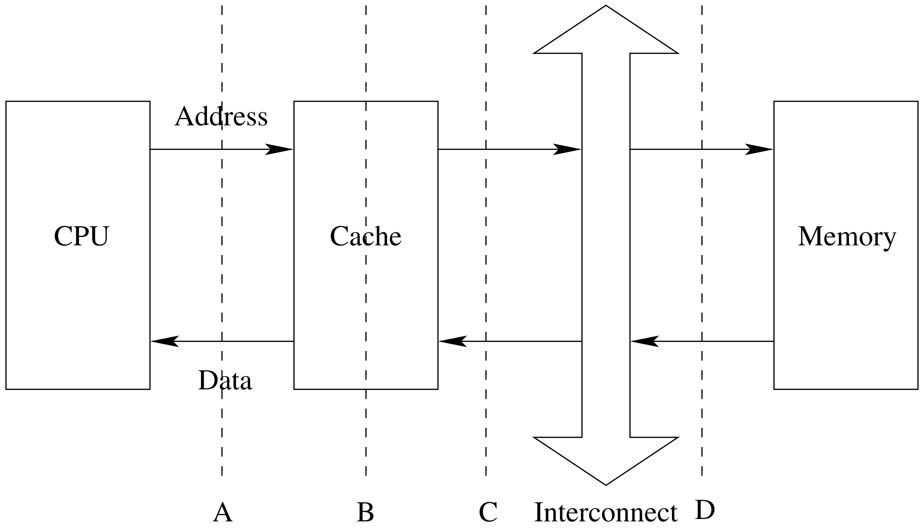

Figure 2.9: Alternative address translation placements from [51].

2.4.1 Page table

A page table is a data structure that stores the virtual-to-physical address mappings for the virtual memory system. A simple way to structure this would be to use a linear array in physical memory. This would have the advantage of a fast lookup time, which is desirable as the page table lookup operation is a major component of the TLB refill cost. Unfortunately, this is wasteful of physical memory and does not scale with larger address spaces. For example, consider a 32-bit virtual address space, with 4KB pages and a 24-bit physical address space with 4KB frames. Assume each page table entry is a single word, 4 bytes. The frame number can easily be stored in the PTE, requiring only 12 bits. However, the array will require 2^{20} PTE locations, and hence require 2^{22} bytes of contiguous storage in physical memory. Even though many of the PTEs may not hold valid mappings, the space is still required to implement the array structure. This presents significant overhead on embedded systems with a small amount of physical memory, for example a system with 64MB of SDRAM with four processes executing in separate address spaces will have 25% of memory used for just page table storage. At larger virtual address space sizes the overhead quickly overtakes the storage capacities of physical memory.

To improve on this, a tree structure may be used to organise the page table, a *multi-level page table* (MLPT). A variation of this is the location of the array in virtual memory, an organisation known as a *virtual linear page table* (VLPT).

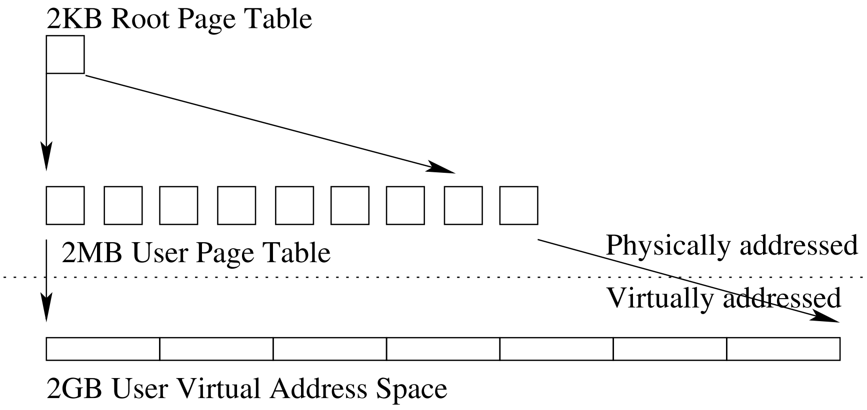

Figure 2.10: A multi-level page table, based on the BSD/Intel page table organisation described by Jacob and Mudge [19].

2.4.1.1 Multi-level page table

As indicated in Figure 2.11, the virtual address is broken up into a number of fields. The *page offset* is ignored as it is identical in both the virtual and physical addresses. The page table lookup begins with the *root page table index*, which is used to select an entry in the root page table. For the 2GB virtual address space in Figure 2.10 this requires 2KB, as there will be 512 locations each storing a pointer to tables in the next level of the table. The *user page table index* is then used to index the user page table pointed to by the relevant entry in the root page table, and the PTE is either found or a page fault exception generated. Each user page table will require 4KB of contiguous space if 4 byte PTEs are used.

This has the advantage of only requiring user page tables for the parts of the address space that have valid mappings, and not requiring the user page tables to be contiguous in physical memory. However, every successful page table lookup will require two memory accesses, doubling this component of the TLB refill cost.

31	22	12	0
	Root PT index	User PT index	Page offset

Figure 2.11: The index and offset fields in a virtual address.

2.4.1.2 Virtual linear page table

If the linear array is placed in virtual memory, then the space inefficiency can be avoided by not mapping unused parts of the page table to physical frames, and while the array in virtual memory is contiguous, it is mapped at page granularity to frames in physical memory, and so does not need to be contiguous in the physical address space. This also has the advantage of allowing the virtual memory system to swap out infrequently used parts of the table to disk through the same mechanism that user pages are swapped with. See Figure 2.12 for an example of this organisation.

This allows a best case lookup time of one memory access as the table may be directly indexed with the upper bits of the virtual address. However, because the table is in virtual memory, a virtual-to-physical mapping for the table itself will be required to be found before the PTE can be returned. If this is in the TLB or a V/V cache is used and the PTE is present in the cache, only one access will be necessary. If this mapping is missing, a root page table which, as with the MLPT, is located in the physical address space, and contains the base addresses of all the user page tables, is consulted, requiring a second memory access.

A VLPT will result in PTEs in the TLB and cache for the page table occupying space, possibly conflicting with the application's virtual-to-physical address mappings, unlike the MLPT.

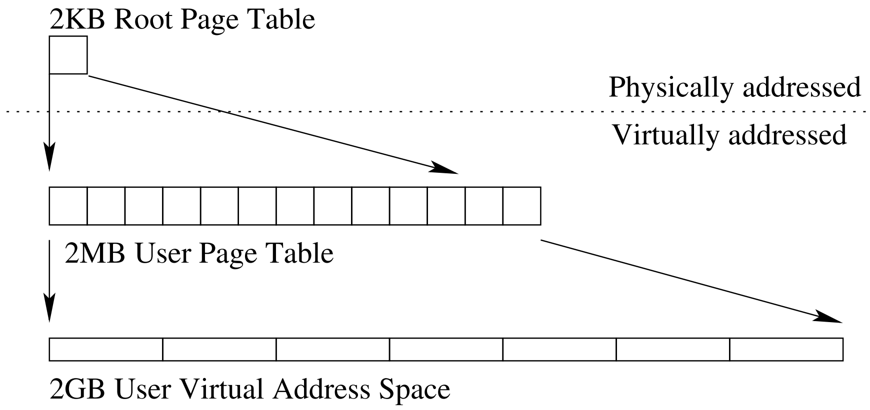

Figure 2.12: A virtual linear page table, mapping the same address space as the MLPT in Figure 2.10.

2.4.2 Translation Lookaside Buffer

Since performing a page table lookup on every memory access would present a massive overhead, the virtual-to-physical mapping are cached in a *translation-lookaside buffer* (TLB). Since the TLB misses are infrequent, on a TLB miss, the page table lookup mechanism, the *TLB refill handler*, may be either implemented in hardware or software. Hardware implementations are the fastest, but fix the page table organisation for a processor. Hardware implementations also do not have the interrupt overhead of software refill handlers, and may not even require a pipeline stall as they may execute concurrently with the pipeline [19]. Software implementations are suitable when the TLB miss rate is low enough, allowing the page table to even be tailored to a particular application, and have a lower hardware complexity. A compromise between the hardware and software approaches is also used, where a hardware walker implements the first step in the page table lookup, and on the infrequent case that this also misses, a software walker executes a complete page table lookup.

The placement of a TLB will affect its expected functionality and use. The options available here are described below.

2.4.2.1 Conventional TLB

A conventional TLB is a TLB that is consulted either sequentially before the cache lookup or in parallel to the cache lookup, i.e. at *A* or *B* in Figure 2.9.

Conventional TLBs are usually fully-associative caches. This eliminates conflict misses and allows the TLB to support multiple page sizes, known as *superpages*. Efficient use of superpages allows for an increase in TLB coverage and hence a decrease in the TLB miss rate and overhead. Fully-associative caches allow any number of power-of-two page sizes to be used, but set-associative organisations only allows two different page sizes [43]. Conventional TLBs are limited in size due to the access time contributing to the hit cost of the cache and processor cycle time. See Section 2.5.1 for further discussion of this. Other means by which capacity misses on conventional TLBs are reduced include *subblocking* [42], in which multiple adjacent PTEs are held in a single TLB entry, reducing the tag overhead and accommodating more PTEs for a given TLB size.

Besides address translation, a conventional TLB typically provides additional functionality:

- The memory protection mechanism: the TLB lookup involves a check with the cached PTE to establish whether or not the executing process has access to the memory at the virtual address requested. The handling of protection in the TLB causes some problems, however. Multiple TLB entries may be needed to support sharing of the same page on some TLBs and so translation information will be duplicated and the coverage of the TLB reduced. The desired granularity of translation and protection are also not identical and this can, for example, cause problems such as false sharing on systems that implement distributed shared memory [25].

- When a page needs to be swapped in from secondary storage, another page will need to be swapped out to make room for this. A replacement algorithm selects the page with the goal of choosing the page that will least likely be accessed in the future. Since an optimal algorithm is not possible in practice, algorithms such as *least recently used* (LRU), which selects the page that has been used in the past least recently for swapping, are utilised to approximate this desired behaviour.

  Reference bits are used in some page replacement algorithms to approximate an LRU policy. The reference bit is maintained in the PTE and stored and set on access to that page in the TLB. When the reference bit is reset as part of the page replacement algorithm, the TLB entry will be modified as well as the PTE.

A dirty bit is also maintained in the PTE and used to avoid swapping to disk, unnecessarily, unmodified pages. This is typically stored, and set on a write access to that page, in the TLB.

On systems that do not use a conventional TLB it will be necessary to update the reference bit on memory accesses and to be able to reset this from the replacement algorithm when necessary. It will also be desirable, although not entirely necessary for acceptable performance [53], to update the dirty bit on write memory accesses.

- On a cache miss it is necessary to write back dirty lines. In a virtually-addressed cache, as well as requiring the translation of the requested address, it is also required that the line being written back have its virtual address translated. This may be implemented, for example, through the TLB, by storing the physical address in the cache entry, or by performing a page table lookup. Storing the physical address in the cache entry is the least complicated, but has significant hardware overheads. Performing a page table lookup may be the least hardware intensive, but has an increased miss penalty and deadlock conditions [51] may arise depending on page table organisation which may complicate the hardware necessary to prevent this.

2.4.2.2 Off-chip TLB

If the TLB is placed behind the cache, at position *C*, the TLB is removed from the cache hit path, and so may be moved off the CPU chip or away from the processor core. This organisation will also allow larger TLB sizes, with the TLB access time no longer as critical.

The MIPS R6000 used a large off-chip TLB, part of the second level cache, which was 2-way SA and had 4096 entries. The use of a smaller TLB slice allowed full translation to be deferred to a cache miss [45] on this processor.

Whether an off-chip TLB actually improves performance enough to justify its presence or cost is questionable, and in Section 2.4.2.4, several architectures that have eliminated the TLB, but performed translation at position *C*, are surveyed.

2.4.2.3 Memory-based TLB

The TLB can be placed at the other end of the memory bus, position *D*, a configuration referred to as a *memory-based TLB* [46]. Memory is addressed virtually, with a TLB on each memory bank providing translation for the pages that have corresponding frames in the bank. The TLB may even be able to cover all the mappings for the bank, resulting in a TLB miss only on a page fault [9].

One example of this is the Sun SBus [12], which in some configurations has a centralised TLB, through which the CPU and DMA devices may perform address translation.

Multiprocessor TLB consistency, discussed in Section 2.5.4, is not a problem with memory-based TLBs when there is only one memory bank, and with no-TLB configurations. If multiple memory banks, each with its own TLB, exist then consistency needs to be maintained between these and the common page table. In addition, it is necessary to address the memory bank that contains the relevant mapping. On a bus based system this is straight forward, however on other kinds of interconnect networks this is more complicated and may place restrictions on address mappings.

On embedded systems, which typically have a single processor, a memory-based TLB is identical to an off-chip, with the same questionable benefit from its utilisation.

2.4.2.4 No TLB

Three systems have been described in the literature which eliminate the TLB completely from address translation. Both of these systems perform address translation by a page table lookup after a miss on a virtually-addressed cache. The in-cache address translation mechanism on the SPUR uses a hardware page table walker to achieve this, the same mechanism that would be used in a hardware TLB refill implementation. The SoftVM and VMP use a software page table walker.

Since the related literature has tended to focus on no-TLB configurations when investigating alternative address translation mechanisms, a detailed description of these systems and the studies performed on them is provided below. A discussion and analysis of these studies is deferred to Section 2.4.4, with the aim of clearly separating the reporting of what has been done from speculation and discussion of how this might apply to the domain of embedded systems.

In-cache address translation (SPUR)

In-cache address translation (ICAT), first used on the SPUR (see Section 2.7.2 for further description), works by storing cached PTEs in the normal cache rather than in a separate TLB [52]. This is not unique to ICAT, however, with many modern systems with a TLB also caching PTEs, such as the Intel x86 hardware page table walker and the Ultrix and Mach software page table walkers on the MIPS [19]. This caching of PTEs is the key to making the page table lookup costs on systems without TLBs practical. An overview of address translation on the SPUR is provided below.

On any memory access the global virtual address was first calculated by using the two most significant bits in the process virtual address to index four segment registers containing the segment number. This segmentation also provided the memory protection system, with processes only able to share data at segment granularity.

With a read memory access, one of four cases was possible, in the VLPT lookup:

1. The virtual address of the memory access hit on the cache. In this case the corresponding word was produced in a single cycle. In parallel to this the address of the PTE in the virtual linear array was prepared for the miss cases below.

2. The virtual address of the memory access missed on the cache. The hardware walker attempted to load the relevant PTE from the address prepared above. If the PTE access hit on the cache, the physical address for the line was produced in a single cycle and the line was then loaded into the cache from memory. Otherwise, the next case was taken.

3. The virtual address of the memory access missed on the cache, the PTE missed on the cache and the root PTE hit on the cache. The PTE was first loaded from memory, from the physical address contained in the root PTE, into the cache, then the steps in case 2 were taken.

4. The virtual address of the memory access missed on the cache, the PTE missed on the cache and the root PTE missed on the cache. The root PTE was first loaded from memory, from the physical address in the active segment register, then the steps in case 3 were taken.

A 128KB direct mapped cache with a line size of 32 bytes was used for the simulation by Wood et al [52].

On a write access resulting in a cache hit the relevant line was updated. If a cache miss occurred, the dirty line was first written back to memory at the physical address that was stored in the cache entry at load time. The storage of the physical address was in the cache entry, avoiding the complex logic necessary to avoid potential deadlock in a recursive in-cache translation, but resulting in a larger area required for the cache [52].

The conventional page-dirty bit in the TLB was replaced with a page-dirty bit in the cache entry, in addition to the cache line dirty bit used to determine whether write-back of a cache line is necessary, that acts as a mirror of the page-dirty bit in the PTE. When a cache line was loaded, the page-dirty bit was set to that of the PTE. On a write to a cache line without the page-dirty bit set, the PTE was checked, and the page-dirty bit set on the cache line if set in the PTE. If it was not set in the PTE an exception was generated and the PTE was updated by software. The advantage of performing the PTE check and update of the cache line page-dirty bit in hardware was that changes to the dirty status of the page did not require the cache lines belonging to the modified page to be iterated through and flushed, rather the cache lines were updated lazily and without the overhead of an exception.

Wood and Katz [53] analysed other alternatives, including always faulting on writes to non-updated cache lines, flushing the page from the cache, and always checking the PTE on a write to a clean line. Due to the low frequency of excess faults observed on their benchmarks they concluded that always faulting on a write to a non-updated cache line would be a better policy, as it simplified the required logic and only performed slightly worse than the page-dirty scheme that was implemented on the SPUR. In retrospect Wood and Katz state that this would have been chosen for the SPUR if the results were available during its design. In addition, an interesting point is made in this paper that the benefits of a dirty bit were only minor, and if a system were not to offer a dirty bit the number of write-backs to disk of clean pages would only increase the total number of disk write-backs by 3%.

The reference bit usually found in a PTE was replaced with a *miss* bit in the cache entry, which was only checked when a cache line in the page was loaded. An exception was generated if the miss bit was not set and the PTE was updated by software. This only provided an approximation to the reference bit as references to lines already in the cache would not set the miss bit. Wood and Katz [53] also analysed this and the alternatives of true reference bits requiring the lines belonging to the page to be flushed from the cache on an update, or, alternatively not maintaining any reference bit. The true reference bit was always slower on their benchmarks because of the high cost of flushing, but this would depend on the time required to flush and the time required to read a page in from disk. Not maintaining a reference bit performed the best for large memory sizes, comparably for medium sizes and about 50% worse for a small memory size [53].

The performance of ICAT on the SPUR was evaluated by Wood [51]. Trace-driven simulation was carried out on a variety of traces captured from modified microcode on the DEC VAX, the SPUR instruction level simulator, and a hardware tracing system for the MC68000. Workloads used include FORTRAN/Pascal/Lisp compilers, VLSI routing, SPICE, switch-level and behavioural hardware simulations, a relational database system, and an edit/compile/debug development sequence. ICAT was found to be comparable to the performance of most TLBs that were used in the evaluation. Some of the TLBs used had similar sizes and associativity to current commercial TLBs (64, 128 and 256-entry fully-associative). ICAT was shown to perform better on larger caches, while a TLB performed better when cache size was less than 32KB, although increasing the cache associativity to 2-way set-associative lowered this to 16KB. ICAT was also sensitive to other cache parameters such as miss penalty and associativity. This was, in part, due to the competition between PTEs and other data. This is not unique to ICAT however, other architectures place PTEs in the cache for performance reasons [18].

Wood [51] also describes performance of ICAT on the SPUR prototype. ICAT was implemented on a custom VLSI cache controller chip, part of a SPUR processor. The effects of page table placement, non-cacheable pages and cache controller register accesses produced a discrepancy between simulation and prototype performance. ICAT was found to consume less than 4% of all cycles on the SPUR.

Wood [51] compared the performance of an inverted page table and the linear virtual array and found them to be within 1% of each other. An inverted page table had other desirable properties such as being less sensitive to address space utilisation and potentially consuming less physical memory.

Software managed caches

Software managed caches simplify cache miss handling and page table lookup hardware and provide greater flexibility by allowing policy to be defined in software.

VMP, described in Section 2.7.1, entered an exception handler stored in local memory on a cache miss. A two-level page table was used to perform the virtual-to-physical address translation, with recursive cache misses during lookup reentering the exception handler. Once the physical address was found, the line was loaded by the block copier into the cache.

SoftVM [18] stored the cache miss handler in the cache itself, unlike VMP in which a local memory was used for this at each node. On a cache miss, the cache miss handler was executed and needed to perform address translation to find the physical address of the faulting line. When this was found, an additional privileged instruction, added to the instruction set, called *mapped-load* allowed the miss handler to specify the physical address to be loaded at the virtual address causing the miss on the cache. Execution then resumed at the faulting address. *Mapped-load* also allows protection bits for the line to be specified. The cache miss handler code executed out of a portion of physical memory, directly mapped onto the virtual address space and a miss on the handler was handled by hardware to prevent an infinite regression. The page table used was a form of VLPT.

Traces on gcc, vortex, and winword showed that SoftVM performed comparably to the TLB based ULTRIX and MACH virtual memory systems on larger caches and line sizes, with a large (128 byte) L2 cache line size being the most important determinant of SoftVM performance.

Another possible software cache management technique is the handling of translation in cache in hardware, but handling a cache miss on a PTE in software, much like the hybrid page table walkers used with some conventional TLBs.

2.4.3 Related work

A study by Qiu and Dubois [32] compared the alternatives for the placement of translation on *cache-only memory architectures* (COMAs). This is discussed here as it also compares the address translation mechanisms available for virtually-addressed caches, but on COMAs, rather than embedded systems.

COMAs consist of multiprocessors with a distributed main memory and directory-based cache coherence. The memory at each node, the *attraction memory* (AM), acts only as a cache and the data is "migrated or replicated in main memory depending on the memory reference pattern" [41], see Figure 2.13.

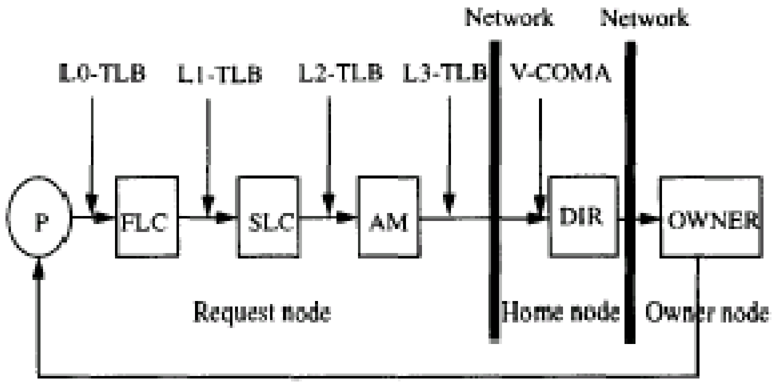

Figure 2.13: Possible locations of the TLB in COMA (from [32]).

The L0-TLB corresponds to a conventional TLB on the critical path. L1-TLB and L2-TLB place translation after the first and second level caches respectively, with L2-TLB requiring a second-level virtually-addressed cache. With all these translation options, the AM is addressed with a physical address. If the AM misses, it becomes necessary to perform a coherence operation, through, for example, a hierarchical directory [13]. Two distinct address spaces, virtual and physical, are used with these translation mechanisms, with nodes sharing a common physical address space. Placing translation after AM, L3-TLB, makes the AM virtually addressed. This is still similar to the above schemes, in that the TLB is local to each node, and requires consistency management [32].

If translation is deferred, and the virtual address is used to address the home node, the physical address space is eliminated, in a similar way to memory-based TLBs. This is called V-COMA. The conversion between a virtual address and an address in a directory at the home node is cached by a *directory-lookaside buffer* (DLB).

In the simulations run by Qiu and Dubois they observed a number of effects on address translation misses:

- *filtering effect* The number of misses in a TLB cannot be larger than the number of misses in the cache above it. This results in a decrease in address translation misses as the TLB is moved further from the processor, on a hierarchy of virtually-addressed caches.
- *sharing effect* The DLB is shared which increases its coverage when compared with the TLBs which are local to each processor node and for which duplication of entries will be necessary. This results in significantly fewer DLB misses than TLB misses.
- *prefetch effect* The translation required for the DLB occurs just once for the entry, making it more efficient than repeating the address translation for the TLB at each node.

ICAT and software managed caches were considered as zero-entry L2-TLBs by Qiu and Dubois [32]. V-COMA was found to have the lowest number of address translation misses and execution time.

Qiu and Dubois only considered a fixed cache size and a fixed number of nodes. The studies on address translation mechanisms [18, 51] have shown that cache size, associativity and line size are important parameters. Varying the number of nodes would also provide information on the scalability of the translation mechanisms. Only shared data was considered in this study.

2.4.4 Discussion

When a virtually-addressed cache is used, the alternatives of an off-chip TLB or no TLB address translation are available. Whether an off-chip TLB is beneficial will depend on whether it brings about any significant reduction in PTE pollution of the cache, and the relative cost of the TLB lookup/refill and the memory line fetch. Whether the results above are directly applicable to embedded systems will depend on how close the system parameters and costs simulated reflect those of current embedded systems. That a non-conventional TLB and virtually-addressed cache are justified is discussed further in Section 2.5.

The evaluation of ICAT and SoftVM both took place on direct-mapped caches. This might have made likely degenerate cases in which application data and PTEs competed for the same line. On modern architectures, higher associativities are common. Several embedded systems have 2-way set-associative caches, with this being an energy-efficient [16] compromise between the increased hit energy and access times with larger cache associativities and the reduction in conflict misses it brings about. The StrongARM [29] processor has 32-way set-associativity, achieved by the use of CAM-tag caches [55], which also have high energy-efficiency. Caches on embedded systems tend to be smaller, than that used in the ICAT study, in the range of 8–32KB, again being an energy-efficient compromise.

The SoftVM system has a high page table walking cost relative to the memory line fetch on embedded systems, and is assessed by Jacob and Mudge [18] on systems with second level caches, not typically present on embedded systems. Architectural features such as long pipelines and non-blocking caches may make software cache management less efficient than reported as these were not simulated.

Many of the benefits in the Qiu and Dubois study come from multiprocessor consistency improvements. With the exception of the filtering effect observed these are non-applicable to the class of embedded systems under consideration.

It would appear that a reevaluation of the above work on parameters like those found on current embedded systems is justified. In addition, none of these studies considered the energy consumption of the different mechanisms in their evaluation, a cost of great importance when evaluating the suitability of these to embedded systems.

2.5 Virtually-addressed cache benefits

While the address translation options for virtually-addressed caches have been outlined, the motivation for their use, and the use of non-conventional address translation mechanisms, is now presented.

2.5.1 Improved translation coverage

It has been found that handling TLB misses can consume from 5–40% [17, 22, 23, 34] of execution time, depending on workload. To improve this it is necessary to either improve the TLB hit ratio or the miss penalty in hardware and software, or to find alternative means of providing translation with lower overheads. For example by increasing the TLB coverage, using a multilevel TLB hierarchy, or prefetching [22] the hit ratio or miss penalty may be improved.

Virtually-addressed caches do not require translation on a cache hit. A conventional TLB can then be eliminated, removing the performance degradation from TLB miss handling when TLB coverage is poor. The improvement in performance will depend on the cost of the previous TLB refill and the relative number of TLB misses that occurred on a cache hit and miss.

2.5.2 Improved cache access and processor cycle time

Simply increasing the size of a TLB to solve the translation coverage problem is not practical. A conventional TLB is either placed sequentially before, or in parallel with, the cache. In either case, the TLB access time will increase the processor cycle time if this becomes dominant in the stage of the pipeline in which it is placed. For example, at a clock speed of 600 MHz and at 0.18 microns, the CACTI 3 model gives 128 entries as the largest fully-associative cache that could be supported. 128 entry TLBs have been at the upper end of TLB size for at least a decade (Wood compared ICAT with such TLB sizes), suggesting that TLBs do not scale.

A virtually-addressed cache allows for the removal of this conventional TLB. If the TLB was previously contributing to the cache access time or processor cycle time, its removal from the cache hit path will also yield direct speedups in these. If the TLB had contributed to the lengthening of the pipeline, improvements in interrupt/exception overheads and branch costs may also occur with its removal from the cache hit path.

A larger TLB is also practical if only consulted on a cache miss, as the increased access time will not degrade the processor's cycle time.

2.5.3 Reduced energy consumption

In addition to the time spent handling misses, conventional TLBs on the critical path may consume a significant portion of the power budget of a processor. Typically, a fully-associative TLB lookup is performed, when it is on the cache hit path, at every cycle for the instruction TLB and whenever a load or store is performed for the data TLB, on many current architectures. For example, 17% of the StrongARM SA-110 power consumption is due to its TLBs [29].

Improved TLB design [21,55] will reduce this, however virtually-addressed caches effectively eliminate this cost by only requiring translation to be performed on a cache miss.

The energy consumption of the TLB may also be reduced by a hardware mechanism that only performs a TLB lookup when a page boundary is crossed. Since for data, and especially instruction, references there may be a great amount of spatial locality this could yield an energy saving, at the expense of some additional logic on the cache access path. This is a special case of the more general use of multi-level TLBs.

2.5.4 Multiprocessor TLB consistency

A change to a virtual-to-physical address mapping or page state bits may result in an inconsistency between the TLBs at different processors on multiprocessor systems [9].

TLB shootdown [7] is a widely used software mechanism to provide multiprocessor consistency. When a processor needs to change a mapping, an interrupt is sent to other processors, the interrupt is acknowledged by the other processors which spin until the change is complete. The mapping is then changed and the other processors update their TLB with the new mapping when released. This clearly involves some overhead from the inter-processor interrupts and time wasted spinning.

Some of the alternative translation mechanisms to a conventional TLB that a virtually-addressed cache makes feasible avoid the problem of multiprocessor TLB consistency by removing the TLB from each processor and either moving it closer to memory or eliminating the need for a TLB. In the latter case the consistency of cached page table entries is guaranteed by the hardware handling of multiprocessor cache consistency. These alternative translation mechanisms make the hardware simpler and reduce the problem of multiprocessor consistency to cache consistency only.

2.6 Virtually-addressed cache considerations

The main problems that virtually-addressed caches introduce — the possibilities of inconsistency both within and between processors — are discussed below. This provides some insight into the reasons that virtually-addressed cache have been limited in use on commercial systems in the past. Any experimental approach to their evaluation will need to consider or make assumptions about these consistency issues.

2.6.1 Uniprocessor cache consistency

2.6.1.1 Synonyms

A *synonym*, also referred to as an alias, occurs when more than one virtual address is mapped to the same physical address. If entries from different pages of a synonym are present in the cache at the same time there is a possibility of cache inconsistency occurring in a virtually-addressed cache. This will occur when one of the cache entries is written to and the other synonym entries remain unmodified. Cache inconsistency may be avoided by either synonym prevention, synonym avoidance or synonym detection [8].

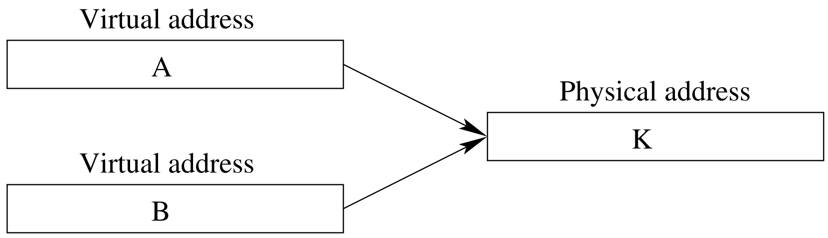

Figure 2.14: A synonym example.

Synonyms can be prevented by either using segments for sharing or by single address space operating systems (SASOSes) [49]. Both segments and SASOSes prevent synonyms by having the address that is used in the cache lookup belong to a single global virtual address space.

Architectures with segments use some bits in the virtual address generated by the processor to index a set of segment registers which then provide an address in the global virtual address space to the cache, see Figure 2.15. SASOSes [49] have a single global virtual address space visible directly to all processes. Sharing is achieved through the same virtual address, therefore no synonyms are necessary in a SASOS.

A simple way to avoid synonyms is not to cache synonyms unless they are read-only. This will degrade performance considerably for access to these regions of memory. Synonyms can also be avoided by only allowing one of the page mappings to be valid at a time. On access to one of the synonyms, the page is invalidated, flushed from the cache and the new page is made valid. This also has performance considerations

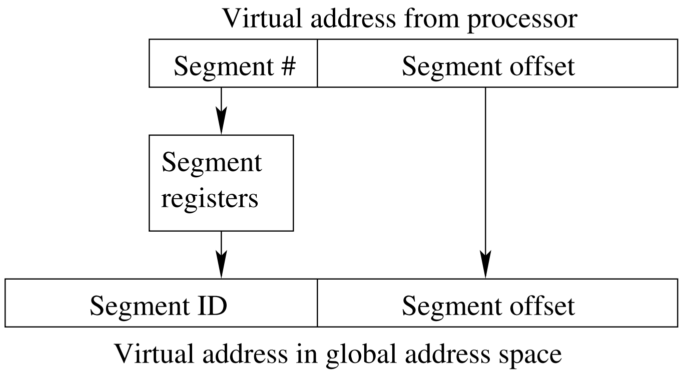

Figure 2.15: Address decoding for segments.

due to the high costs of the exception and flush required when a synonym requires the current valid mapping changed (also called remapping) [10].

For the purpose of this thesis, it will be assumed that the system prevents synonyms. The rationale for this is:

- Synonyms are a problem that affect V/V caches independently to the address translation mechanism.
- There have been a lot of solutions proposed to synonyms in both software and hardware, and it is beyond the scope of the thesis to more than briefly summarise the problem and some of these solutions.
- The performance of the above solutions is largely dependent on the operating systems features (e.g. shared libraries, fork, zero copy, etc). This is related to the entire operating system design which is well beyond the scope of this thesis.
- Since there are real systems (e.g. PowerPC [28] or a SASOS like Mungi [14] or Sombrero [39]) that do prevent synonyms it appears justifiable to make this assumption. Both the SoftVM [18] and COMA [32] papers have assumed PowerPC-like segments.
- The benchmarks considered later do not require synonyms for their operation.

2.6.1.2 Homonyms

A *homonym* occurs on architectures that support more than one virtual address space when the same virtual address in different address spaces map to different physical addresses. If the cache or TLB is not tagged with an address space identifier (ASID) then cache inconsistency may occur. This occurs because there is no means of identifying the address space to which the virtual tag belongs.

A naive solution is to flush the cache on a context switch. This will degrade performance if context switches are frequent, such as on systems based on microkernels.

ASIDs work by extending the virtual address to give every virtual address in each process a unique address in a larger address space, therefore eliminating homonyms.

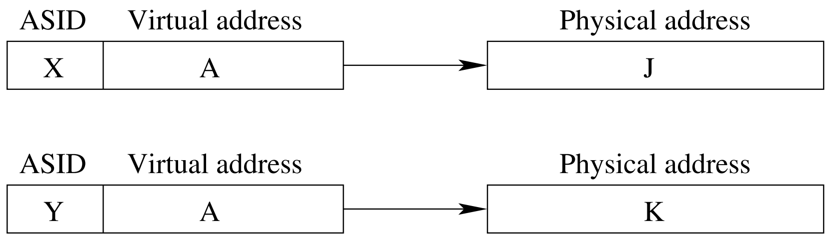

Figure 2.16: A homonym example.

A popular solution is to use a conventional TLB and access it in parallel with the cache lookup, with the TLB entries tagged with an ASID. On architectures with no TLB lookup prior to or parallel with the cache, an ASID would be part of the cache tag.

Homonyms are also avoided in a SASOS as all processes share a single global address space.

ASIDs are considered in the experimental component of this thesis.

2.6.1.3 Protection without the TLB

If the TLB is removed from the cache hit path an alternative mechanism needs to be provided for memory protection.

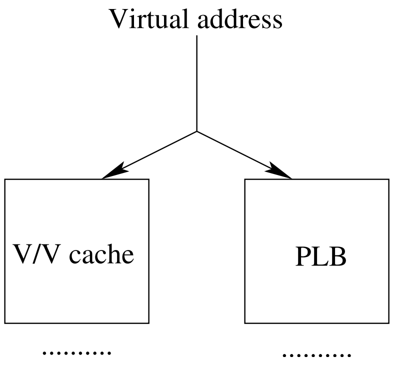

Figure 2.17: Protection lookaside buffer.

Koldinger et al [25] discuss models for separating protection from the TLB, the domain-page model and the page-group model.

The *domain-page model* is implemented by placing protection information in a separate cache called the *protection-lookaside buffer* (PLB). The PLB is similar to a conventional TLB with respect to replacement, but it does not cache address translation information. If the same protection granularity is used as with a TLB, a similar miss ratio will be experienced, and the same tradeoffs between miss ratio, access time and energy consumption will have to be considered. However, the advantage of this separation of protection from translation is that it is possible to use much larger protection pages. For example, the entire code segment may be mapped by a single page if it is appropriately aligned. A PLB also offers the possibility of finer grain protection than a single page.

The *page-group model*, used on the Hewlett-Packard PA-RISC architecture, adds to the TLB a field called the access identifier (AID), which is then used to reference a separate cache of protection information.

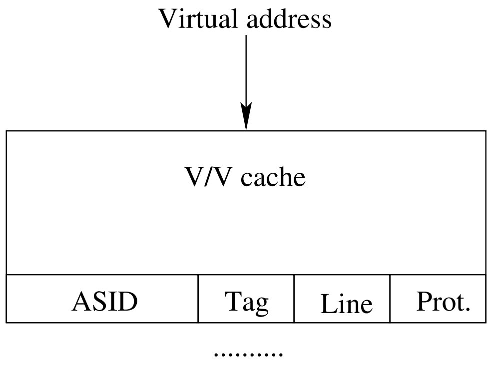

Figure 2.18: Addition of ASID and protection bits to the cache entry.

For this thesis, the addition of an ASID and protection bits to the cache line itself, as done by Jacob and Mudge [18], is considered. This will result in greater area required for the cache, and increased energy consumption from the longer tag length required.

2.6.1.4 Virtual-to-physical address-mapping changes

A change to a virtual-to-physical address mapping may cause cache inconsistency as existing lines tagged with the virtual address will still contain data from the former physical address. This will require either the entire cache to be flushed or the lines in the affected page to be selectively flushed or modified: page mapping, demapping and page out operations will all require this flushing.

Chao et al [10] discus changes to Mach to improve its performance on a virtually-addressed cache. Mach handled sharing by remapping as described in Section 2.6.1.1 on the PA-RISC virtual address cache. Chao et al improved on this by allowing multiple mappings for a synonym to exist for shared text and other read-only mappings, as well as some other minor optimisations. This reduced execution time by about 10%.

If protection is implemented on a cache line, it will also be necessary to flush or iterate through the affected cache lines on a protection change, as well as tagging the cache line with an ASID or flushing the cache on a context switch.

Jacob and Mudge [18] discuss modifications to page protection information requiring each line of that page to be updated if present in the L1 or L2 caches. On Mach3 this was given as occurring on average 11.3 times per million instructions. The workload used to derive this was compress, jpeg_play, IOzone, mab, mpeg_play, gcc and ousterhout. This gave an overhead of 2–7%, but under the Ultrix VM this would decrease by three orders of magnitude [18].

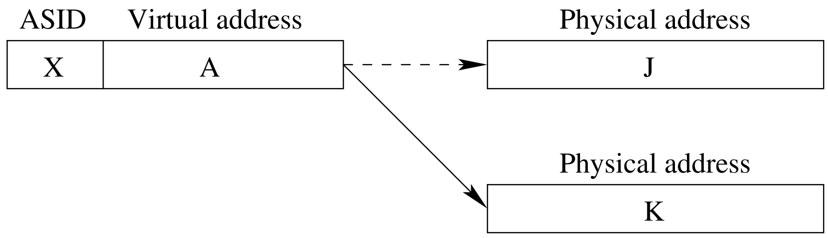

Figure 2.19: Virtual address A has its address mapping changed from J to K

2.6.1.5 Multiprocessor cache consistency

Since the same address may reside in more than one processor's caches, it is necessary to maintain the consistency of the data at this address across processors.

Systems with no synonyms have no additional multiprocessor cache consistency problems to systems with physically-addressed caches. It is sufficient to snoop on either the virtual or physical address to maintain cache consistency [9]. Snooping involves monitoring the bus for addresses held in the cache and either updating or engaging in a cache coherency operation when such a memory access is detected.

Synonyms introduce further multiprocessor cache consistency problems, but while this survey has acknowledged that there are both benefits and drawbacks on multiprocessor systems when virtually-addressed caches are used, for the class of embedded systems being considered this is not applicable, and so multiprocessor issues will not be further considered.

2.6.1.6 I/O cache consistency

Direct memory access (DMA) is a means of transferring data between memory and I/O peripherals efficiently by having a component of the peripheral device, called the DMA controller, master the memory bus and control transfers between the peripheral and memory. This is more efficient than copying data through the processor as the DMA controller may implement the copy operation through simple state machines, and may only use the bus when it is not utilised by the CPU, freeing the CPU to execute out of cache.

The DMA controller may use either virtual or physical addresses, depending on the memory bus. A virtually-addressed memory bus allows for virtually-addressed DMA, as does one with a centralised TLB such as the Sun SBus [12].

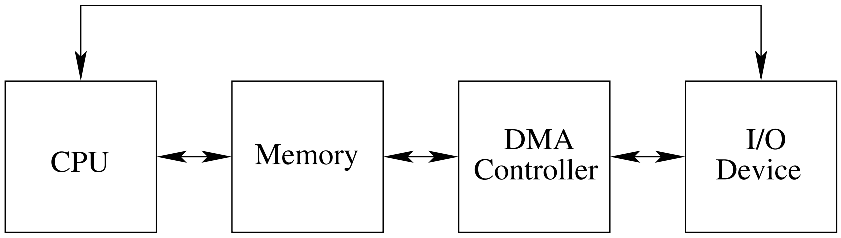

Figure 2.20: Direct Memory Access

The same consistency issues that apply to multiprocessors above apply to DMA. A simple solution is to only perform DMA to or from uncached regions of memory. This has a performance impact, so a scheme in which the target region of memory is only uncached temporarily is preferable. Another means to deal with this consistency problem is to implement the same cache coherency protocol as used for multiprocessors.

VMP [11] provided DMA through a combination of operating system support and the multiprocessor cache consistency protocol. Before a DMA operation a processor would assert ownership on the target region of memory and prevent, through the generation of cache consistency interrupts, any processor, including itself, from gaining access to this region of memory during the transfer. The assertion of ownership would force other processors and synonyms to be flushed prior to the transfer. On completion the region of memory was released to the usual cache consistency protocol.

I/O consistency will not be considered further in this thesis as the primary emphasis is on the overheads of CPU intensive applications, but should be considered as future work.

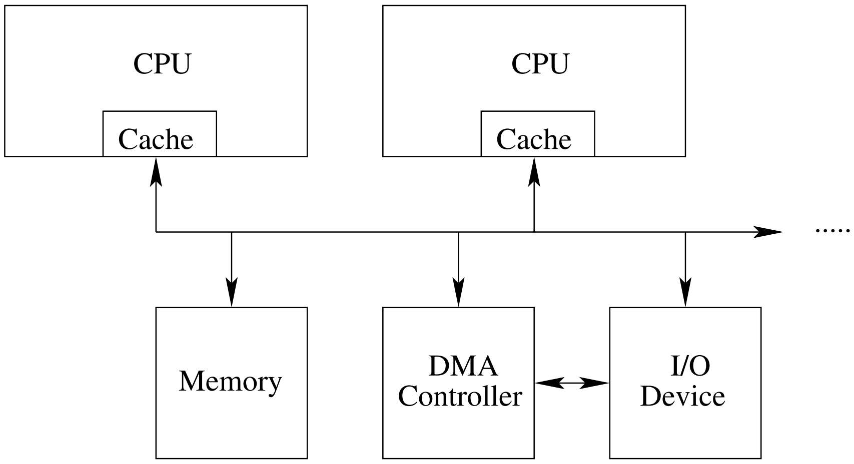

Figure 2.21: DMA on a shared memory multiprocessor.

2.7 Processors with virtually-addressed caches

Some processors that have used virtually-addressed caches are now briefly surveyed to gain an understanding of the issues that may arise when virtually-addressed caches are used as part of a larger architecture, and how the consistency problems outlined in the previous section may be solved in practice.

2.7.1 VMP

VMP was a shared memory multiprocessor with software controlled caches [11]. Each processor had a virtually-addressed cache providing a high memory bus bandwidth and a software miss handler reducing the complexity of the hardware logic. On a cache miss an exception was generated and the miss was handled by code stored in local memory.

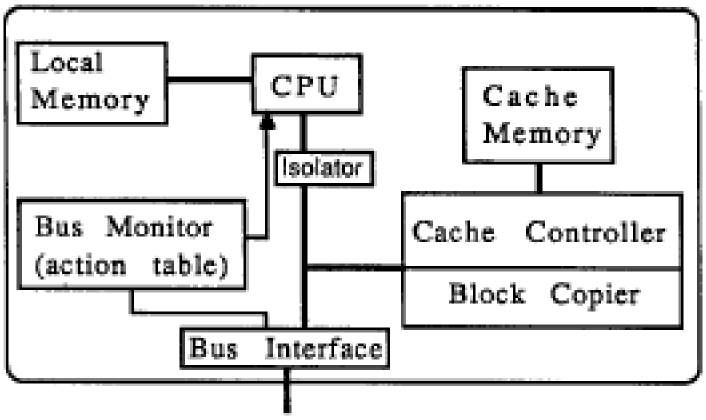

Figure 2.22: VMP Processor Board Organisation from [11]

If the line was dirty the handler performed write-back, followed by the traversal of a two-level page table by the handler to find the physical address of the required line and the use of the block copier to copy the line from shared memory to the local cache.

Synonyms were handled using the same mechanism used for multiprocessor cache consistency. State machines at each processor monitored bus transactions. Shared memory was divided into cache page frames, each the size of one line. An *action table* stored the status of each cache page frame in shared memory, at each processor. The bus monitor used this to interrupt the processor on transactions that required consistency control. Each cache page was either a shared copy, or a private copy, or not in the local cache. Shared cache pages were not writable. When a processor attempted a bus transaction to mark the cache page as private, any processor (including the requesting processor) would interrupt the requester if its action table already contained the cache page frame. Consistency was based on the physical address.

2.7.2 SPUR

The SPUR had a 38-bit global virtual address space, divided into 256 1GB segments, each divided into 4KB pages [52]. A process had access to 4 of these segments. Each of these 4 segments had an associated *active segment register*. A VLPT was used for the page table, with the physical address of the root page table for a segment held in its active segment register and the page table array in the segment's virtual address space.

In-cache address translation [52] (ICAT) was used as the address translation mechanism on the SPUR. When a memory access hit on the cache, no translation was necessary as the tag was matched with the virtual address. On a read cache miss it was first necessary to access a page table to obtain the physical address of the line to be fetched. The page table entries (PTEs) were also stored in the cache, and so the page table lookup begins in the cache. If the PTE was found in the cache then the physical address was extracted and the cache line loaded. Otherwise lookup proceeded similarly in physical memory. On writes, if necessary, the line was written back before the new line was bought into the cache. This necessitated an additional translation for the old line, and was achieved using the physical address stored in the cache line when the line was bought into the cache.

2.7.3 Intel i860

The Intel i860 was a high performance RISC microprocessor with on-chip MMU, data/instruction caches, floating point and 3D graphics units [24]. The i860 was able to work in either a scalar mode or a superscalar mode in which it could issue up to three instructions per cycle, with 64-bit wide internal instruction bus and 128-bit wide data bus to support this.

The i860 XR had a 4KB instruction cache and an 8KB data cache, both addressed virtually and two-way set-associative with a 32-byte line size. Software enforced a policy which did not allow synonyms (other than code and read-only data) and homonyms were prevented by invalidating the instruction cache and flushing the data cache on context switches [24]. According to Kohn and Margulis, this gave only a 1.3% performance degradation in the worst case with 200 context switches per second. Changes to virtual-to-physical mappings also required the cache to be flushed. Shared data was not cached on the XR, thus avoiding multiprocessor consistency problems. The later i860 XP used hardware snooping [5] and additional physical tags in the cache to maintain multiprocessor cache consistency. The data cache used write-back and read-allocation with write buffers hiding the cost of write misses and write-backs. Demand fetching was used with the required word passed directly to the requesting functional unit while the rest of the line was loaded [30].

A TLB was still required on data cache accesses to provide protection. The data cache, which had a write-back policy, and instructions which bypassed the cache and used the external data bus directly, such as the vector instructions (e.g. PFLD), used the TLB for translation. A 64-entry four-way set-associative TLB cached accesses to a two-level page table. A 32-bit virtual address space divided into 4KB pages was available to processes. On TLB misses, a hardware page table walker located the PTE.

2.7.4 Motorola MC68030

The MC68030 was a 32-bit microprocessor with integrated core, data and instruction caches, and MMU [2].

The MC68030 had split virtually addressed instruction and data caches, both 256 bytes in size, direct-mapped and with a 16 byte line size [2]. Context switches and virtual-to-physical mapping changes required the caches to be flushed. The data cache used write-through and a user configurable choice between load-allocate and write-allocate policies. When write-allocation was used synonyms were not a problem, as the data cache size was always less than or equal to a page size and direct-mapped, so synonyms mapped to the same line.

The TLB was accessed in parallel on both reads and writes, providing both write protection, a cacheability check and a physical address for cache misses and write-throughs. A 32-bit address space was available to processes, divided into 256B/512B/1KB/2KB/4KB/8KB/16KB/32 KB sized pages. The TLB was 22-entry fully-associative.

2.7.5 StrongARM

The StrongARM family consists of integrated embedded microprocessors, with a 32-bit RISC core, MMU, data and instruction caches, and PCMCIA, LCD, serial and other general purpose I/O on a single chip [3]. A high performance is available with a low power consumption, for example the SA-1100 with a 220 MHz clock rate consumes approximately 0.5 W. The StrongARM RISC core is based on ARM V4 architecture, with a simple 5-stage scalar pipeline. The StrongARM, and its descendant the XScale, are the most widely used architectures that use virtually-addressed caches today.

The SA-1100 has a split 16KB instruction cache and an 8KB data cache, both caches 32-way set-associative with a 32 byte line size. The replacement policy for both caches is round robin. The instruction cache requires invalidating on changes to virtual-to-physical mappings. In addition, a mini data cache is provided, which is only 512 bytes in size. This is intended for accessing large data structures that might otherwise thrash the main data cache [3]. The data caches both use write-back and load-allocation, with write buffers and prefetch buffer able to hide store and load latencies. The corresponding physical address is stored with the line in the data caches. The relevant entries need to be flushed from the data cache on a change to the virtual-to-physical mappings. Synonyms are avoided by requiring the operating system to map these as uncacheable.

The SA-1100 also has a split instruction TLB and data TLB, both 32 entry fully-associative. The TLB is accessed in parallel with the cache and provides protection and translation for misses and accesses to uncacheable regions. A 32-bit virtual address space, divided into 1 MB sections and 1/4/64 KB pages, is available to processes. On misses a hardware page table walker locates the PTE in a two-level page table. Protection is provided through 16 domains. A running process has its access to a domain determined by the contents of the *domain access control* register [4]. As processes are likely to have overlapping address spaces and thus require constant flushing of the TLBs and caches, the *fast context switch extensions* [4] allow the bottom 32 MB in the address space for a process to be relocated through the *PID* register.

2.8 Summary

In this chapter the basic operation of a cache and the model that will be used to derive quantitative timing and energy consumption data have been described. The address translation options for virtually-addressed caches have been enumerated, and the benefits of and considerations when using virtually-addressed caches have been outlined, and some example processors that have used virtually-addressed have been presented.

A reevaluation of the performance, and an examination of the energy efficiency, of the alternative address translation mechanisms available for virtually addressed caches has been justified and is the basis for the experimental component of this thesis.

Virtually-addressed caches have been shown to be potentially advantageous on embedded systems as they reduce virtual memory performance and energy overhead by allowing the TLB to be eliminated from the cache hit path. They do, however, present consistency problems, in particular the synonym problem, which is, for the purposes of this thesis, ignored, with a justification for this provided in Section 2.6.1.1.

Many of the example processors above still have the TLB on the cache hit path, primarily to provide a protection mechanism. It therefore worthwhile comparing this, as done in [18] for performance, for energy efficiency with in-cache access rights.

Chapter 3

Experimental Method

3.1 Overview

In this chapter the experimental method that was used to evaluate the address translation options for virtually-addressed caches, described in the previous chapter, is presented. The intention of this comparison was to quantify the energy and cycle overhead improvement that might be achieved by the use of non-conventional TLB and TLB-less configurations over conventional sequential or parallel TLB placement with respect to the cache.

First, the simulation environment is described. A trace-driven approach was taken, and the rationalisation for this approach and the tools used to perform this are detailed. Then, the workloads used for simulation are described. As the emphasis in this evaluation is embedded systems, workloads that would be realistic on these systems are used. The simulation results provided statistics such as the number of cache/TLB hits and misses and page table accesses, which when applied to an appropriate model yielded the cycle and energy costs from the memory hierarchy. Address translation alternatives are evaluated against the *ideal case*, in which there are no address translation overheads.

The metrics used in the evaluation are described in the following section. *Cycles per instruction* (CPI) and *energy per instruction* (EPI) are the metrics by which the address translation overhead are described in the Results chapter. The cache miss ratio and PTE pollution ratio are also used to explain the differences in the above metrics between configurations in terms of application and system cache behaviour.

The parameters used in this study are based on those common to high-performance low-power embedded systems, which are the motivation here for the consideration of the energy consumption of address translation. The baseline configuration, and parameters varied for sensitivity analysis, are derived from example parameters from commercial processors and are described in the following section. The chapter concludes with a discussion of each of the parameters considered for the sensitivity analysis.

3.2 Simulation environment

A trace-driven approach [48] was taken in this study. This involved the generation of address traces for the workloads and then a simulation of the traces on different cache and TLB configurations.

....	
i	0x000080e8
i	0x000080ec
D	0xbfffc000
i	0x00080f0
d	0x0008108
....	

Figure 3.1: A fragment of an address trace. The address of the memory reference is indicated on the right and the type of memory access on the left, with 'i' representing an instruction fetch, 'D' a data write and 'd' a data read.

This approach simplifies the memory hierarchy simulation by separating the *instruction set architecture* (ISA) simulation from the memory hierarchy simulation, and allows for tools to be written to work independently to a particular ISA simulator. The use of traces was primarily motivated by the wish to modularise the simulation environment, and not for efficiency concerns.

3.2.1 Trace generation

A number of simulation tools were considered for the ISA simulation stage. The requirements for this tool were that it:

- be able to run standard workloads such as the SPEC and MediaBench benchmark suites;
- support the ISA of an embedded processor such as the ARM or MIPS architectures;
- provide an OS layer emulator rather than full system simulation. For reasons explained below, it was desirable to limit execution of kernel code to the TLB refill handler only;
- have modifiable source code freely available;
- be stable and widely used to reduce the likelihood of bugs and errors;
- support Linux as a host.

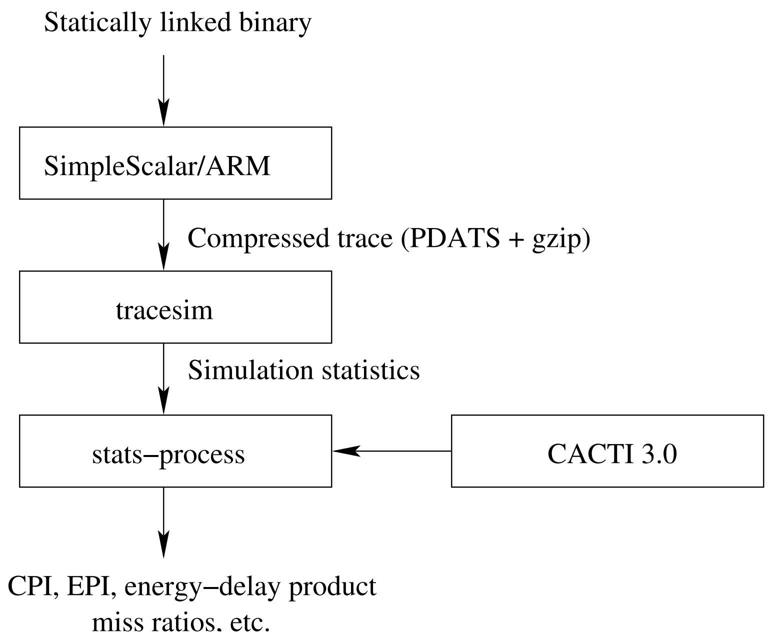

Figure 3.2: Simulation environment.

The Sulima [54], Simics [27], SimOS [35] and SimpleScalar [6] simulators were all considered. The only simulators that supported an OS emulation layer were SPARC Sulima and SimpleScalar. SPARC Sulima was still in a developmental state, and required hosting on a SPARC box, so SimpleScalar, which met all the other requirements, was selected for trace generation.

Traces were generated using the ARM ISA target of the SimpleScalar v4.0 test release.

SimpleScalar [6] "provides an infrastructure for simulation and architectural modelling". A number of micro-architecture models are available from a bare-bones instruction set interpreter to models with branch prediction, out-of-order execution and multilevel caches. These models provide simulation of user code, and system calls are trapped and handled by the host OS.

For the purposes of address trace generation, the *sim-safe* model, which does not model the details of the pipeline or memory hierarchy, was modified to generate traces in the desired format. Support for several system calls necessary to run the MediaBench and SPEC benchmarks was added — *utime, rename, rmdir, umask, gettimeofday, munmap, llseek*.

A limitation of the above approach was that the kernel impact on the caches and TLBs was ignored, with the exception of the TLB refill handler, in some the cases the most time consuming OS component [34], which was simulated. The actual kernel overhead would depend on the specific kernel used, and would introduce additional parameters, together with potential kernel modifications, resulting in greater complexity

Benchmark name	Number of instruction references to skip
go	1000M
m88ksim	N/A
gcc	200M
compress	2500M
li	2000M
ijpeg	1000M
perl	N/A
vortex	3000M

Table 3.1: Number of initial instruction references skipped for each of the SPEC benchmarks.

and slower simulation time. Additionally, multitasking was ignored as the assumption (made in many other cache studies) was made, namely that the separate benchmark results will be indicative of the real system performance. All other user-level code was simulated, including the standard libraries.

The MediaBench benchmarks were runnable to completion, however the SPEC benchmarks posed a problem, due to their long execution times. In order to make simulation tractable, only 250 million instruction references were simulated. While it would have been ideal to simulate a larger window, time constraints on the thesis results meant that this was not possible. The approach taken in [38] was used here, where simulation does not begin until a certain period of time has elapsed, so that uncharacteristic startup behaviour is skipped.

This startup behaviour can be seen in the graphs indicating the data and instruction cache miss ratios versus time for some of the SPEC benchmarks in Appendix A. Most noticeable is the startup behaviour of the compress and vortex benchmarks.

Based on the warmup instructions skipped in [38], and visual inspection of the graphs in Appendix A to correct for differences in instruction sets and code generation, the simulation starting points in Table 3.2.1 were found for the SPEC benchmarks used. These initial instruction references were skipped completely, with the simulated window beginning after this. The m88ksim and perl benchmarks were omitted, due to time constraints on thesis results.

System calls that necessitated a protection change were instrumented and a record was made on each invocation. This was used to provide an optimistic best case number of flushes required for the caches and TLB/PLBs. Real systems may involve many more protection changes, such as systems that use protection changes as part of their *inter-process communication* (IPC) system. Protection changes were considered to occur on the following system calls:

- ARM_SYS_exit
- ARM_SYS_old_mmap
- ARM_SYS_munmap

- ARM_SYS_brk

3.2.2 Trace compression

Each memory reference required 5 bytes to represent: 4 bytes for the address and an additional byte to annotate the reference with the type of memory access, e.g. instruction read, data write, etc. Since some of the benchmarks produced 1–2 billion memory references, the storage requirements for the address traces would have been unmanageable.

PDATS [20] lossless trace compression, together with Lempel-Ziv (*gzip*) was used on the address traces. PDATS takes advantage of the spatial locality in each reference, data or instruction, stream. PDATS stores address offsets and in the case of sequential access the number of repetitions. When Lempel-Ziv compression is applied to traces in this format one and a half orders of magnitude reduction in size were achieved by Johnson et al [20], providing an improvement over just Lempel-Ziv compression on the original address trace.

3.2.3 Trace simulation

The SimpleScalar cache simulation code was used to write a tool that modelled the memory hierarchy, consisting of caches, TLBs and page table walkers, for the configurations in the experiment. Multiple configurations were simulated on a single pass on the address trace, amortising the cost of trace generation, compression and decompression and reducing the I/O bandwidth requirements.

3.2.4 Post-simulation modelling

Using the statistics generated during trace simulation, such as the number of cache accesses, TLB misses, etc., the details of the configuration, and the cycle model and energy models described below metrics for performance and energy consumption were calculated. *Gnuplot* was used to graphically represent this data.

3.3 Benchmarks

As the aim of this study was to characterise the performance of the translation mechanisms, benchmarks that were computationally and memory system bound were used.

3.3.1 SPEC

The *system performance evaluation cooperative* (SPEC) benchmarks are computationally intensive workloads designed to evaluate the performance of a wide variety of hardware. The SPEC CPU95 integer benchmarks are described in Table 3.2.

The SPEC CPU95 benchmarks were selected over the more recent SPEC CPU2000 benchmarks primarily as a result of memory footprint size. SPEC CPU2000 has very large resident sizes, in the range of 100–200MB, for about half its benchmarks [15].

Benchmark name	Instructions executed	Description
go	250M	Game playing AI
compress	250M	Lempel-Ziv compression
ijpeg	250M	Lossy compression
li	250M	Lisp interpreter
gcc	250M	C compiler
vortex	250M	Object-orientated database
perl	N/A	Perl interpreter
m88ksim	N/A	88100 microprocessor simulator

Table 3.2: SPEC benchmark descriptions.

This is an unrealistic memory footprint for a workload on an embedded system. The SPEC CPU95 benchmarks were developed before such large memory sizes were common, making them more suited to present day memory sizes on embedded systems.

The SPEC CPU95 integer benchmarks were built as static binaries with the default optimisation levels for each of the applications.

3.3.2 MediaBench

The MediaBench [26] benchmarks consist of multimedia and communications applications aimed at evaluating embedded architectures and compilers. The applications in MediaBench 1.0 are described in Table 3.3.

A higher instruction cache hit rate was found in [26] for MediaBench when compared with the SPECint 1995 benchmarks. Data read hit rates were higher but write hit rates lower than SPECint, with a high variance. MediaBench is widely used in research papers for the evaluation of memory systems on embedded systems.

The MediaBench benchmarks were built as static binaries with the default optimisation levels for each of the applications.

3.4 Metrics

3.4.1 Energy per instruction

$$EPI = \frac{\text{execution energy}}{\text{user-level instructions}}$$

For each access to the TLBs or caches, the total power for all banks, as reported by CACTI 3.0, was accumulated, with the addition of the energy consumption of memory accesses providing the execution energy.

For memory accesses, a line fetch or write-back, $E_{line\ fetch}$, will be considered to be 128 times the energy cost of a hit on the cache. The choice of this value was based

Benchmark name	Instructions executed	Description
JPEG compression	4M	Lossy image compression
JPEG decompression	15M	Lossy image decompression
MPEG2 encoding	1017M	Video compression
MPEG2 decoding	160M	Video decompression
GSM coding	296M	Speech transcoding
GSM decoding	103M	Speech transcoding
G.721 coding	388M	Voice compression
G.721 decoding	372M	Voice decompression
PGP encryption	103M	Public key encryption
PGP decryption	82M	Public key decryption
PEGWIT encryption	43M	Public key encryption
PEGWIT decryption	24M	Public key decryption
Ghostscript	1014M	PostScript interpreter
Mesa texgen	122M	Texture mapping
Mesa mipmap	78M	Fast texture mapping
Mesa osdemo	32M	Standard renderer pipeline
RASTA	30M	Speech recognition
EPIC encoding	61M	Image compression
EPIC decoding	10M	Image decompression
ADPCM encoding	8M	Adaptive differential pulse code modulation
ADPCM decoding	7M	Adaptive differential pulse code demodulation

Table 3.3: MediaBench benchmark descriptions [26].

on its used in a previous study [55], and an examination of a datasheet for a low-power SDRAM [1]. This was varied in the sensitivity analysis to allow for consideration of the results if this choice was not correct.

The EPI that is measured in this simulation is the EPI overhead of the memory system. This does not take into account the EPI of other components of a processor, but the EPI overhead of just the caches may account for over half the the total EPI of a processor in some embedded processors [29], and if an estimate is made which weights the power consumption of the SDRAM typically used as equal to that of the processor, over 75% of the embedded system's EPI is being accounted for in this EPI overhead measured.

The number of user-level instructions was reported directly by SimpleScalar.

3.4.2 Cycles per instruction

The average number of cycles executed for each instruction may be found by the following expression:

$$CPI = \frac{\text{execution cycles}}{\text{user-level instructions}}$$

The number of execution cycles was found as follows. Each instruction cache access was be counted as one cycle. The pipeline implementation was not be modelled, instead the assumption was made that hits on the data cache are hidden by the pipeline and that a miss to either the instruction or data cache caused the pipeline to stall while the required item was fetched from memory. Initially only hardware walked page tables were considered, and later, when the overheads of a software walked page table were considered, additional cycles were added to the handler cost for the exception overhead. On a cache miss, the number of cycles required depended on whether write-back was required, and whether the address translation was successful.

A 20 cycle penalty was assumed for a line to be fetched from main memory, also based on examination of a low-power SDRAM datasheet [1]. This was varied in the sensitivity analysis.

When the TLB was placed after the virtual cache there were two options for the address translation on write-back. Either the physical address for the line might be stored in the line when it was bought into the cache, or a TLB access might be made. To avoid the complexity of the deadlock conditions discussed by Wood [51], the physical address is considered to reside in the line for this study.

A TLB miss, on a cache access when placed in front of the cache or on a cache miss when placed behind the cache, required additional cycles directly, through the cost of the page table walk, and indirectly, through the later miss costs due to displaced lines in the cache. The effects of both of these were modelled by the aggregate cache miss number provided by *tracesim* and the assumption that the walking of the page tables under consideration, namely virtual linear page tables, was achieved in the number of cycles required to access the page table entry. However, since only the aggregate cache hit and miss numbers were provided by *tracesim*, and on a hit to the data cache no cycle was counted, it was necessary to calculate the number of cache accesses during the handling of a miss and add this to the total cycle count.

The CPI measured is only the CPI overhead of the memory system. To place this in context, a simple scalar pipeline architecture might have a CPI of one. So, a CPI overhead of one, for example, would mean the speed of the system had halved.

The number of user-level instructions was reported directly by SimpleScalar.

3.4.3 Cache miss ratio

The cache miss ratio is defined as:

$$m_{cache} = \frac{\text{number of cache misses}}{\text{number of processor references}}$$

For the purpose of this study, as in Wood [51], the denominator only includes references from the user application but the numerator includes all cache misses, so that the miss ratios may be compared across different translation options, where a different number of system cache accesses take place, directly.

3.4.4 PTE pollution ratio

The PTE pollution ratio is defined as:

$$\text{PTE} = \frac{\text{number of PTE lines in cache}}{\text{number of cache lines}}$$

The mean and standard deviation are found during simulation to provide some measure of cache pollution. This is only measured for the data cache, as PTEs are not cached in the instruction cache.

3.5 Baseline configuration

A similar approach to that taken by Wood in [51] was utilised below, in which a baseline configuration is first simulated with the translation parameters varied, and then a sensitivity analysis is conducted in which individual parameters are varied in the baseline configuration to provide a consideration of the generality of the results.

Processor	Clock speed (MHz)	Feature size (microns)	Cache	TLB
Intel XScale PXA250	200–400	0.18	Split virtually-addressed 32KB/32KB (I/D), 32-way SA, 32 byte line size, RR replacement	Split TLB, 32-entry FA, RR replacement
SuperHitachi SH-4 SH7550R	200–240	—	Split 16KB/32KB (I/D), 2-way SA, 32 byte line size, LRU	64-entry FA unified TLB and smaller 4-entry FA TLB for instructions loaded from unified TLB
IBM PowerPC 405GPr	266–400	0.18	Split 16KB/16KB (I/D), 2-way SA, 32 byte line size, LRU	Hierarchical TLB, with a 64-entry FA L2 unified TLB and split L1 4/8-entry (I/D) FA TLBs
Alchemy Au1500	333–500	—	Split 16KB/16KB (I/D), 4-way SA, 32 byte line size, LRU	32-entry FA unified TLB and smaller 4-entry FA TLB for instructions loaded from unified TLB

Table 3.4: Some embedded processors.

A baseline configuration, based on parameters from example embedded processors briefly summarised in Table 3.4, is described below:

- Single level split instruction and data caches. For the baseline configuration both caches are virtually addressed 16 KB, 2-way set-associative, write-back, write-allocate, with a line size of 32 bytes and LRU replacement policy.
- Single level split instruction and data TLBs. Both caches are fully-associative. Each PTE is 4 bytes in size and constitutes one TLB entry.
- A virtual linear page table with 4 KB page sizes is used. PTEs for instruction and data regions are cached in the data cache only, in order to simplify consistency of PTEs on a page table modification.
- The write-back address is assumed to be stored with the cache line after the line is fetched.
- The following time/energy costs are associated with main memory:
  - The time required to fetch a line, $t_{line\ fetch}$, is 20 cycles.
  - The energy required to fetch a line, $E_{line\ fetch}$, is 128 times the cost of a cache hit.
  - The caches had 32 address bits and 32 data output bits with conventional TLB configurations.
  - The TLB has 20 address bits and 16 output bits. This is based on VA space of 32 bits, PA space of 28 bits and 4KB page size.

The assumptions were made that there were no synonyms and that physical memory was large enough so that no swapping was required.

The baseline configuration described above was simulated with the following independent variables:

- Address translation placement — address translation may either take place before or after the cache. When placed before the cache, a TLB is comparable to existing systems, supplying protection and the physical address. Since the goal of the first experiment is to evaluate address translation only, a TLB may appear unnecessary, providing only a slight advantage on write-back but incurring energy overheads on each instruction and data access. However, this configuration allows for comparison with caches that may be physically addressed and existing systems such as the Intel XScale processors.

  In addition, a configuration in which no address translation costs are incurred is simulated to provide an ideal case for comparison.
- TLB size — the TLB size will be varied from 0 to 64 entries. In-cache address translation [51] occurs when a zero entry TLB is utilised after the virtual cache.

The metrics described in section 3.4 will be used to provide the dependent variables for this experiment.

3.6 Sensitivity analysis

The following parameters were adjusted in the sensitivity analysis to provide a consideration of the validity of the previous results when generalised past the specific set of parameters in the baseline configuration:

- Cache size
- Cache associativity
- Cache line size
- Page size
- Handler cycles — rather than the previous assumption of a hardware walker, which only had a 1 cycle penalty on top of the line fetch costs, the handler cycles that might be required for a software page table walker are considered here. Components of this might be the cost of the exception, pipeline flush, refill handler execution and the return from exception.
- Energy consumption and time of a line fetch and write-back to main memory — here the previous assumption of 128 times the cache hit energy and 20 cycle overhead, for energy and time costs of the line fetch respectively, is challenged.
- Multiple address spaces — an ASID is added to the TLB when it is on the cache hit path, or the cache when this is not the case.
- Protection overheads — In the above baseline configuration it was assumed that protection was implemented in the TLB when before the cache and in-cache, through the storage of access rights in the cache line, when placed after the cache. Jacob and Mudge found in [18] that the latter arrangement could give an overhead of 2-7% on workloads executing on top of the Mach kernel. We consider here iterating through all the lines in the cache, on the compulsory protection changes, with a 5 cycle per line cost to update the protection bits on the line, to provide an optimistic estimate for the overheads of protection changes. It should be noted, however, that this ignores the significant impact of the operating system behaviour here.
- Hierarchical TLB — by only consulting the TLB when a page boundary is crossed, an appreciation of the gains possible through the use of hierarchical TLBs is provided, reducing the TLB overhead on the cache hit path potentially.

3.7 Summary

A description of the simulation environment has been provided in this chapter, and the benchmarks that were utilised have been described. The metrics and energy and time models that provide the quantities by which the address translation mechanisms may be compared were detailed following this. Finally, in this chapter, the baseline configuration and sensitivity analysis on which the selected benchmarks were run were presented.

Chapter 4

Results

4.1 Overview

This chapter presents the results from the execution of the benchmarks on the baseline configuration described in the previous chapter. EPI and CPI overheads, and relevant measures of cache behaviour are used to understand the behaviour of the address translation mechanisms. A discussion is provided of the results, followed by some conclusions about the performance of the address translation mechanisms on the baseline configurations.

Following this, a sensitivity analysis is conducted on each of the parameters listed in the previous chapter. A number of observations about the data are provided, and a summary of the significance of the parameter to the results follows this, for each parameter.

Finally, the above conclusions are summarised for discussion in the concluding chapter.

4.2 Baseline configuration

The EPI overhead for the baseline configurations is shown in Figures 4.1, 4.2, and 4.3, with a breakdown into the contributions from the different memory components. On the horizontal axis, as with all other graphs in this chapter, for each benchmark there are 7 bars. The leftmost indicates the ideal case in which there are no translation overheads. The next indicates the case that there is no TLB and so in-cache address translation is used. The next 4 bars indicate different TLB configurations in which the TLB, with size ranging from 8 to 64 entries, is placed on the cache hit path. The final bar indicates a 64-entry off-chip TLB configuration.

Due to the large number of benchmarks that were simulated, each logical graph is split into three physical graphs. The first two physical graphs, in the case of EPI for the baseline configurations Figures 4.1 and 4.2, present the results for the MediaBench workloads, arbitrarily split into two groups. The third physical graph, in the case of

EPI for the baseline configuration Figure 4.3, presents the results for the SPEC CPU95 integer benchmarks.

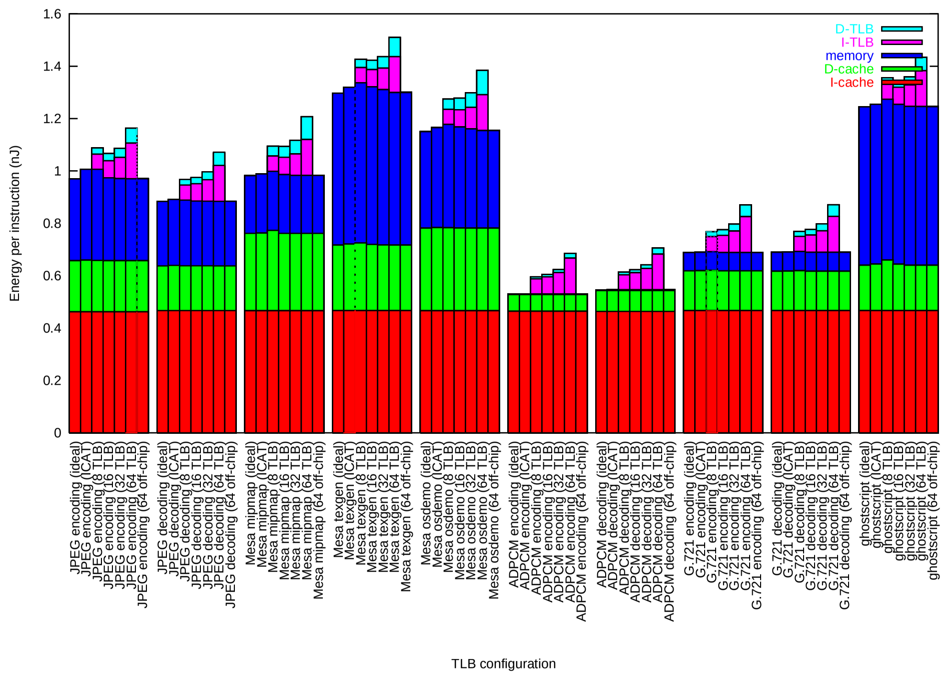

Figure 4.1: Energy per instruction overhead for baseline configurations on MediaBench group 1.

The EPI overhead is for the memory system, accounting for cache, TLB and memory energy consumption. Each bar is broken down into the following components: instruction cache hit energy, data cache hit energy, memory line fetch energy, instruction TLB hit energy and data TLB hit energy.

The EPI contribution from the instruction cache is fixed across the configurations, as the PTEs are cached in the data cache, and a hardware page table walker is used, resulting in no difference in the number of instruction cache accesses between different address translation configurations. The data cache access energy show some variability as a result of the difference in the number of page table lookups between configurations, however this is only slight. The MESA mipmap, ghostscript, RASTA, compress, go and vortex benchmarks are the most visible examples of this. Figures 4.4, 4.5, and 4.6, discussed below, confirm this explanation as these are the benchmarks with the greatest contribution to the CPI from the walking of the VLPT.

The difference in the contribution to the EPI from the memory accesses can be understood as a result of PTE pollution of the data cache. The benchmarks which show this most clearly are JPEG encoding, RASTA, compress, go, gcc and vortex. The RASTA benchmark exhibits a very large increase in contribution due to memory accesses from the 8-entry TLB configuration. The data cache miss ratio climbs sharply

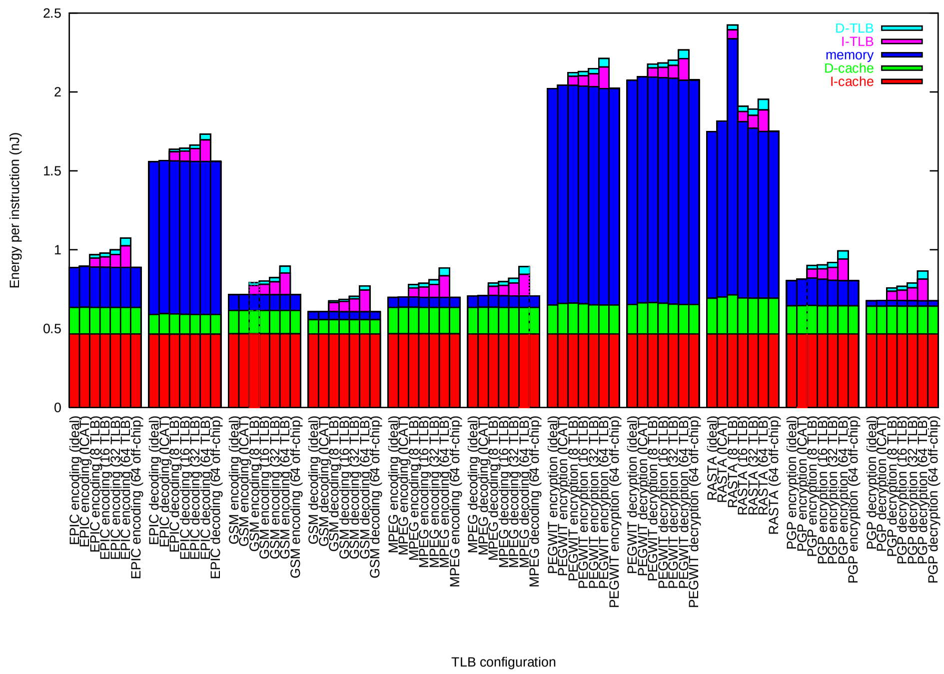

Figure 4.2: Energy per instruction overhead for baseline configurations on MediaBench group 2.

here, as indicated in Figure 4.8, and it is hypothesised that this is a result of a degenerate conflict in a set between PTEs and application data causing a large number of conflict misses. Later in this chapter, the sensitivity analysis on cache associativity shows a reduction in this anomaly with higher associativity caches. Some benchmarks, for example ADPCM encoding, ADPCM decoding and PGP decryption, have a close to zero contribution from memory access energy to their EPI overhead, due to the fact that the data and instructions caches are large enough to accommodate both the application working set and PTEs. The go benchmark provides an example showing, where cache pollution from PTEs results in a significantly higher number of memory accesses, that larger TLBs can save energy by reducing PTE cache pollution.

Cache pollution of PTEs was found to be reduced by larger TLB sizes in most benchmarks. The mean PTE pollution ratio is shown for the baseline configurations in Figures 4.10, 4.11 and 4.12. Error bars indicate one standard deviation in either direction. Vortex has the largest mean PTE pollution ratio, greater than approximately 10% for all configurations, about double that of any other benchmark.

It is the benchmarks with very low cache miss ratios, and hence negligible contribution from memory access costs, or page table lookups in the case of the no-TLB configuration, that indicate the most optimistic gains that can be expected from the removal of a TLB from the cache hit path. The ICAT EPI overhead is indistinguishable from the ideal EPI overhead, and gives a reduction in EPI overhead over the TLB con-

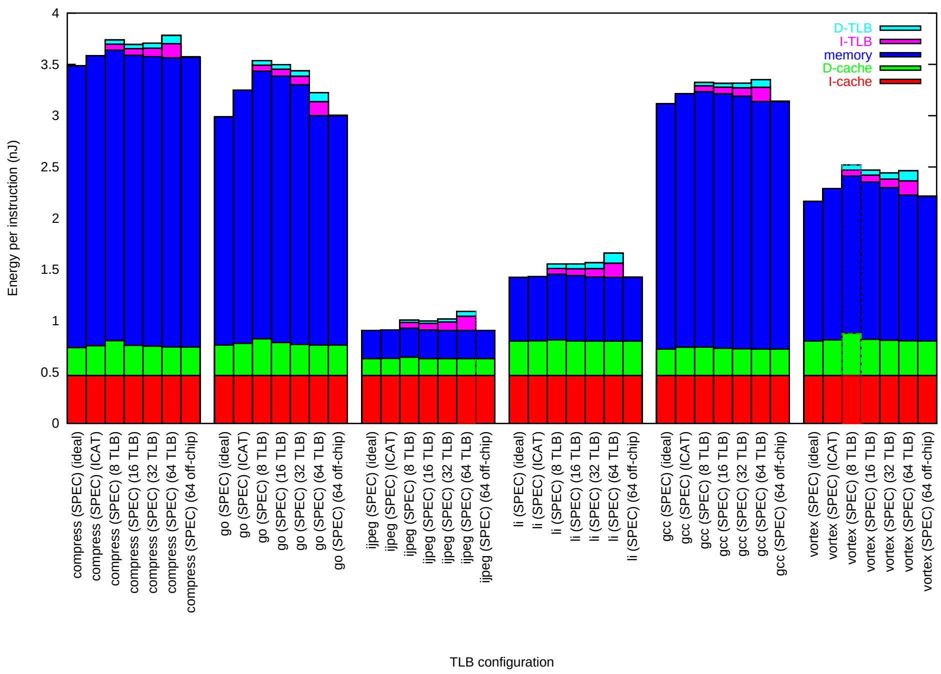

Figure 4.3: Energy per instruction overhead for baseline configurations on SPEC benchmarks.

figurations of at least 11%, in the case of ADPCM encoding. The worst case behaviour from ICAT is evident in go with approximately 8% of the ideal case additional EPI. For almost all other workloads, however, a much lower overhead is present with ICAT, close to the ideal in many cases and better than the configurations with a TLB on the cache hit path in all cases. An off-chip TLB may reduce the address translation overhead further, in some cases. In the case of go, this reduces the EPI overhead to close to ideal. The hit energies of the TLB are negligible when off the cache hit path, as indicate in the EPI breakdown. The reduction in overhead comes from the reduction in cache pollution. On the compress benchmark there appeared to be no reduction in overhead. This can be understood by considering the mean PTE pollution for this benchmark and observing no further decrease in PTE pollution. A large page working set may explain this, and improvements would come from improving TLB coverage off-chip as well as for the other TLB configurations.

Figures 4.4, 4.5 and 4.6 demonstrate the CPI overhead of the workloads on the different address translation mechanisms. The four components of interest are the memory access cycles resulting from instruction and data cache misses, the VLPT lookup costs, and the TLB component which represents the additional cycle overhead for TLB lookup when the TLB is not on the cache hit path.

The same variations in memory costs as seen in the EPI graphs can be seen in the CPI graphs.

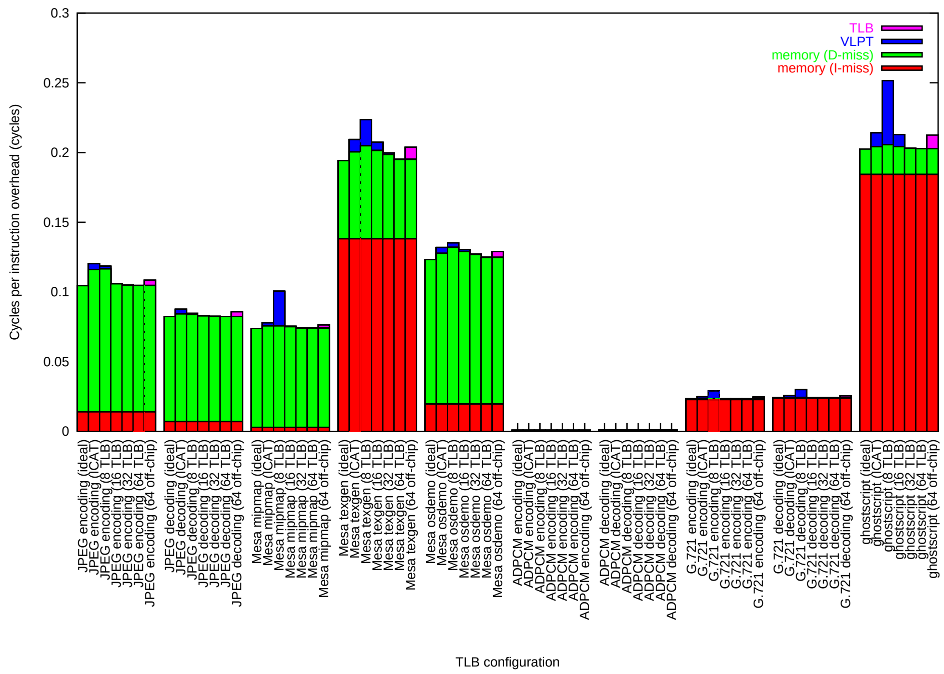

Figure 4.4: CPI overhead for baseline configurations on MediaBench group 1.

Only a handful of benchmarks have significant VLPT walking costs: MESA mipmap, MESA texgen, ghostscript, PEGWIT, RASTA, and the SPEC benchmarks. The VLPT CPI costs from the ICAT configuration vary in their relation to the TLB configuration VLPT CPI costs between benchmarks. These are, expectedly high, relative to the larger TLB configurations. There are two events that determine this. If a TLB miss occurs but no cache miss, for the TLB configurations, then ICAT will have a lower VLPT cost. However, if a cache miss occurs but no TLB miss, then ICAT will have a high VLPT cost. From these benchmarks, ICAT shows evidence of the latter case when compared with the larger TLB configurations. That is, cache thrashing sets in prior to TLB thrashing in most benchmarks on larger TLBs, resulting in a higher VLPT CPI for ICAT than the larger TLB configurations with most benchmarks. ICAT does, however, perform comparably with the smaller 8-entry configurations and far better in the case of ghostscript, MESA mipmap, MESA texgen, RASTA and the SPEC benchmarks (with the exception of gcc). In the most optimistic case, ICAT performs close to ideal, such as with GSM encoding and decoding. However, go presents the worst ICAT overhead of approximately 17% of the ideal due to the PTE pollution observed above. Again, an off-chip TLB may reduce this, to 5% of the ideal. Since the TLB lookup cost is not hidden in an off-chip TLB by the cache lookup, as is the case for the other TLB configurations, there is a noticeable CPI penalty for this. For this reason the off-chip TLB performs, in all benchmarks where there is deviation from the ideal, no better than the larger TLB configurations on the cache hit path. The off-chip TLB also performed

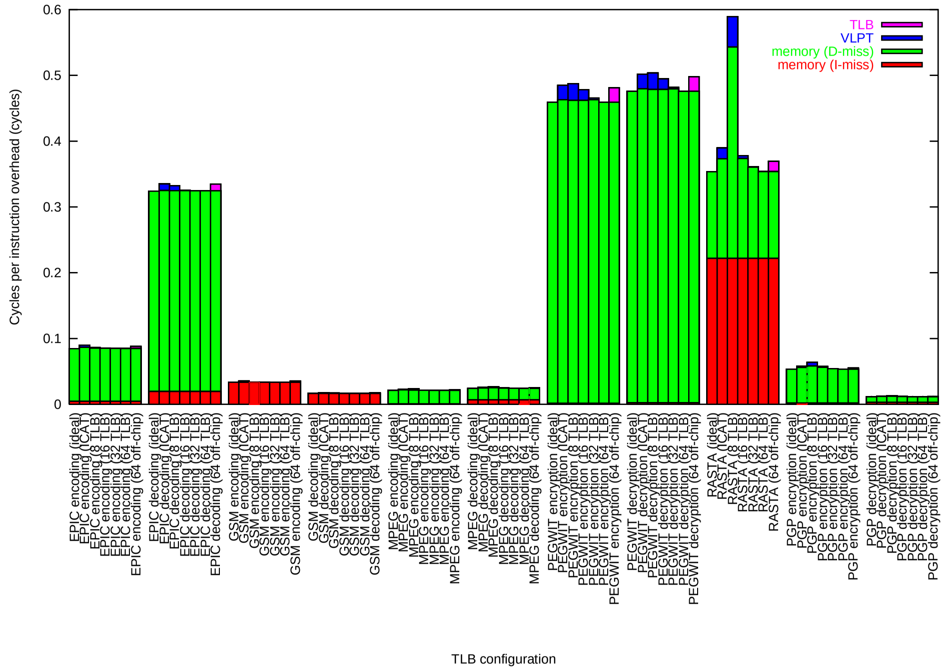

Figure 4.5: CPI overhead for baseline configurations on MediaBench group 2.

closely to ICAT for most of the other benchmarks. In one case the off-chip TLB performs worse than ICAT, on the compress benchmark. This has already been suggested to have poor page locality above. ICAT has a CPI overhead noticeably slightly worse than any other configuration in two benchmarks: JPEG encoding and EPIC decoding.

The miss ratio graphs for the benchmarks on the different address translation mechanisms are shown in Figures 4.7, 4.8 and 4.9, and use a combined instruction and data cache miss ratio, so that the contribution to memory accesses from each of the caches may be placed in context. As a result, the actual miss ratios are lower than those usually reported in the literature, however only the relative sizes of these miss ratios is of importance.

It is misleading to consider the above results as a true comparison between address translation options on virtually-addressed caches. Since memory protection is not considered until Section 4.3.8, a comparison between the TLB configurations on the cache hit path that do provide protection and ICAT is not valid. However, as Section 4.3.8 indicates, the results are valid in the optimistic case considered. The results in the baseline configurations also indicate the best case energy improvement that might hope to be gained by the utilisation of a virtually-addressed cache in the first place.

The above discussion and graphs may be summarised with the following observations:

- Smaller 8-entry TLBs on the cache hit path generally perform the worst, with

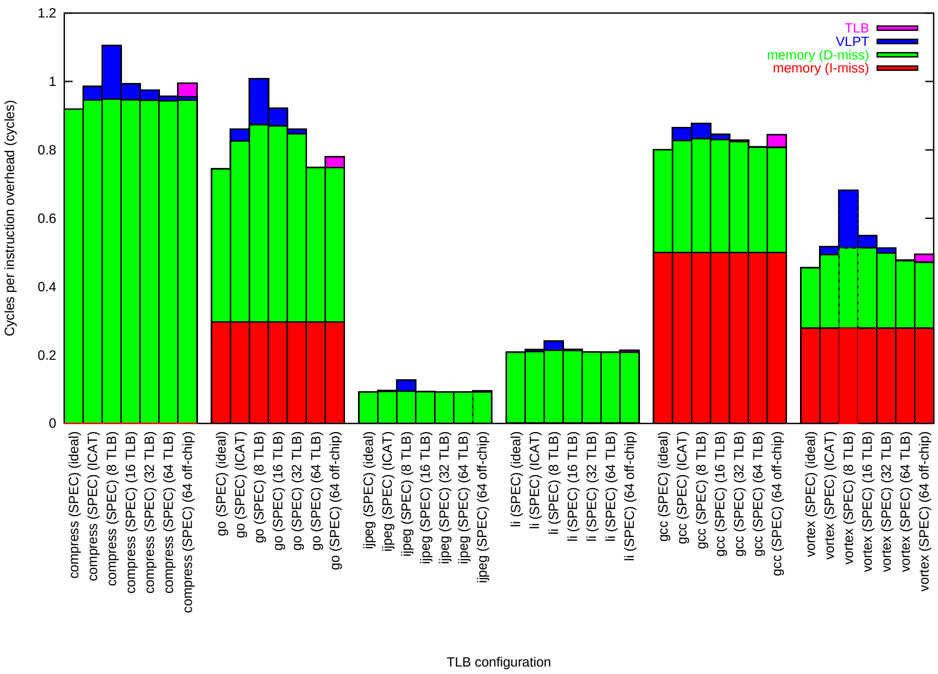

Figure 4.6: CPI overhead for baseline configurations on SPEC benchmarks.

little improvement and even worse energy efficiency than larger TLBs on some benchmarks, and significantly worse CPI overheads on many benchmarks.

- By either eliminating the TLB or removing it from the cache hit path, memory system energy consumption may be reduced by up to 11% on the baseline configuration. At the same time, in-cache address translations tends to exhibit worse cycle time than the larger, less energy efficient, TLBs, with additional cycle overheads of up to 17% of the ideal. The selection of address translation mechanism is therefore, in part, a tradeoff between energy and cycle overheads. This will depend on the relative significance of these overheads in the overall processor architecture.

- If energy and cycle overheads were of equal importance, the 16 and 32-entry TLB on the cache hit path appear the better choice for a TLB on the cache hit path.

- Cache pollution of PTEs is significant on some benchmarks, such as go and vortex. This is similar to the finding by Jacob and Mudge [19] in which the indirect costs of virtual memory were found to approximately double the overheads.

- Larger TLBs play a role in the reduction of PTE cache pollution, which might even outweigh their TLB hit energy consumption in some cases, such as the go

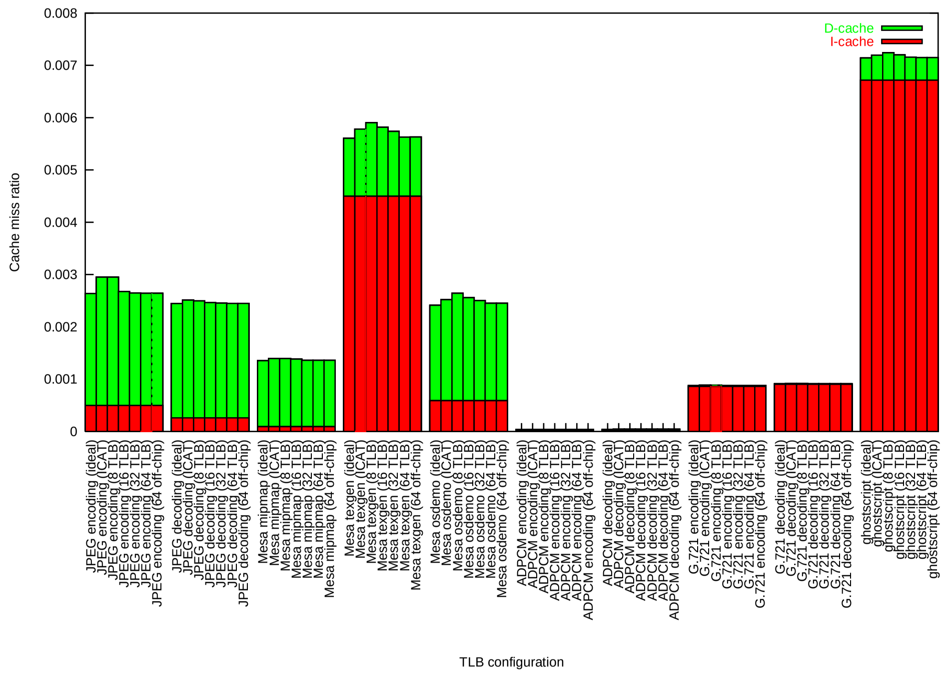

Figure 4.7: Combined cache miss ratio for baseline configurations on MediaBench group 1.

benchmark. On most benchmarks, however, smaller TLBs tended to be the more energy efficient.

- Off-chip TLB configurations have no significant energy overhead in most cases, and are comparable to the cycle overhead of ICAT on most benchmarks. In some exceptional cases, off-chip TLBs may provide lower cycle or energy overhead than ICAT. Off-chip TLBs need not be any larger than conventional TLBs in order to gain these benefits or produce comparable behaviour to ICAT in the common case, on the workloads considered.

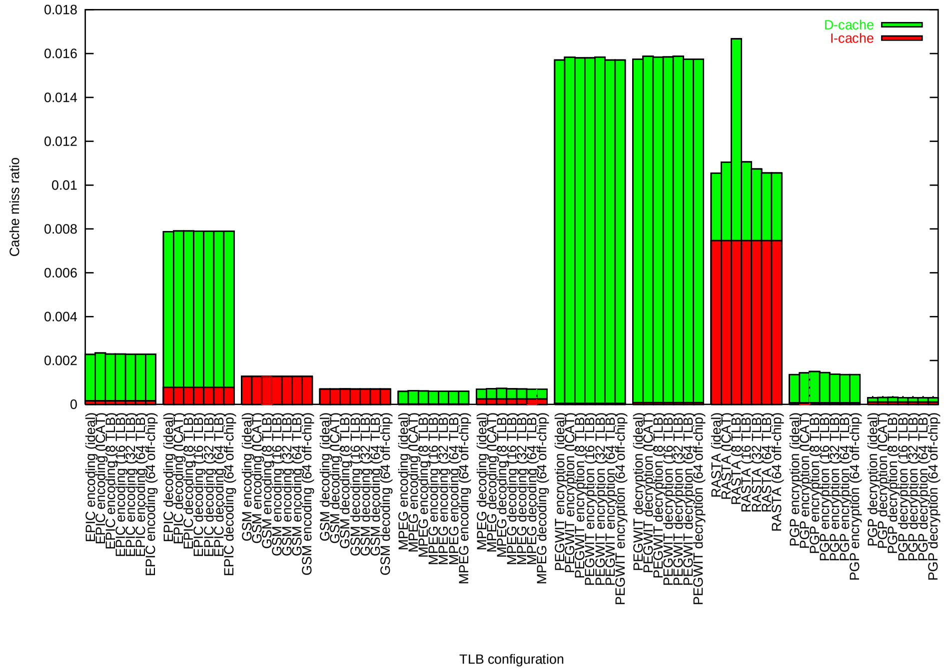

Figure 4.8: Combined cache miss ratio for baseline configurations on MediaBench group 2.

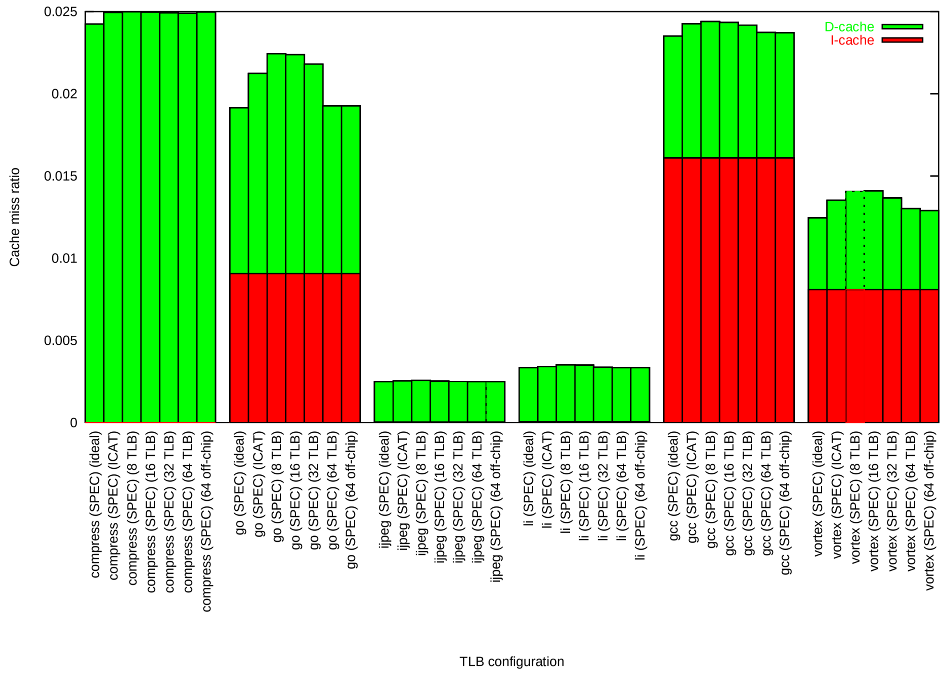

Figure 4.9: Combined cache miss ratio for baseline configurations on SPEC benchmarks.

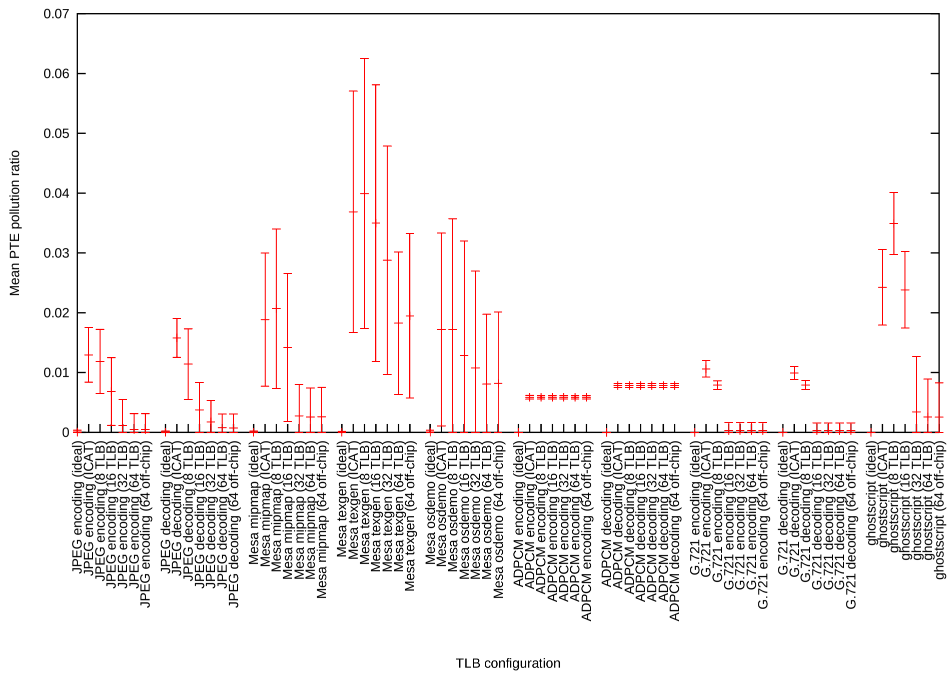

Figure 4.10: Mean PTE pollution ratio for baseline configurations on MediaBench group 1.

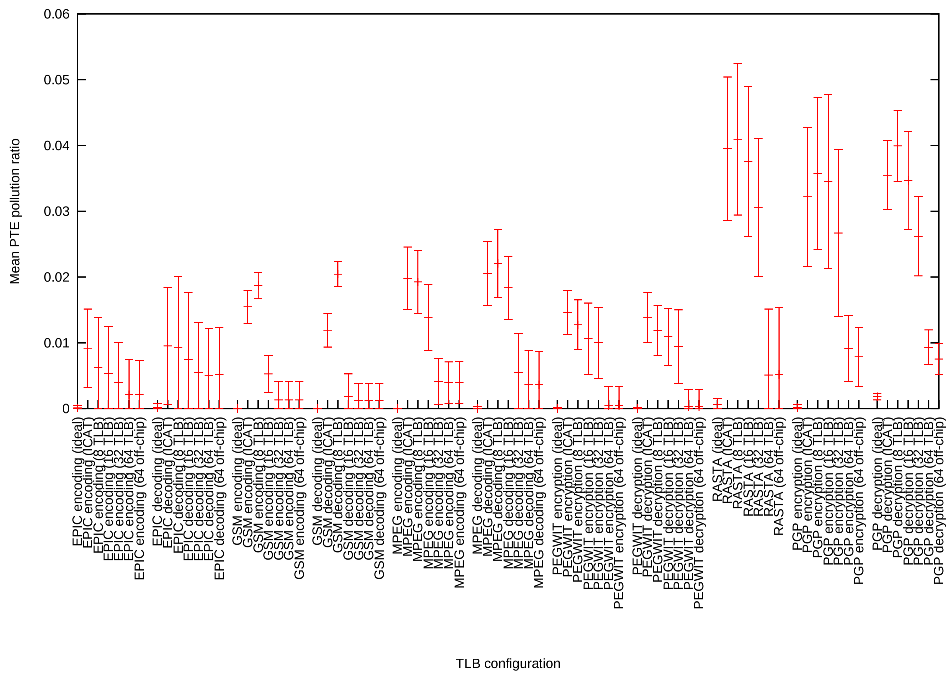

Figure 4.11: Mean PTE pollution ratio for baseline configurations on MediaBench group 2.

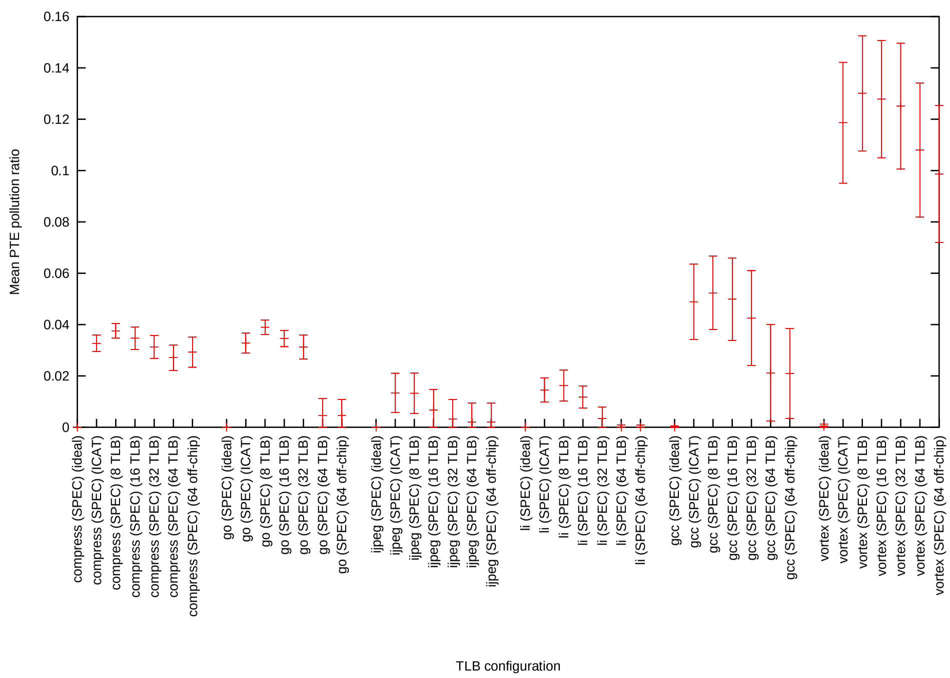

Figure 4.12: Mean PTE pollution ratio for baseline configurations on SPEC benchmarks.

4.3 Sensitivity analysis

An attempt was made to summarise the data below by taking the arithmetic mean between benchmarks. However, the standard deviations were large and did not allow for clear generalisation. The complete experimental data is instead presented, and the trends observed and behaviour of interest made clear through discussion.

Each graph has identical configurations and benchmarks on the horizontal axis as presented in the previous section. The dependent variables considered in this sensitivity analysis are the total EPI and CPI memory system overheads. Each parameter that is considered is presented in a separate set of graphs, with the values through which the parameter is varied indicated in the legend. Each point on the graphs indicates one execution of the benchmark at configuration identified on the horizontal axis and with sensitivity parameter value that was used indicated by the corresponding value in the legend.

Each of the parameters is considered in turn. For the first seven parameters, a series of observations are made about the trends observed and exceptions to these of note. This is presented in point form, and a concluding paragraph then discusses the relative importance of the parameter. The last three parameters are presented differently, as rather than varying the baseline configuration through a number of values, additional overheads that were not accounted for in the original baseline configuration are considered.

4.3.1 Cache size

- Cache size was generally more important in determining the optimum EPI overhead than the address translation mechanism used. When cache size was varied, the address translation overheads, relative to the ideal, did not change in most cases. ICAT and off-chip TLB configurations remain the most energy efficient, and close to the ideal in most cases, with the exception of the case in which the small 8KB cache sizes are the least energy efficient, discussed below.

  For some of the benchmarks, there was some overlap between the range of EPIs between different TLB configurations and different cache sizes. For example, the use of no TLB or an off-chip TLB with a 32KB caches makes the system more energy efficient for JPEG decoding when compared with the 16KB cache with a 64-entry TLB. However, this was only a very slight difference, and only significant in the anomalous RASTA and ghostscript benchmarks.

- Where smaller 8KB cache sizes were the least energy efficient, increasing the size of the TLBs on the cache hit path tended to improve energy efficiency, most likely by the reduction of PTE cache pollution. This is most noticeable in the go, gcc, and vortex SPEC benchmarks, where the energy costs are dominated by memory accesses resulting from capacity misses.

- ghostscript exhibits a large increase in EPI relative to the ideal for the 8-entry TLB configuration at 8KB cache size, even though 8KB is not the least energy

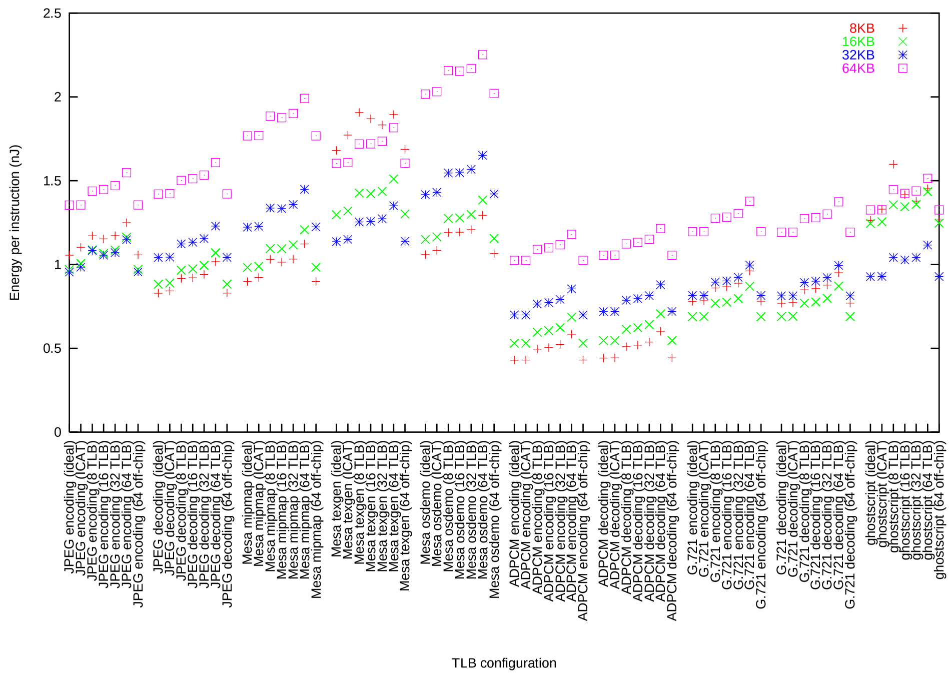

Figure 4.13: EPI overhead vs. cache size for MediaBench group 1.

efficient. Either significant cache PTE pollution or a degenerate set conflict between PTEs and data may be responsible. The anomalous behaviour of RASTA decreased with larger cache size significantly, probably through the set in conflict being split.

- While not directly applicable to the comparison of address translation mechanisms, the cache size that gave the best energy efficiency has been summarised in Table 4.1 below.

Cache size	**Number of benchmarks**
8KB	15
16KB	6
32KB	5
64KB	1

Table 4.1: Cache size with greatest energy efficiency

Unsurprisingly, cache size is a significant influence on EPI overhead, and for these workloads the optimum size is 8–16KB.

The inclusion of multi-process workloads and kernel impact on the caches may shift the cache size with greatest energy efficiency.

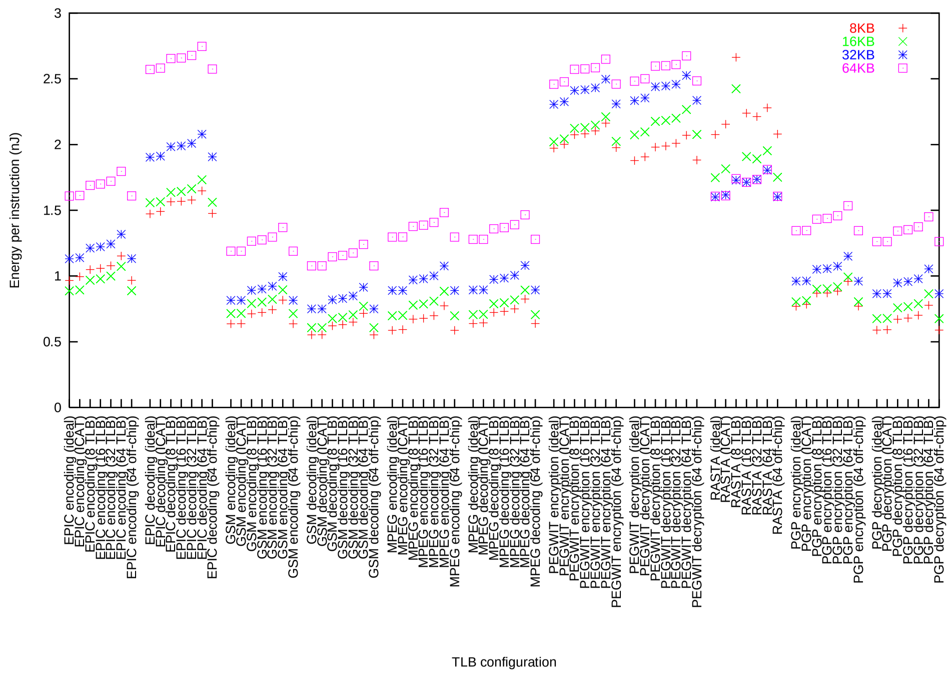

Figure 4.14: EPI overhead vs. cache size for MediaBench group 2.

- The cache size behaviour of the SPEC benchmarks differs significantly to those from MediaBench. The go, gcc and vortex benchmarks all favour larger cache sizes, with gcc being the only benchmark with a 64KB cache size as the most energy efficient. This is likely to be due to the larger working set size of the benchmarks. The go and vortex benchmarks provide the only clear examples of where no-TLB provides worse energy efficiency than TLB configurations, when the least energy efficient 8KB caches are used. The off-chip TLB in these cases, however, remains close to ideal, again suggesting that PTE cache pollution is the likely explanation for this.

- For those benchmarks that show variability in CPI with cache size, performance again tends to be dominated by cache size rather than address translation mechanism. On the SPEC benchmarks, there is some overlap, with an increased TLB size on the go benchmark, for example, improving performance of the 16KB cache past that of the 32 KB cache with small TLB size.

- Smaller 8KB cache sizes accentuate the difference between the configurations for some benchmarks, such as JPEG encoding, Mesa texgen, ghostscript, and RASTA, while not significantly affecting others, for example Mesa osdemo and EPIC decoding. This tends to occur where there is a large spread of CPIs between different cache sizes, suggesting that these benchmarks have larger working sets,

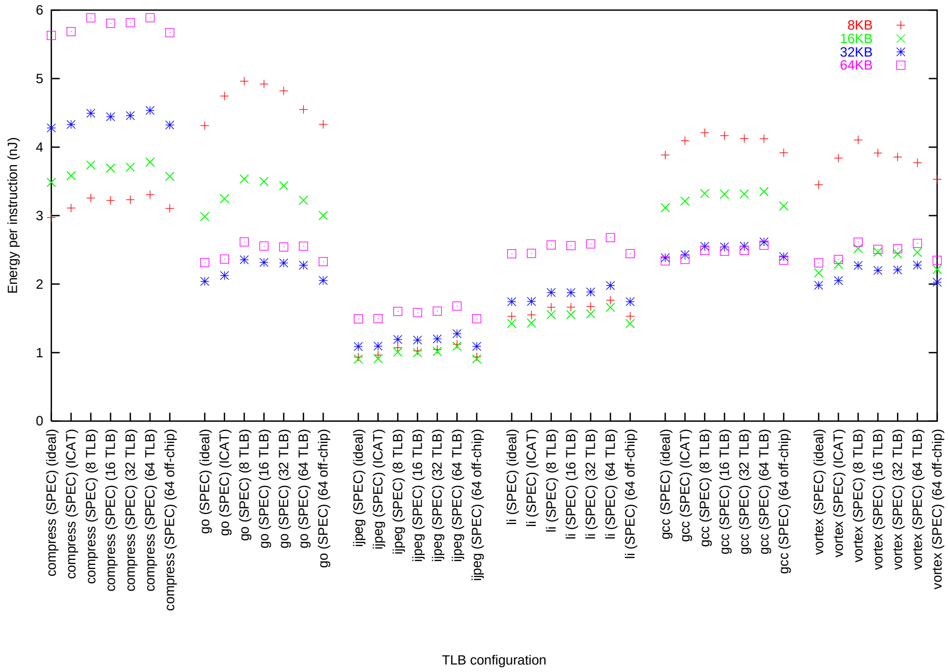

Figure 4.15: EPI overhead vs. cache size for SPEC benchmarks.

and hence PTE interference with the cache is the cause of the increased variance (TLBs on the cache hit path should not experience any change to TLB miss rates across cache sizes).

- For larger cache sizes (32KB and 64KB), ICAT or an off-chip TLB have close to ideal CPI overhead across benchmarks. While performing worse than medium and larger sized TLBs on the cache hit path at low cache sizes, the performance of the no-TLB configurations improve on these same TLBs at larger cache sizes, as expected, where the cache miss ratios fall. Mesa texgen a is clear demonstration of this, where ICAT is worse than all TLB configurations greater than or equal to 16-entry at 8KB cache size, but comparable with or better than these at 32KB. An improvement of ICAT performance with cache size was also observed by Wood [51].

- Again observed is the considerable greater variability in the SPEC benchmarks when compared to those in MediaBench with respect to CPI overhead.

Cache size can therefore be summarised as being a parameter that, while being of importance in overall memory system energy efficiency, is not of great significance to the relative EPI overhead of the address translation mechanisms. The results found for the baseline configuration appear to hold when the most energy-efficient cache is

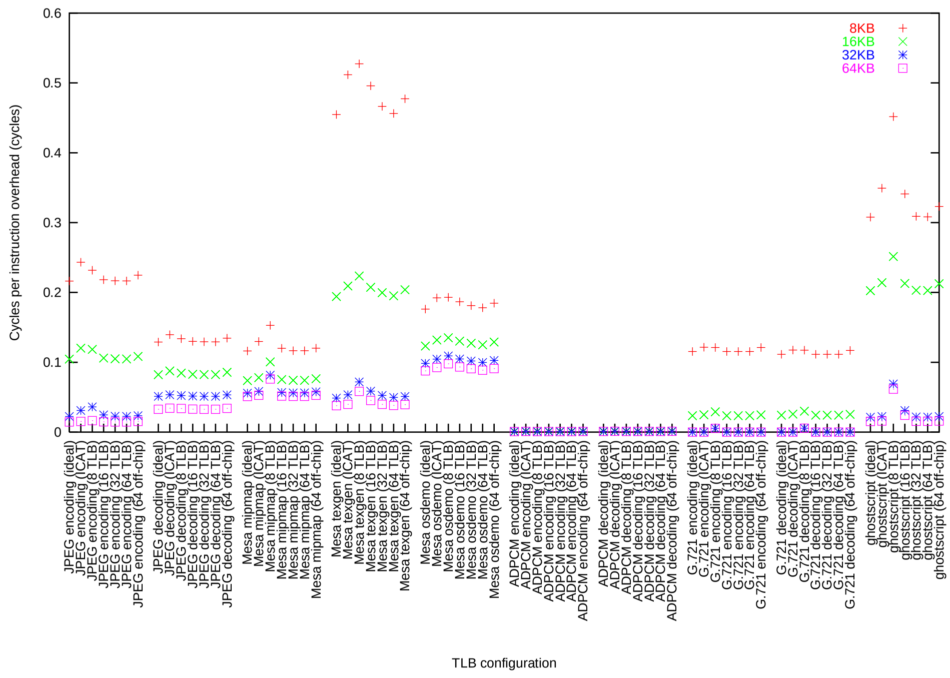

Figure 4.16: CPI overhead vs. cache size for MediaBench group 1.

used. Larger cache sizes improve the relative CPI performance of the systems with translation off the cache hit path.

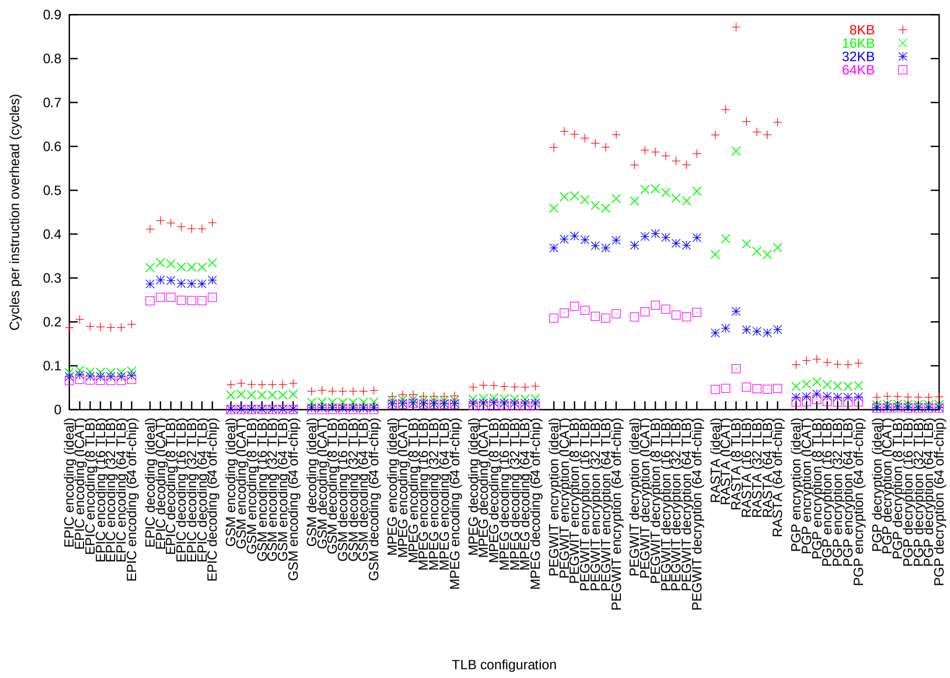

Figure 4.17: CPI overhead vs. cache size for MediaBench group 2.

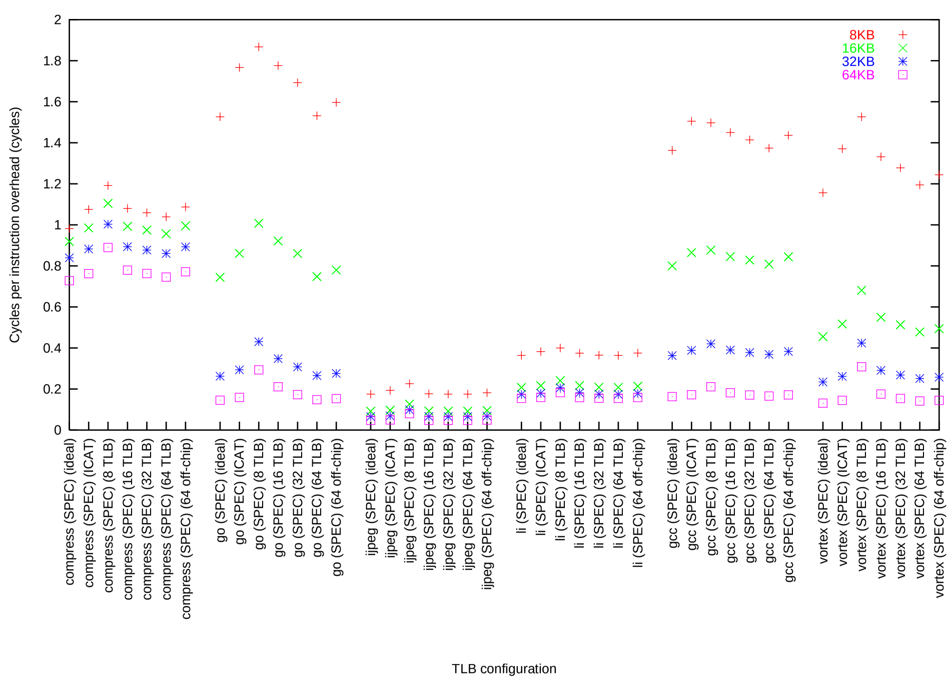

Figure 4.18: CPI overhead vs. cache size for SPEC benchmarks.

4.3.2 Cache associativity

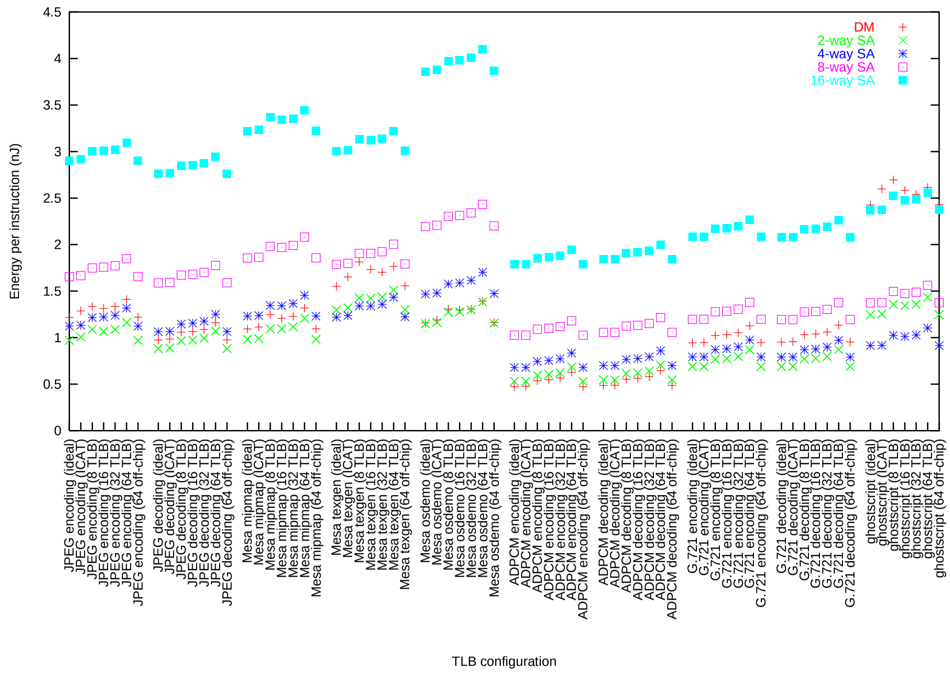

Figure 4.19: EPI overhead vs. cache associativity for MediaBench group 1.

For this parameter, set associativities greater than 16-way were not considered as the CACTI 3.0 model did not allow for them. It should be noted however, that some energy efficient cache designs do have these associativities and should be considered in future work with an appropriate model.

- Little change in the EPI overhead relative to the ideal is evident with the set associative caches. It is the differences in associativity that dominate the differences in EPI overhead.

- Direct mapped caches, in some cases such as Mesa texgen, ghostscript and vortex show slightly worse behaviour of no-TLB and small TLBs on the cache hit path due to increased conflict misses between PTEs and application data.

- The anomaly in the RASTA benchmarks disappears when set-associativity increases to and beyond 4-way. This demonstrates that the correct explanation for this behaviour is a degenerate set conflict.

- While not directly applicable to the comparison of address translation mechanisms, the cache associativities that gave the best energy efficiency have been summarised in Table 4.2 below.

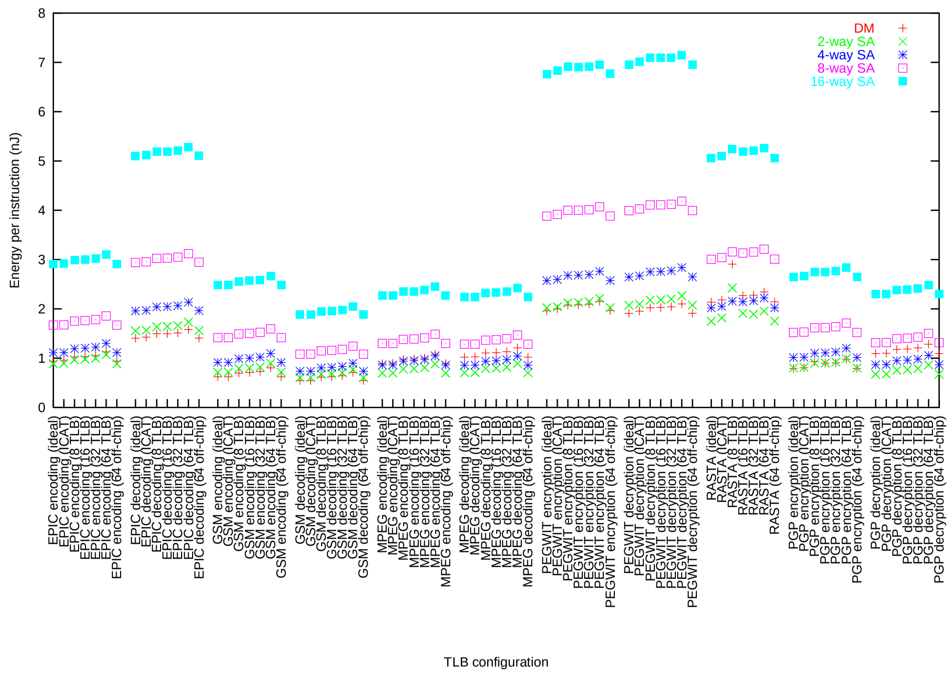

Figure 4.20: EPI overhead vs. cache associativity for MediaBench group 2.

Cache associativity	Number of benchmarks
DM	9
2-way	14
4-way	4
8-way	0
16-way	0

Table 4.2: Cache associativity with greatest energy efficiency

The 2-way set-associativity used in the baseline configuration is clearly the more energy-efficient in the majority of benchmarks.

- It is again observed that differences between associativities, where a significant difference in CPI is observed, are more significant than differences between address translation mechanisms.
- Direct-mapped caches perform significantly worse in terms of CPI overhead in over half the benchmarks to the other associativities. In the benchmarks described in the EPI summary for cache associativity in which ICAT performed poorly on DM caches, poor relative CPI overhead is also observed. An off-chip TLB provided some improvement in these cases.

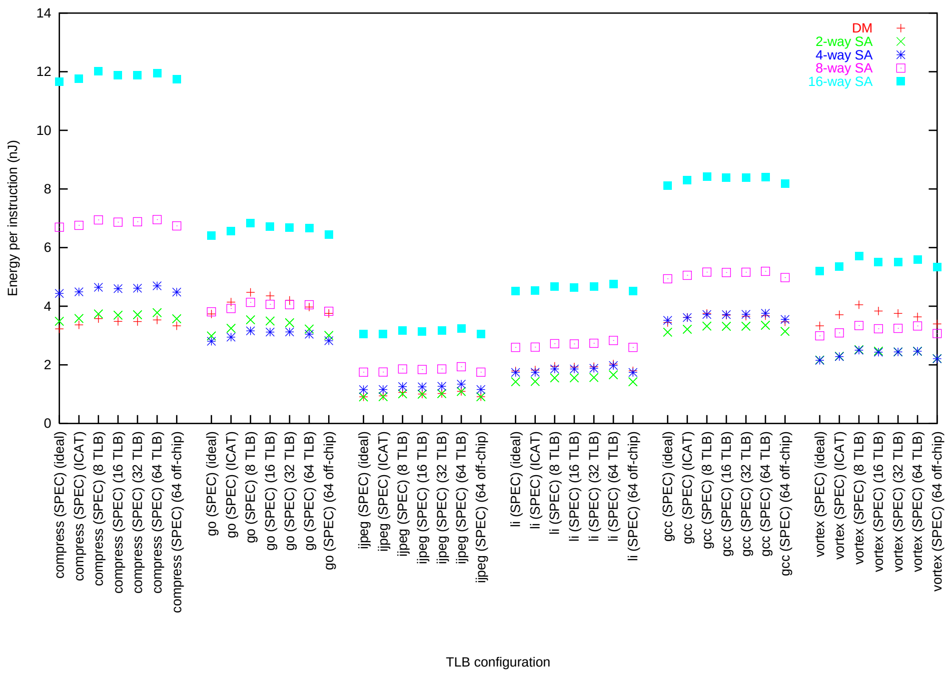

Figure 4.21: EPI overhead vs. cache associativity for SPEC benchmarks.

- The most energy efficient associativity for a benchmark also tended to be close in terms of CPI overhead to the more highly associative caches, suggesting that selection of cache associativity based primarily on energy efficiency yields acceptable performance.

- The MediaBench benchmarks show little improvement in CPI overhead beyond 2-way set-associativity, mostly. The SPEC benchmarks go, gcc and vortex demonstrated a larger spread in CPI, suggesting that some of the larger number of misses evident in these benchmarks from high contribution to the CPI and EPI from memory in the baseline configuration, is a result of conflict rather than just capacity misses.

Cache associativity can be summarised as being a parameter of significance at or below 2-way set associativity to the relative performance, but of little significance beyond this. A 2-way associativity is most energy efficient in the majority of cases, and performs, in terms of CPI overhead, close to higher associativities, suggesting this as the correct choice for an embedded system from the associativities considered.

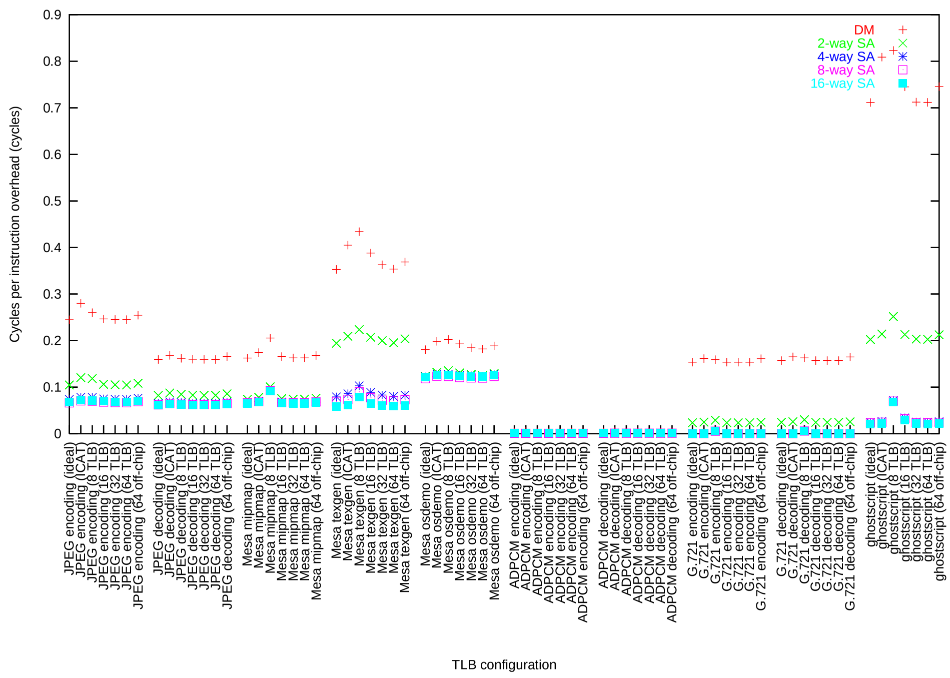

Figure 4.22: CPI overhead vs. cache associativity for MediaBench group 1.

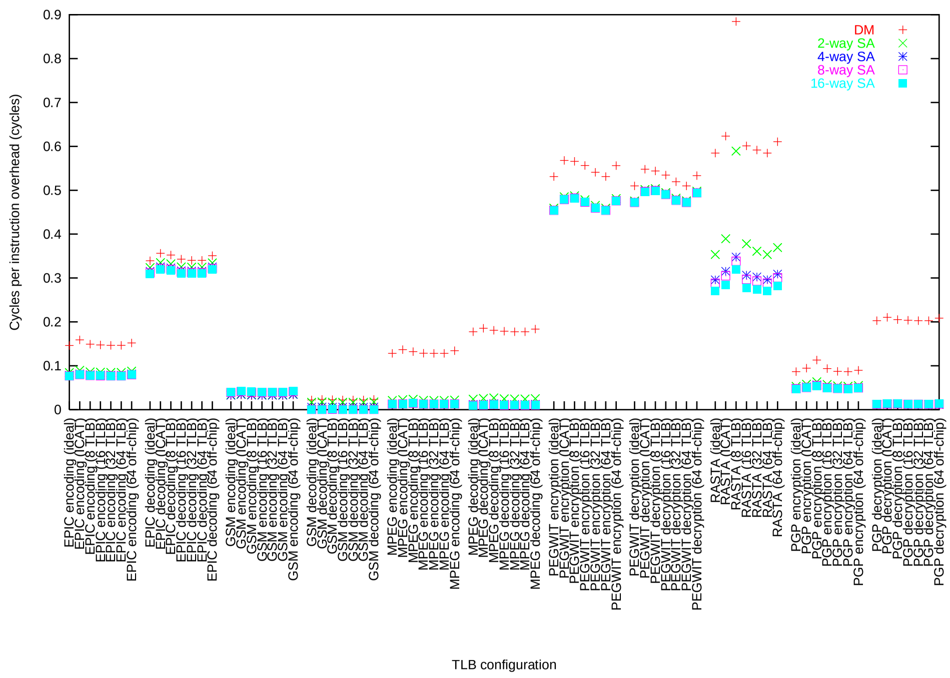

Figure 4.23: CPI overhead vs. cache associativity for MediaBench group 2.

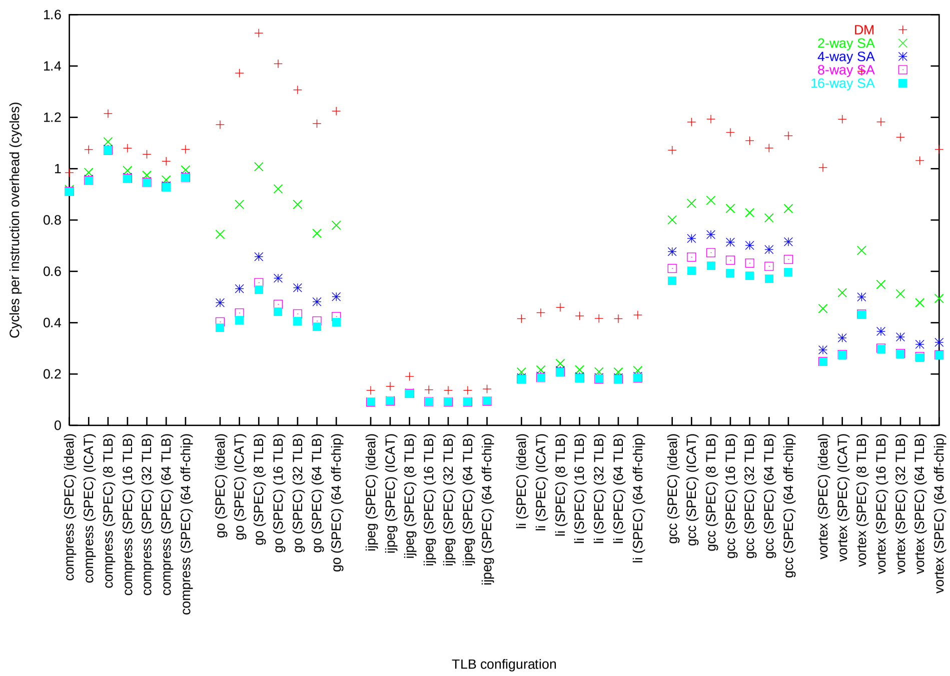

Figure 4.24: CPI overhead vs. cache associativity for SPEC benchmarks.

4.3.3 Line size

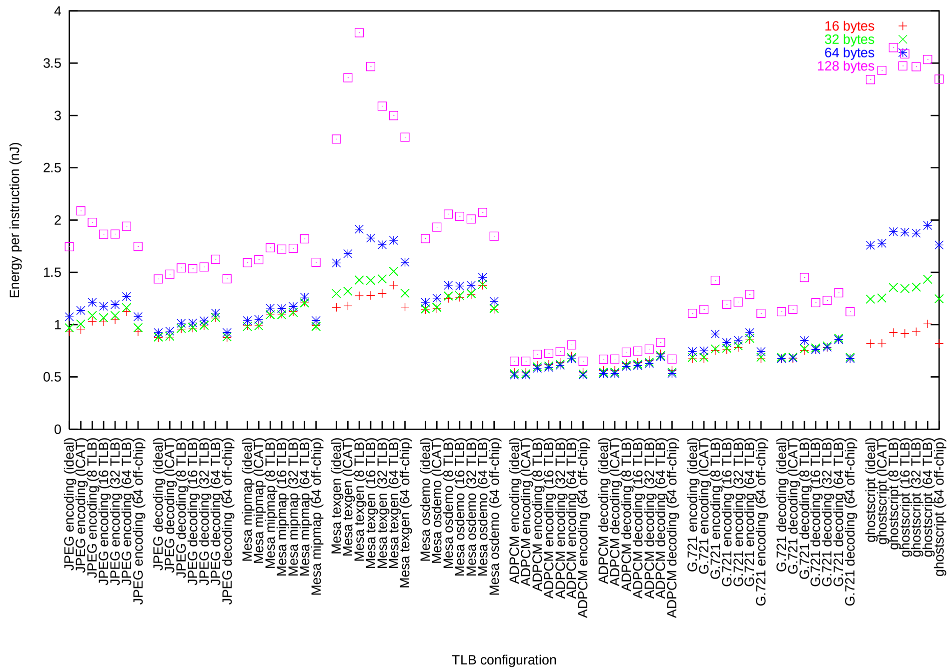

Figure 4.25: EPI overhead vs. line size for MediaBench group 1.

- 16 and 32 byte line sizes can be seen to be generally the most energy efficient, given the other parameters assumed in baseline configuration.
- Those benchmarks in which memory access energy was significant in the baseline configuration show the greatest variance with line size. For these benchmarks, large cache line sizes adversely affected energy efficiency.
- Unlike with EPI overhead, the CPI overhead shows no noticeable change in the relationship between the performance of address translation mechanisms on the less efficient larger line sizes. Mesa texgen, go and compress have ICAT worsen relative to larger TLB sizes with larger line sizes.

In conclusion, line size is a parameter that, like cache size, is important to energy efficiency and cycle overhead, but, with the exception of the larger line sizes, which are unrealistic given their poor performance, shows little change between address translation mechanisms. If the most energy-efficient line sizes, 16–32 bytes, are used, then the results from the baseline configuration also hold.

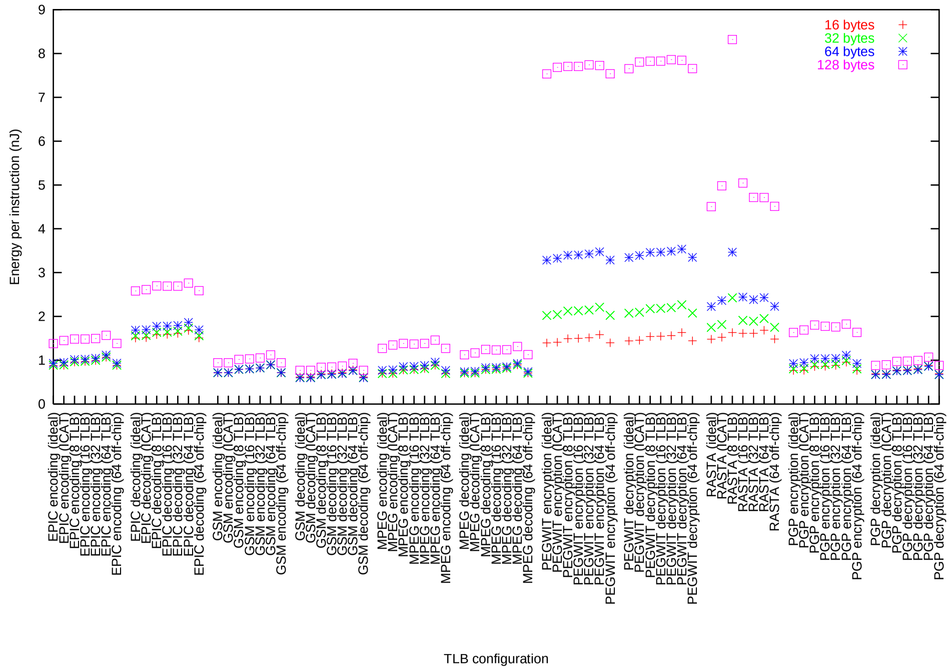

Figure 4.26: EPI overhead vs. line size for MediaBench group 2.

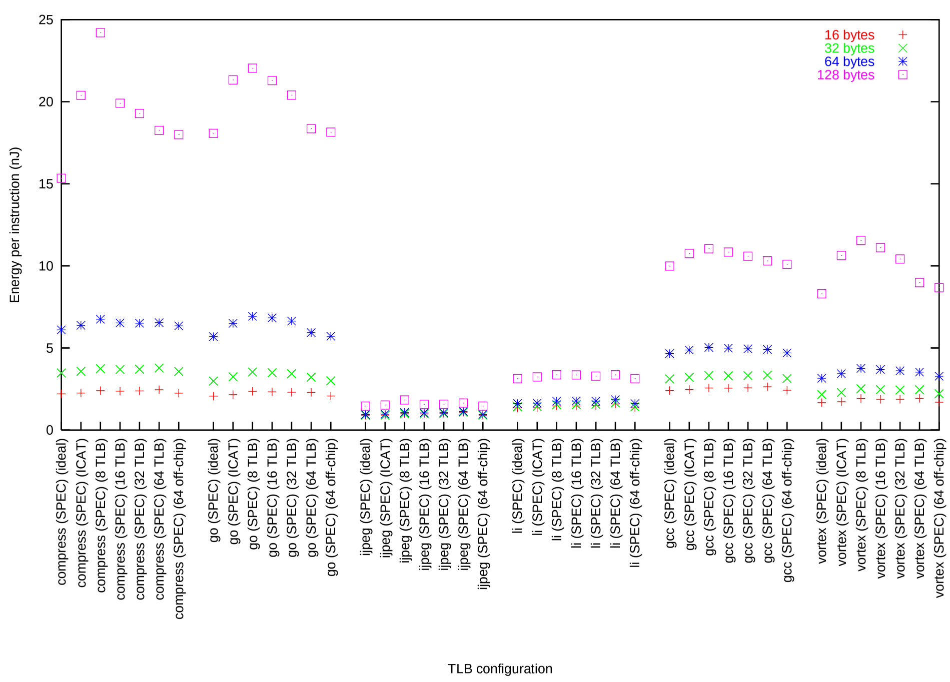

Figure 4.27: EPI overhead vs. line size for SPEC benchmarks.

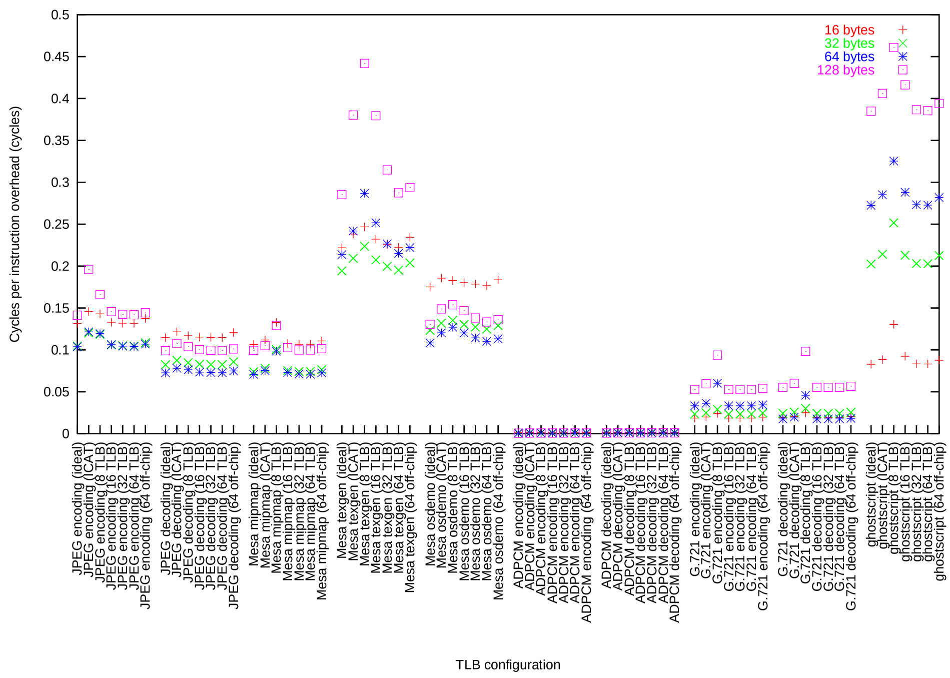

Figure 4.28: CPI overhead vs. line size for MediaBench group 1.

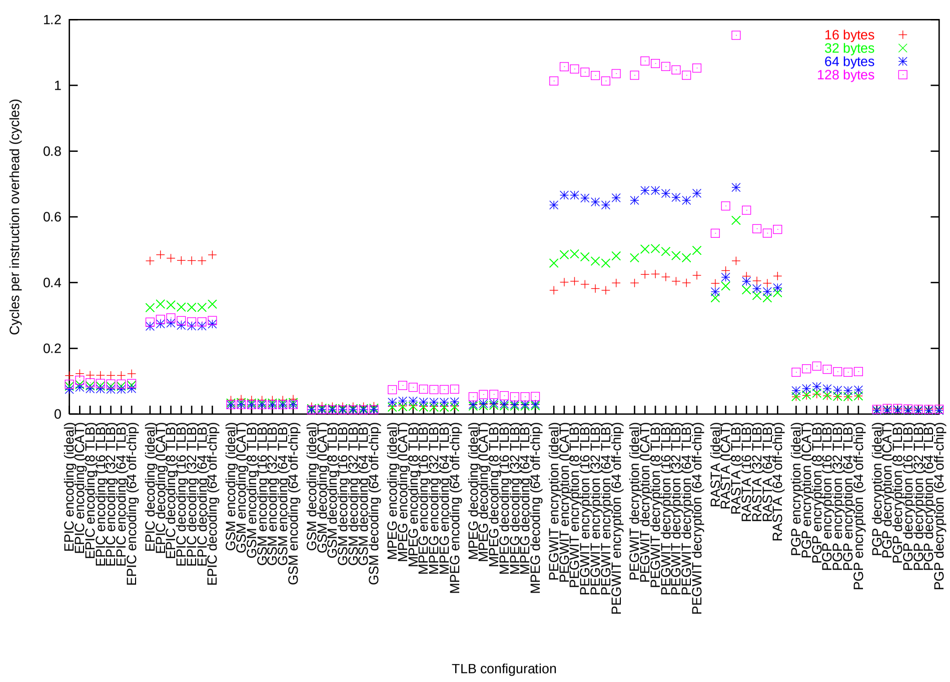

Figure 4.29: CPI overhead vs. line size for MediaBench group 2.

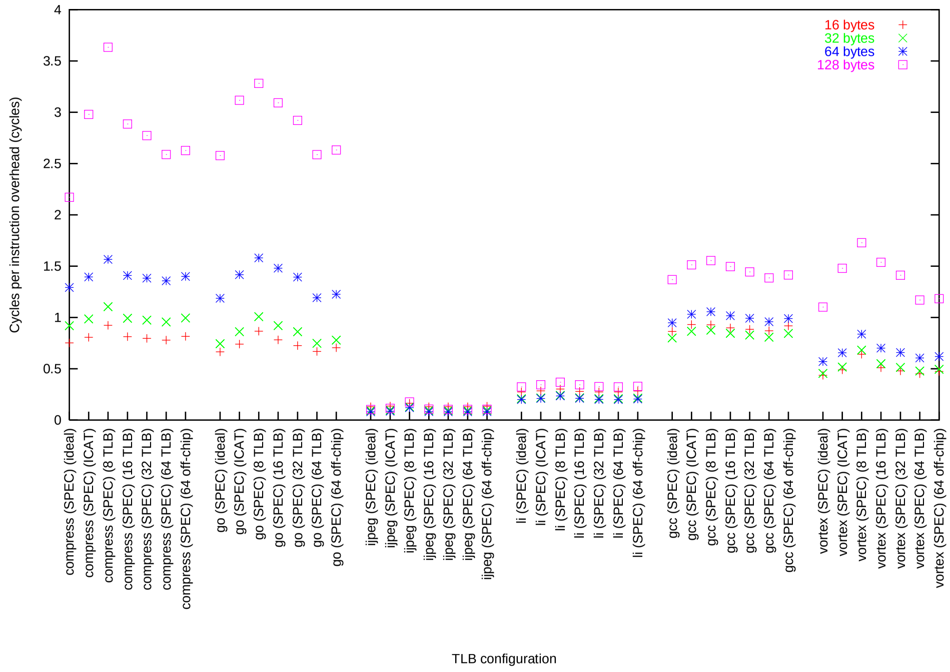

Figure 4.30: CPI overhead vs. line size for SPEC benchmarks.

4.3.4 Page size

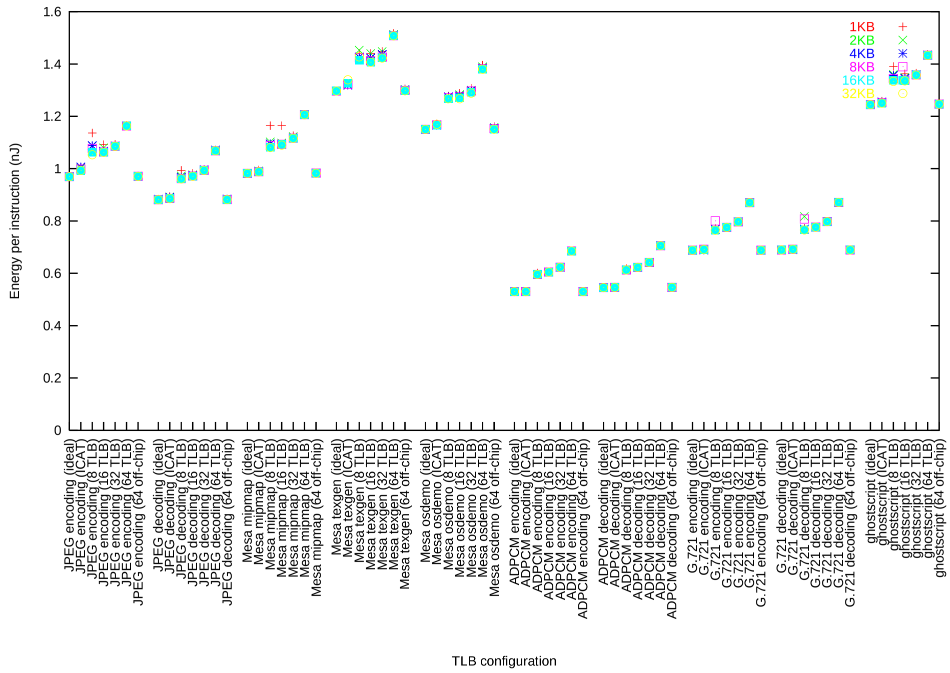

Figure 4.31: EPI overhead vs. page size for MediaBench group 1.

- Page size had little effect on energy efficiency for the MediaBench benchmarks. A slight improvement from larger page sizes is evident in the SPEC benchmarks. G.721 and vortex show anomalous behaviour here, with some larger page sizes showing slightly worse energy efficiency than others, possibly due to a set conflict.

- Page size significantly affected the CPI overhead for some benchmarks, but left others mostly unchanged.

- 1KB page sizes gave a large overhead for the smaller TLB on the cache hit path, visibly demonstrated in the JPEG encoding and go SPEC benchmarks. For the SPEC benchmarks, where the affects of different page sizes are most visible, the TLBs on the cache hit path are the most affected, with larger spreads than the other configurations.

- Beyond 8KB the returns on increased page size diminish.

- With the exception of the compress benchmark, where performance improved with page size for TLB configurations on the cache hit path, it did not do so

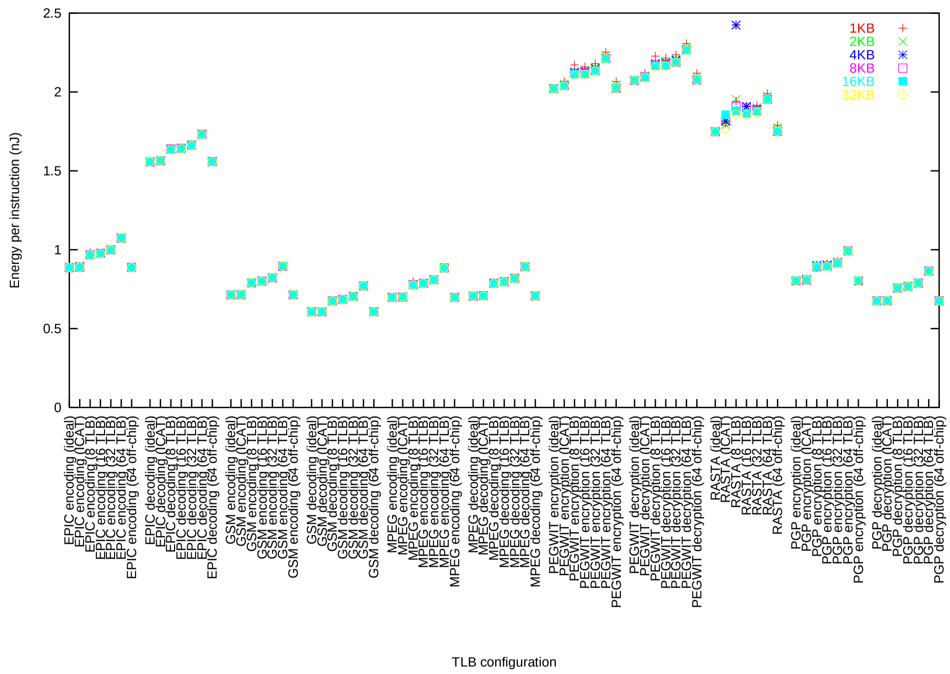

Figure 4.32: EPI overhead vs. page size for MediaBench group 2.

as dramatically for the off-chip TLB. This suggests that an off-chip TLB may not benefit from superpage support. It should be noted, however, that a single PTE was used to map a single page, and so with multiple page size support the off-chip TLB may offer some reduction in cache pollution.

- The smaller TLBs appear to suffer more than ICAT, suggesting that it is the TLB refill costs that dominate for smaller page sizes rather than cache pollution.

Page size can be summarised as significantly affecting the relative performance of address translation mechanisms in terms of CPI overhead but not in terms of EPI overhead. Smaller page sizes adversely affected smaller TLBs on the cache hit path while leaving those configurations with translation after the cache mostly unaffected. Larger page sizes, and so efficient use of superpages by the operating system, improve the CPI significantly for a few benchmarks, such as go and compress, suggesting that the results for the baseline configuration for these would be affected somewhat by an increased page size beyond the 4KB considered in the baseline configuration.

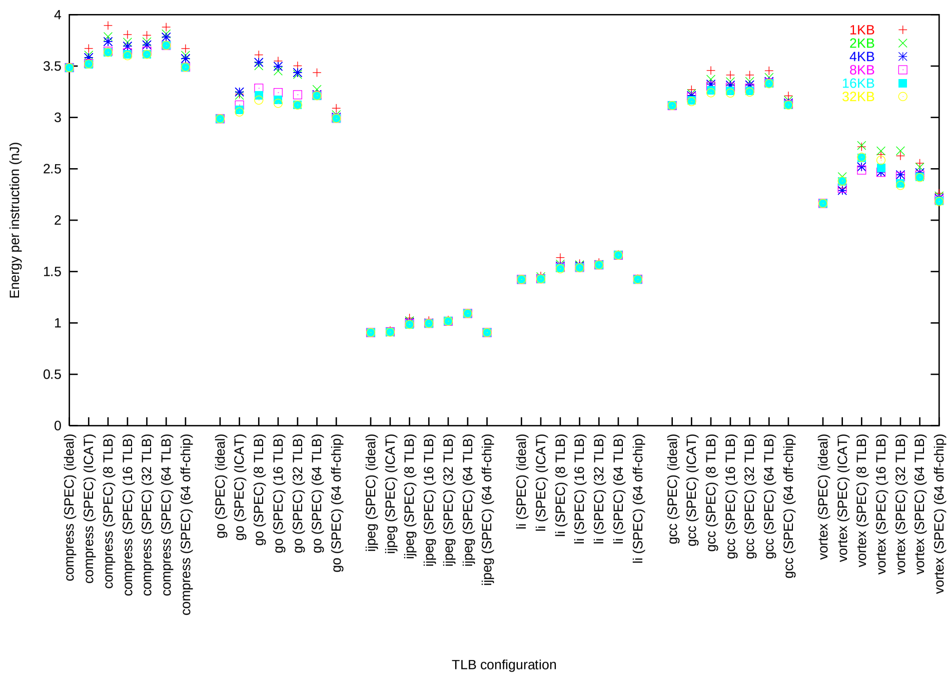

Figure 4.33: EPI overhead vs. page size for SPEC benchmarks.

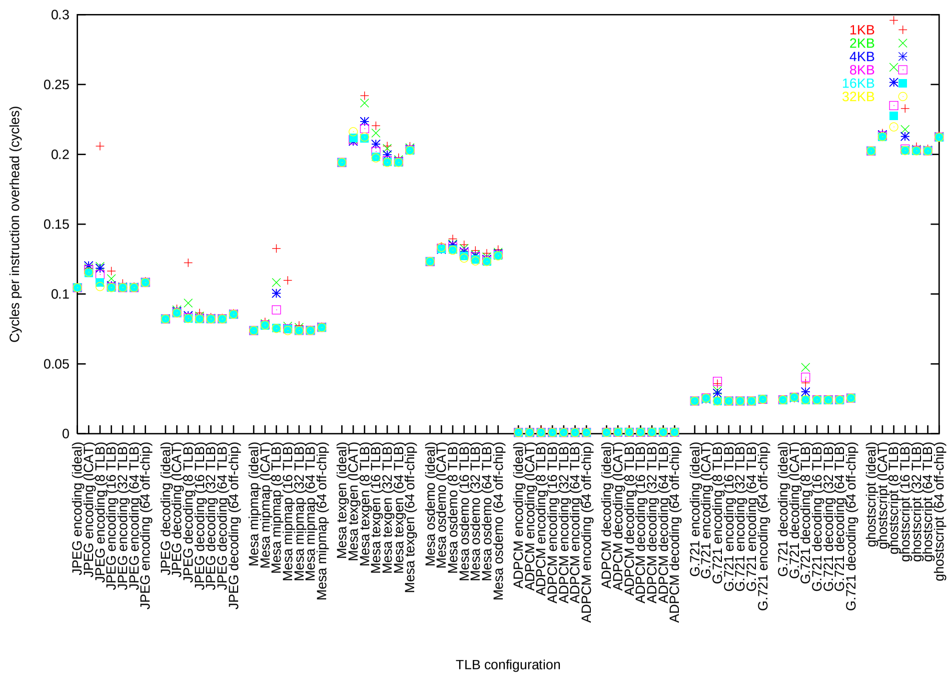

Figure 4.34: CPI overhead vs. page size for MediaBench group 1.

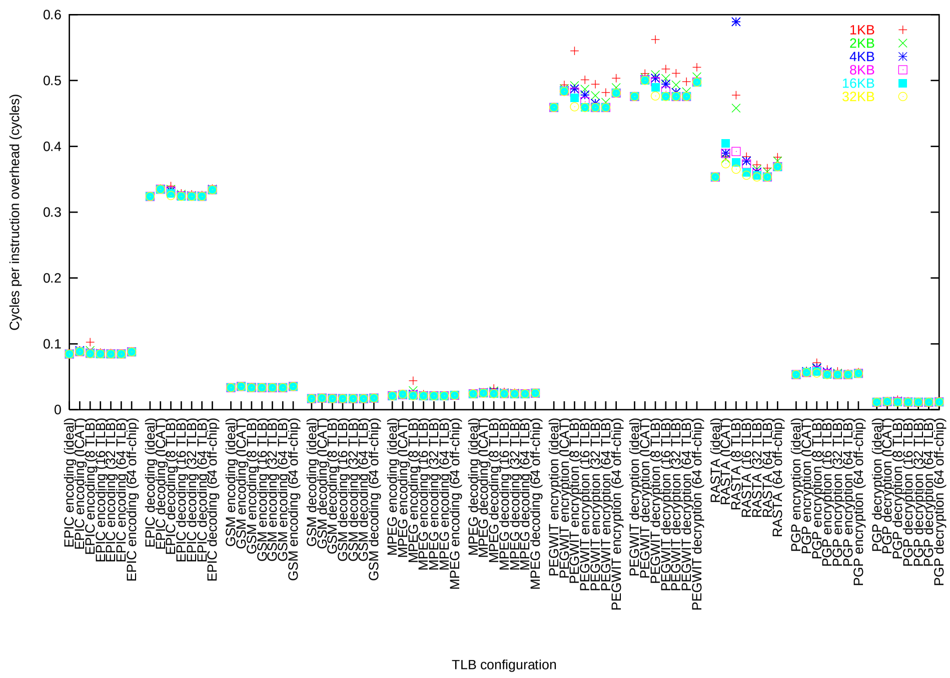

Figure 4.35: CPI overhead vs. page size for MediaBench group 2.

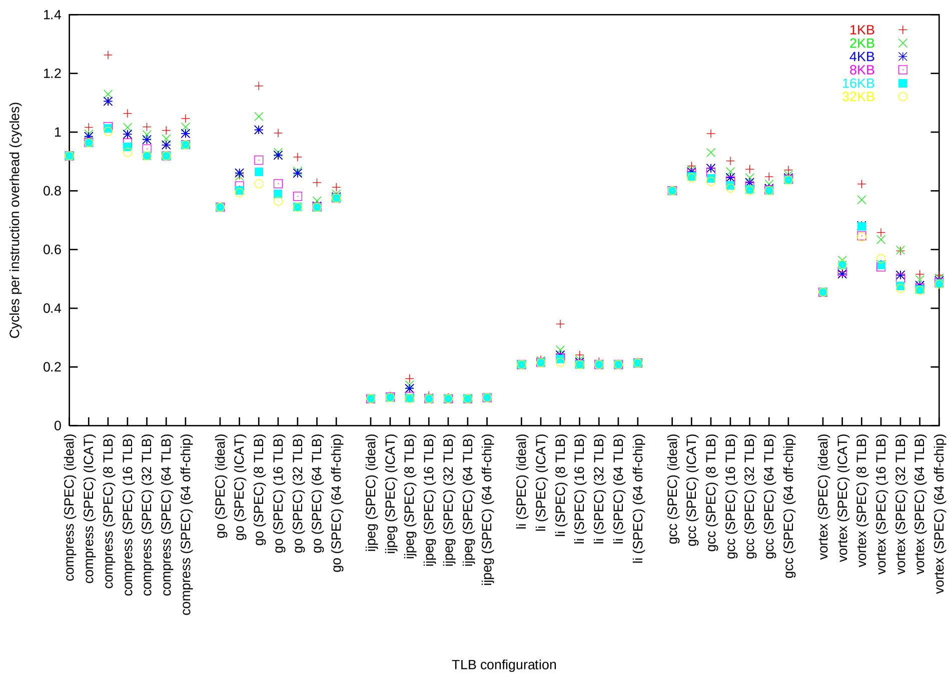

Figure 4.36: CPI overhead vs. page size for SPEC benchmarks.

4.3.5 Handler cycles

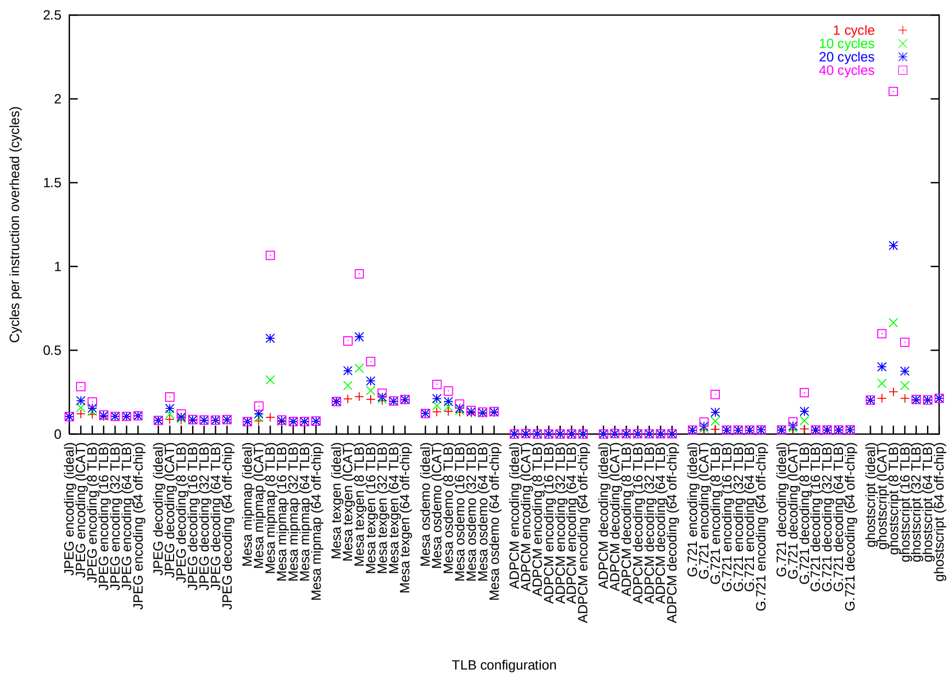

Figure 4.37: CPI overhead vs. translation miss handler cycles for MediaBench group 1.

- Several benchmarks remained unaffected by a change to miss handler cost: JPEG, ADPCM, EPIC, GSM, MPEG, and PGP. The conclusion that can be drawn from this is that these benchmarks have a low TLB miss rate and therefore low working set of pages.
- A very high increase in overhead is observed for larger handler cycle costs with smaller TLBs on the cache hit path and ICAT. With the exception of ghostscript and EPIC, the worst affected is the 8-entry TLB. ghostscript and EPIC performance might be due to the cache working set size being greater than that of the page working set size.
- Off-chip TLBs remain unaffected by an increase in handler cycle costs. Therefore off-chip TLBs might suit software translation miss handlers.

Handler cycles significantly affect the relative CPI overhead of address translation mechanisms. The original assumption of 1 cycle, when increased, makes ICAT and smaller TLBs clearly unsuitable for a number of different benchmarks. ICAT is therefore only suitable where hardware translation miss handlers are used. An off-chip TLB is more suited to the case where a software translation miss handler is used.

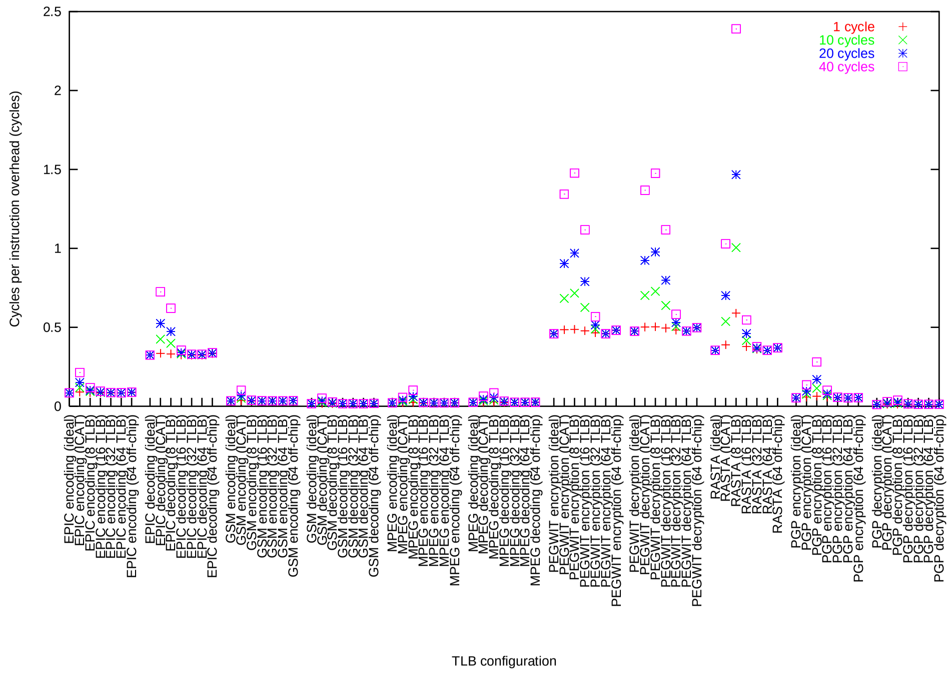

Figure 4.38: CPI overhead vs. translation miss handler cycles for MediaBench group 2.

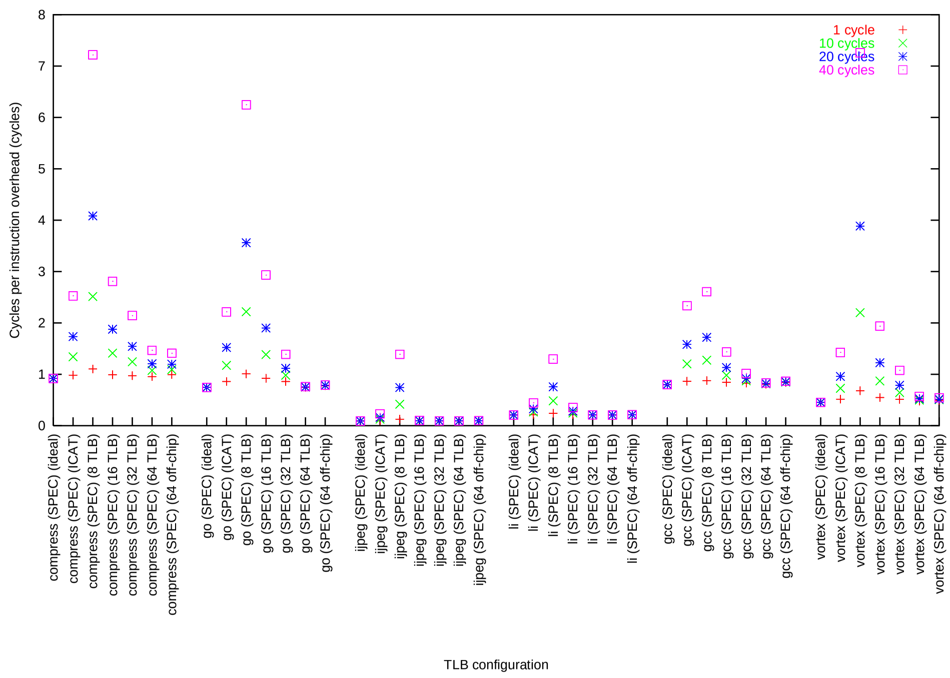

Figure 4.39: CPI overhead vs. translation miss handler cycles for SPEC benchmarks.

4.3.6 Line fetch costs

As can be seen in the graphs below, varying line fetch costs only shifts the absolute values of the overheads, and does not affect the relative performance of the address translation mechanisms in almost all cases. The go and rasta benchmarks are the exceptions, in which the original observations are accentuated by an increase in line fetch cost, as these are the only benchmarks in the baseline configuration that showed a large variance in the data cache miss rates between address translation mechanisms. Those most affected are those in which memory access was the dominant component of the baseline configuration overheads.

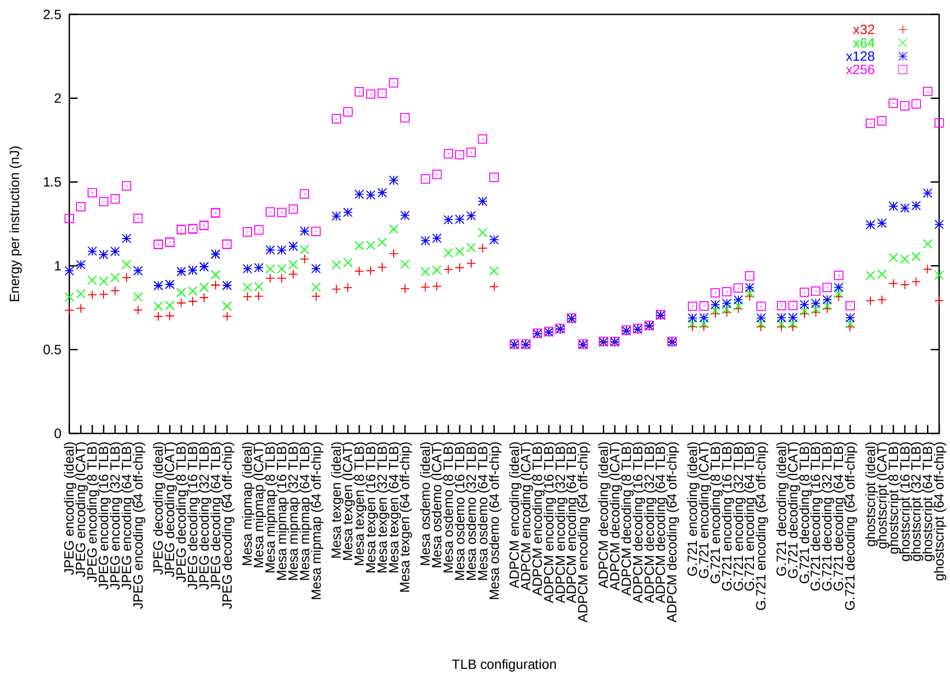

Figure 4.40: EPI overhead vs. line fetch energy cost for MediaBench group 1.

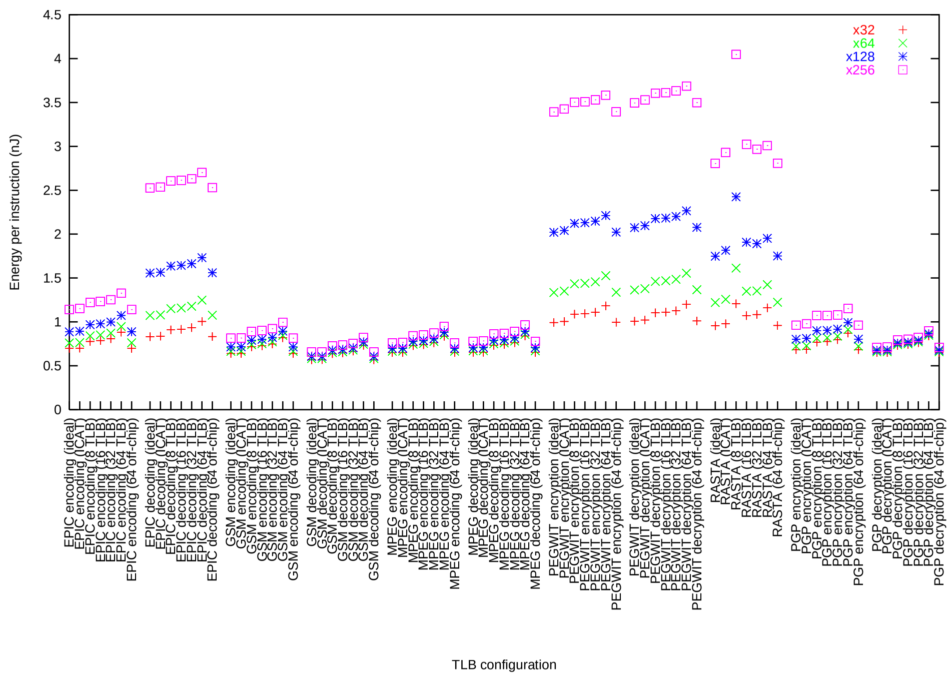

Figure 4.41: EPI overhead vs. line fetch energy cost for MediaBench group 2.

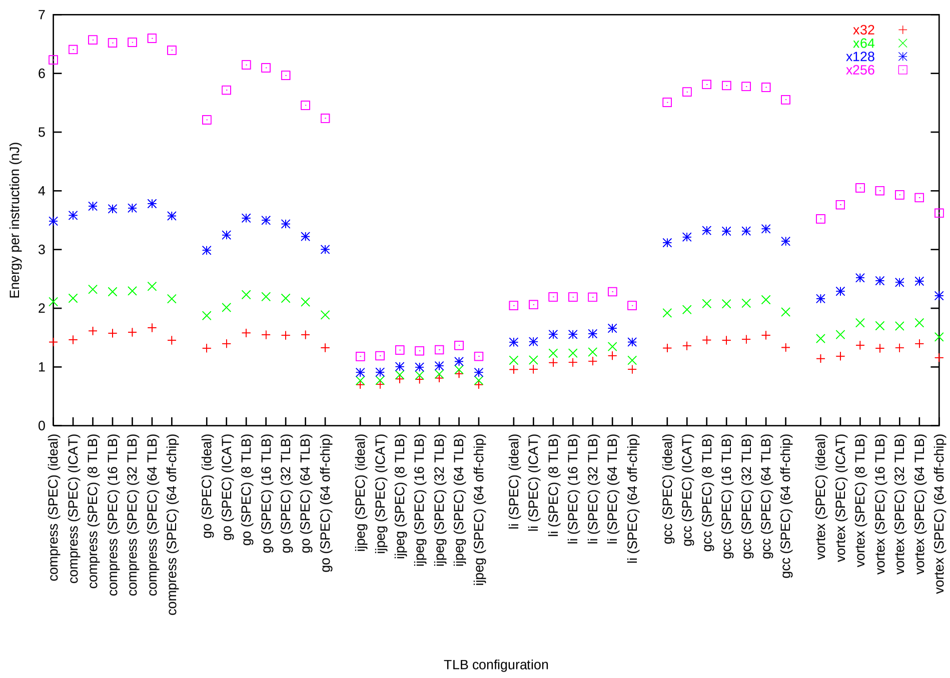

Figure 4.42: EPI overhead vs. line fetch energy cost for SPEC benchmarks.

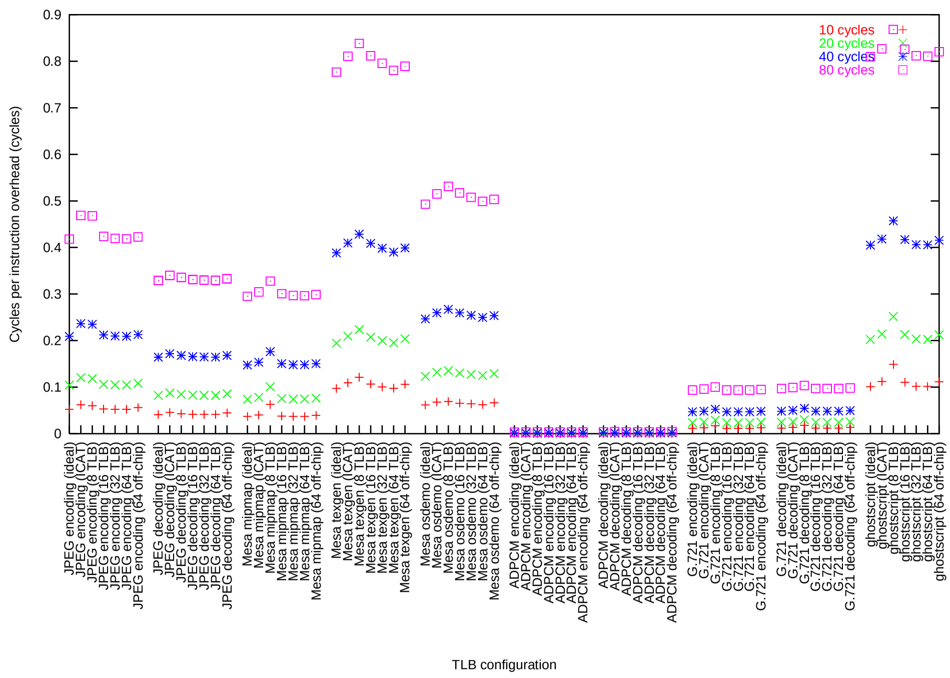

Figure 4.43: CPI overhead vs. line fetch time cost for MediaBench group 1.

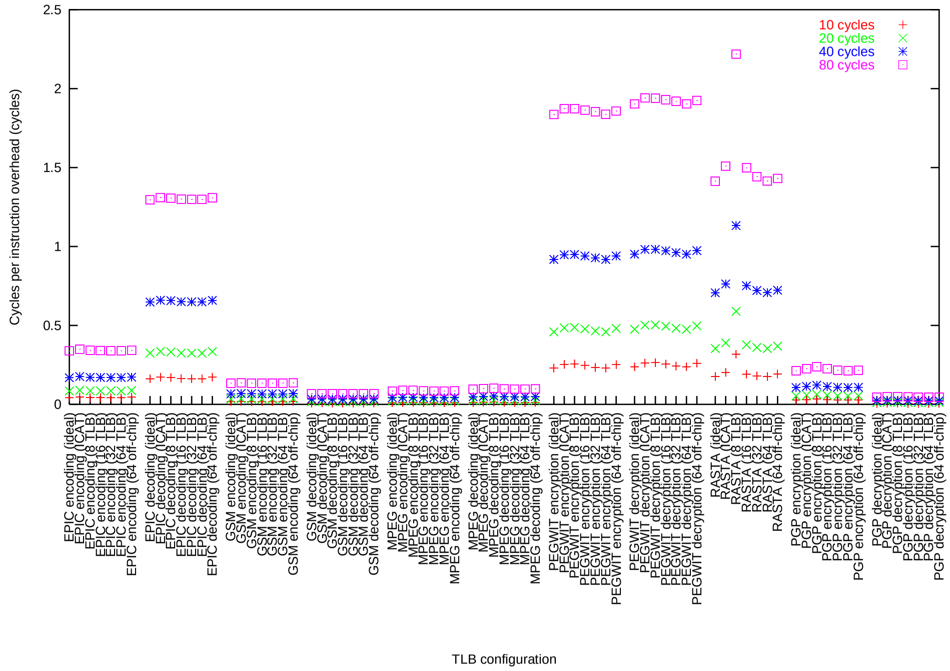

Figure 4.44: CPI overhead vs. line fetch time cost for MediaBench group 2.

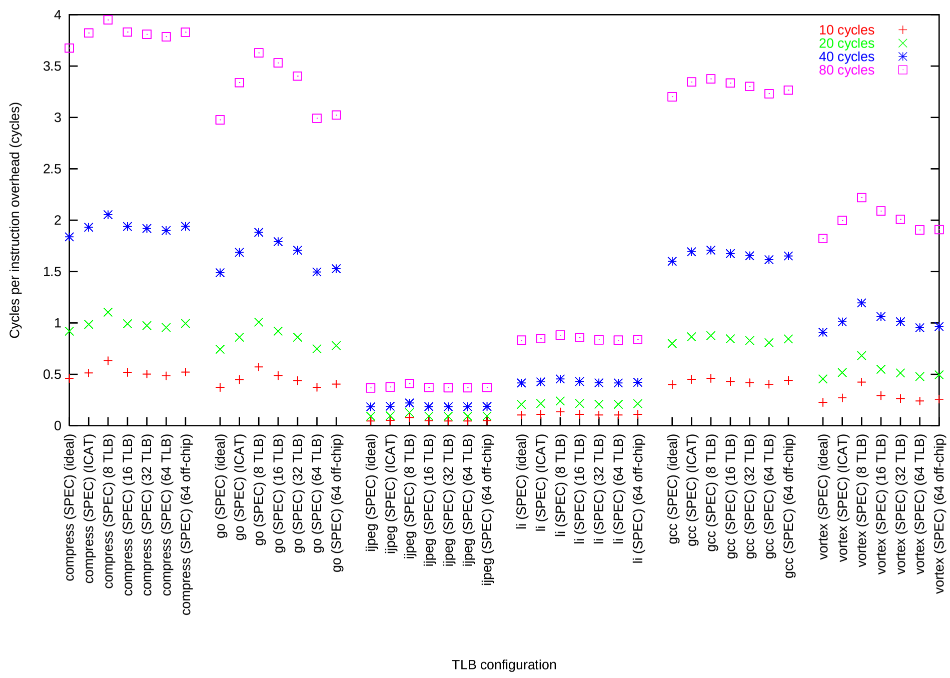

Figure 4.45: CPI overhead vs. line fetch time cost for SPEC benchmarks.

4.3.7 ASID

The addition of an 8-bit address space identifier diminished much of the gains the removal of the TLB from the cache hit path gave. The instruction and data cache hits energies now are clearly higher for those configurations which do not have a TLB on the cache hit path.

In the case of ADPCM encoding, which previously provided the best case gain of 11% reduction in overhead, it can be clearly seen that this has been approximately halved. The EPI overhead is now no longer indistinguishable from the ideal in all benchmarks. In some benchmarks, for example PEGWIT, compress, go and gcc, the ICAT overheads are now greater than those from the TLB configurations on the cache hit path. It can be seen in the graphs that the benefits to be gained from the removal of a TLB from the cache hit path is much smaller in all other cases.

Address space protection through the use of an ASID then makes ICAT or off-chip TLB configurations far less desirable from an energy efficiency perspective than found in the baseline configuration.

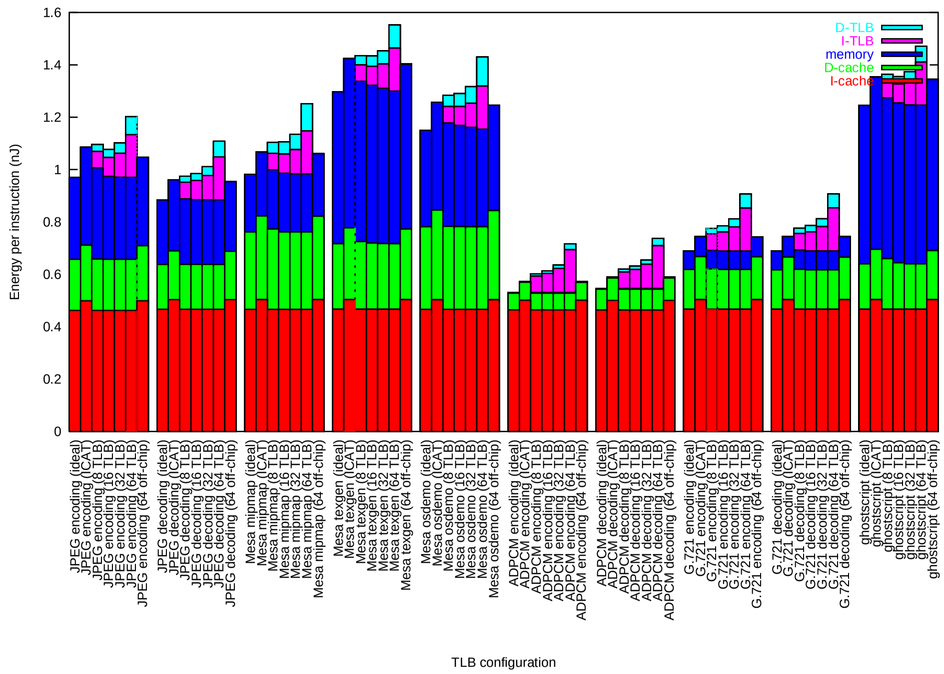

Figure 4.46: Energy per instruction overhead with ASID on MediaBench group 1.

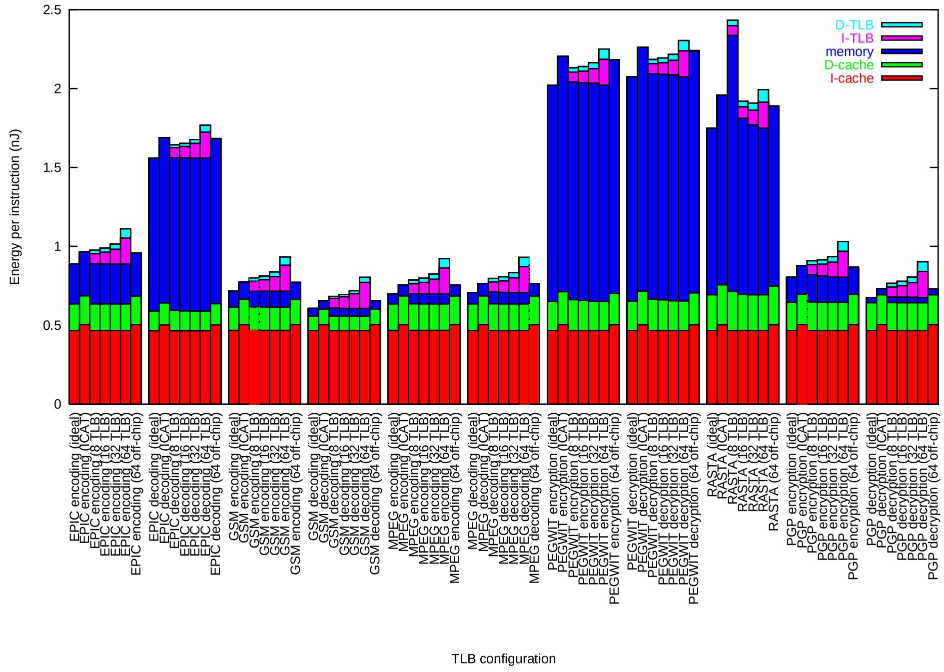

Figure 4.47: Energy per instruction overhead with ASID on MediaBench group 2.

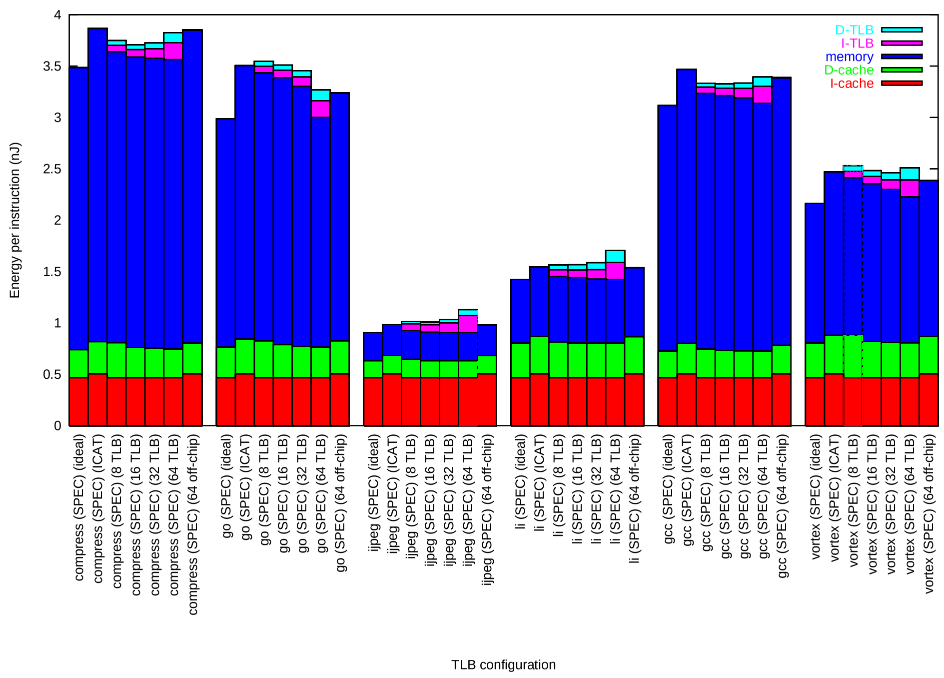

Figure 4.48: Energy per instruction overhead with ASID on SPEC benchmarks.

Benchmark name	Number of protection changes
ADPCM encoding	4
ADPCM decoding	4
EPIC encoding	22
EPIC decoding	18
G.721 encoding	7
G.721 decoding	7
ghostscript	162
GSM encoding	8
GSM decoding	8
JPEG encoding	10
JPEG decoding	16
Mesa mipmap	24
Mesa texgen	25
Mesa osdemo	36
MPEG encoding	55
MPEG decoding	13
PEGWIT encryption	12
PEGWIT decryption	10
PGP encryption	57
PGP decryption	65
RASTA	21
gcc (SPEC)	288
go (SPEC)	5
vortex (SPEC)	3108
compress (SPEC)	5
li (SPEC)	37
ijpeg (SPEC)	60

Table 4.3: Number of compulsory protection changes for benchmarks.

4.3.8 Protection

In Table 4.3 the number of compulsory protection changes, as described in the previous chapter, is listed for each benchmark. It should be clear that these are small relative to the number of instruction references executed, and only vortex shows a noticeable number of protection changes. The graphs consequently show no noticeable EPI or CPI overhead from the update to cache line for these, and so have not been included. This is, of course, optimistic, and the real overheads from in-cache protection rights will depend on operating system structure and performance.

4.3.9 Hierarchical TLB

By only performing a TLB lookup when a page boundary is crossed, a special case of hierarchical TLBs, it is clear that the overheads of a TLB on the cache hit path decrease significantly. The optimal TLB size, from an energy-efficiency perspective, now shifts to 16–64 entries, and the gains in the most optimistic case for ICAT now appear to be almost negligible.

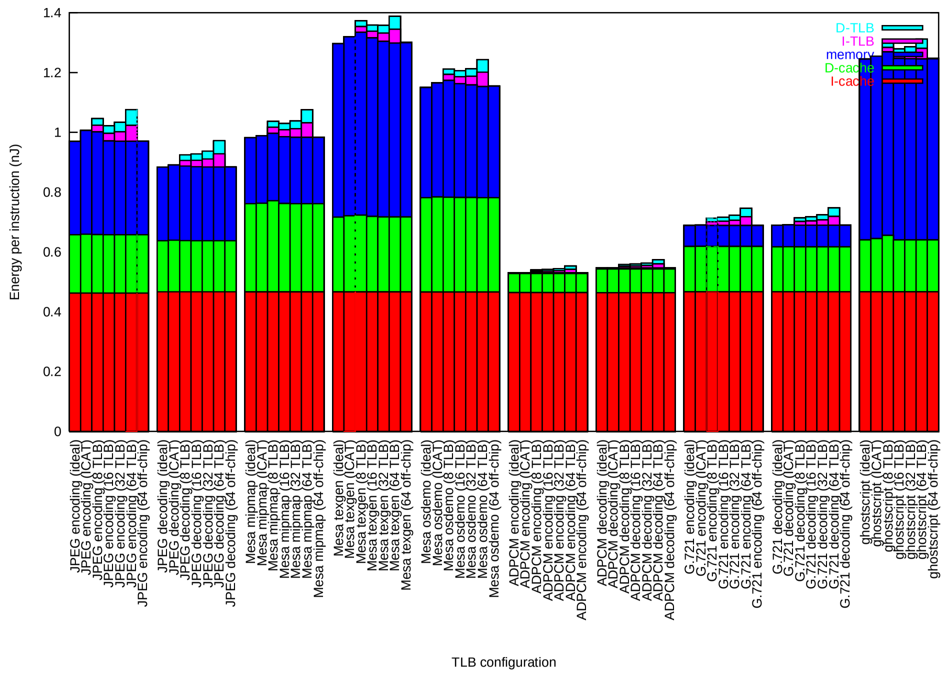

Figure 4.49: Energy per instruction overhead with only necessary TLB lookups on MediaBench group 1.

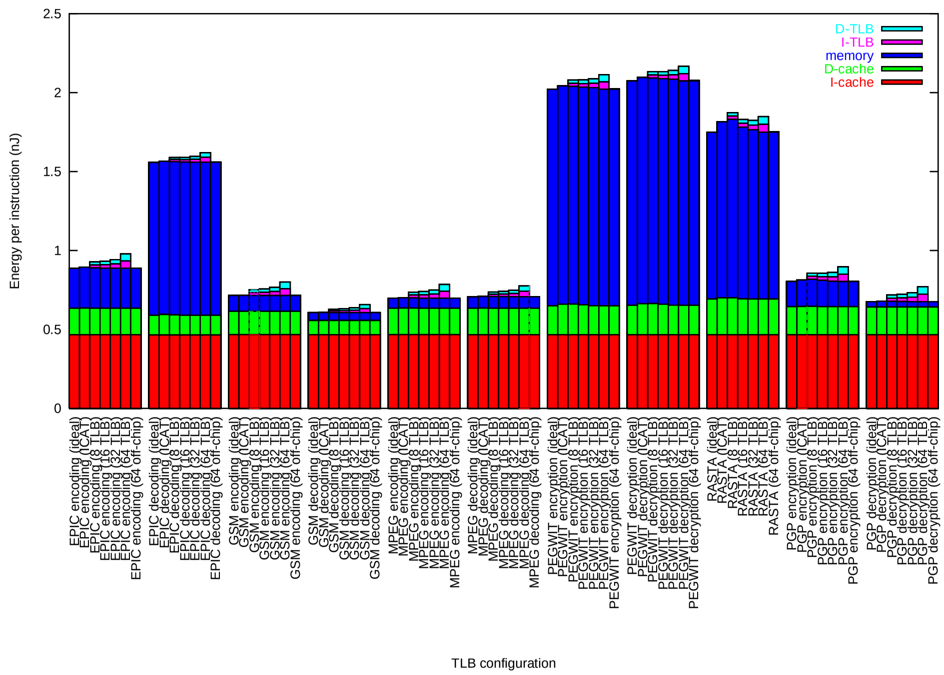

Figure 4.50: Energy per instruction overhead with only necessary lookups on MediaBench group 2.

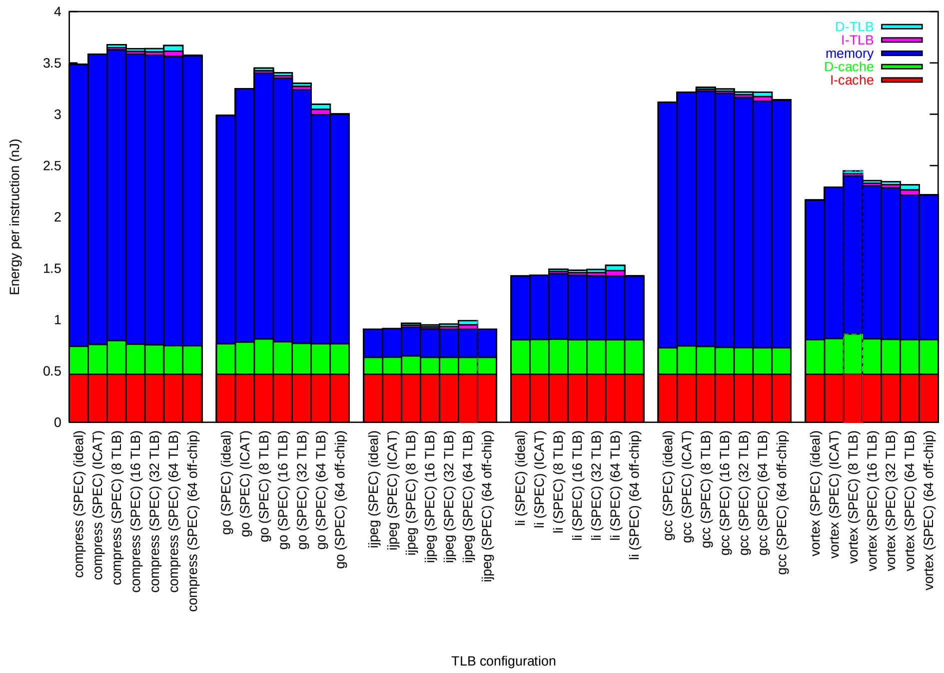

Figure 4.51: Energy per instruction overhead with only necessary lookups on SPEC benchmarks.

4.4 Summary

In summary, this chapter has presented the results in the form of graphs indicating EPI and CPI overheads, and discussion for the baseline configuration and sensitivity analysis.

The alternatives to the traditional placement of the TLB on the cache hit path were shown to be generally more energy efficient but to suffer some cycle penalty in the baseline configuration, leading to the conclusion that this architectural feature allows a tradeoff to be made between energy consumption and performance. Savings of up to 11% of the ideal memory system EPI were found in the best case, and cycles overheads of 17% of the ideal found in the worst. For configurations with a conventional TLB, a 16 or 32 entry TLB appeared to be a good compromise between energy efficiency and cycle overhead. Cache pollution was found to be significant in some benchmarks, and larger TLBs have been shown to play a role in the reduction of this.

Sensitivity analysis showed that cache size was a significant contributor to EPI overhead, but that when the most energy efficient cache size was used that little difference in the relative performance of the address translation mechanisms to the baseline configuration was evident. Increasing cache size also, expectedly, improved the performance of translation mechanisms not on the cache hit path by the reduction of cache capacity miss rates. The 2-way set-associativity used in the baseline configuration was found to be the most energy-efficient in the majority of benchmarks, and the relative results to differ little from the baseline configuration results when the most energy-efficient associativity is used. The same applies to line size, with 16–32 bytes being the most energy-efficient and have the lowest CPI overhead on almost all benchmarks. Page size was found to be a significant parameter, and to affect smaller TLBs on the cache hit path more significantly than off-chip TLBs or ICAT. Handler cycles significantly increase the CPI overhead of many benchmarks, and the conclusion that was drawn was that off-chip TLBs are the most suitable where software page table walkers are utilised. Line fetch costs were generally unimportant to the relative performance of the translation options.

The addition of an 8-bit ASID to the TLB and to the cache on configurations where there was not a TLB on the cache hit path diminished the gains that the removal of a TLB from the cache hit path gave, and in some cases made ICAT and off-chip TLB configurations the least efficient. From this it was concluded that from an energy-efficiency perspective, ICAT and off-chip TLB configurations are unsuitable when multiple address spaces are required. An optimistic look at protection change overheads showed them to be negligible. The use of improved TLB configurations on the cache hit path was also shown to diminish the utility of non-conventional address translation placement.

Chapter 5

Conclusion

In this thesis, a comparison of the address translation options available for a virtually-addressed cache has been made, in the context of embedded systems.

From the background and related work, it was seen that the provision of virtual memory, and in particular address translation, through a conventional TLB had several problems and energy and time overheads that might have been potentially improved upon by the use of virtually-addressed caches, and the deferment of address translation until cache miss. Since energy-efficiency, in particular, is of relevance to embedded systems, this provided the rationale for the experimental component of the thesis.

An experimental method was outlined and then followed, producing quantitative results from which the address translation mechanisms could be compared. It was found that, given the assumed parameters in the baseline configuration, that non-conventional address translation placement allows for energy-efficiency to be traded with cycle overhead. On sensitivity analysis, non-conventional address translation placement yielded good results on the early parameters. This suggested that the original conclusion that an off-chip TLB or the elimination of the TLB provided gains in terms of energy-efficiency. However, when other factors, such as ASIDs and improved TLB design, were introduced, the gains diminished with the cycle overheads not improving. From the experimental work it can be concluded that:

- Virtually-addressed caches may be more energy efficient than standard TLBs on embedded systems if multiple address spaces and hierarchical TLBs are not used.
- The energy-efficiency of virtually-addressed caches may be improved by work on more efficient means of providing address space protection.
- When software translation miss handling is utilised, an off-chip TLB is clearly the preferred choice from a performance point-of-view on embedded systems.

The limitations of the experimental method should be acknowledged. The only component of non-user code simulated was the TLB refill mechanism, and only single process workloads were utilised. The consistency problems associated with virtually-addressed cache have implications to operating system structure — for example SASOSes are clearly well suited to virtually-addressed caches as they do not suffer from the problems of synonyms and homonyms. On a SASOS it would be unnecessary to use an ASID, and hence greater energy efficiency may be achieved. These implications could be investigated in future work, in the context of embedded systems (many of them have been investigated in other contexts previously in the literature).

Other future work might focus on improved memory protection. Since non-conventional TLB placement allows for a separation of translation and protection mechanisms, an energy-efficient and flexible protection mechanism would be desirable. In-cache access rights rely heavily on low frequency of protection changes to be efficient — a less rigid requirement may be of utility.

Appendix A

SPEC benchmark suite cache miss ratios

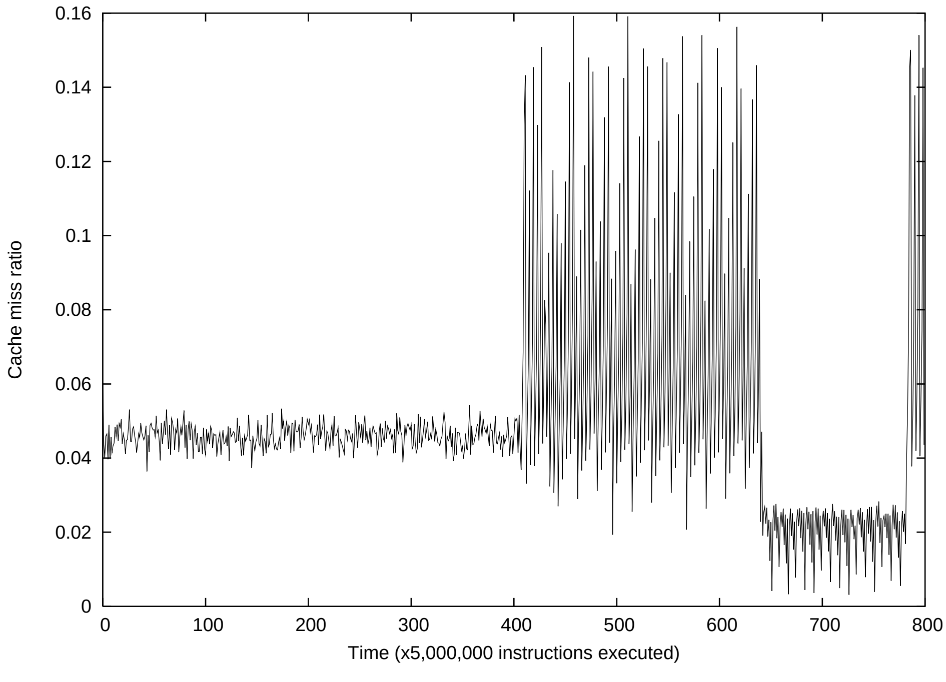

Figure A.1: Data cache miss ratio vs. time for *compress*

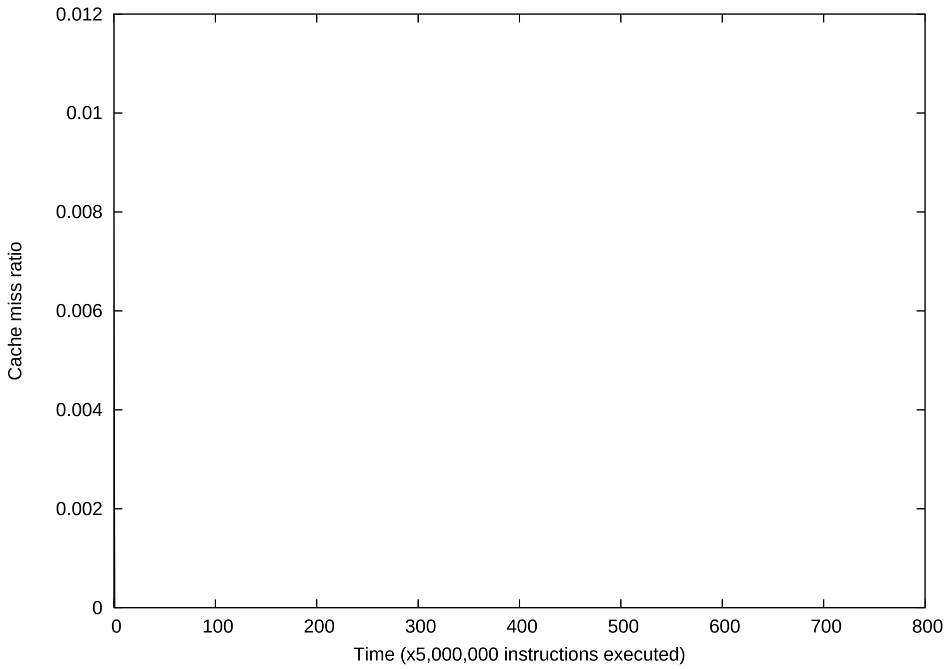

Figure A.2: Instruction cache miss ratio vs. time for *compress*

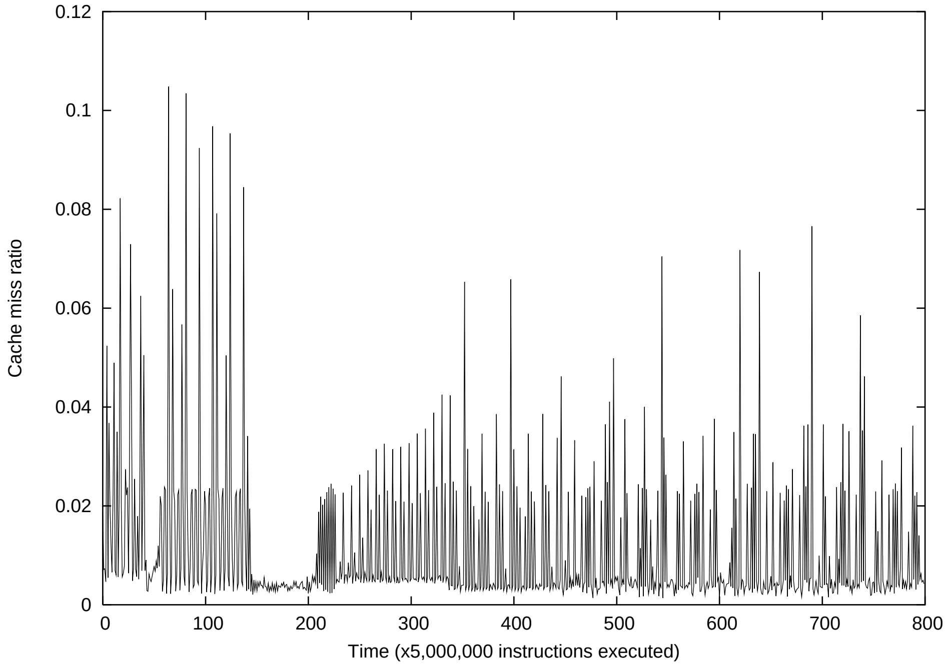

Figure A.3: Data cache miss ratio vs. time for *li*

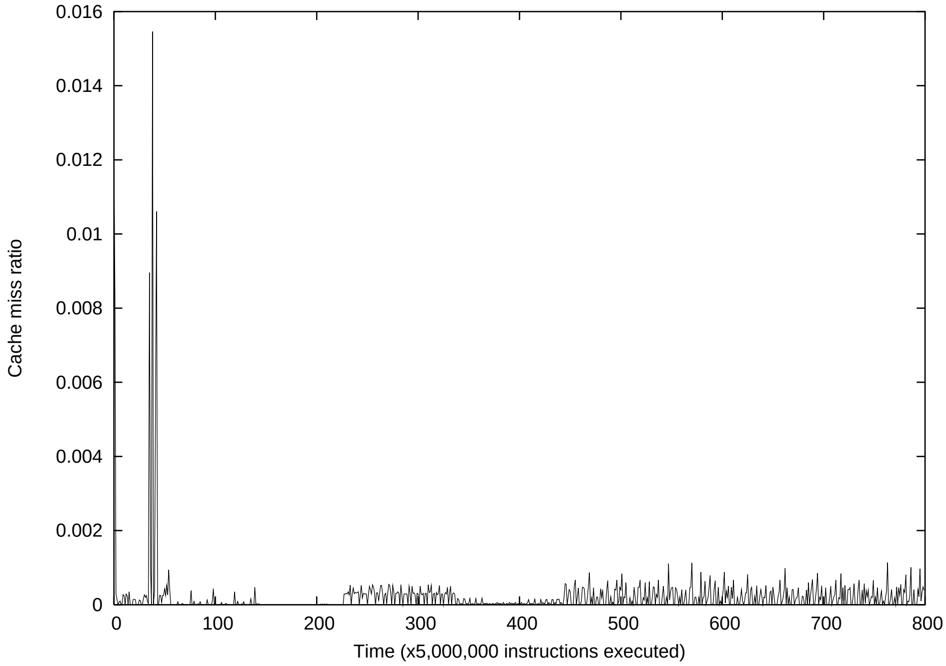

Figure A.4: Instruction cache miss ratio vs. time for *li*

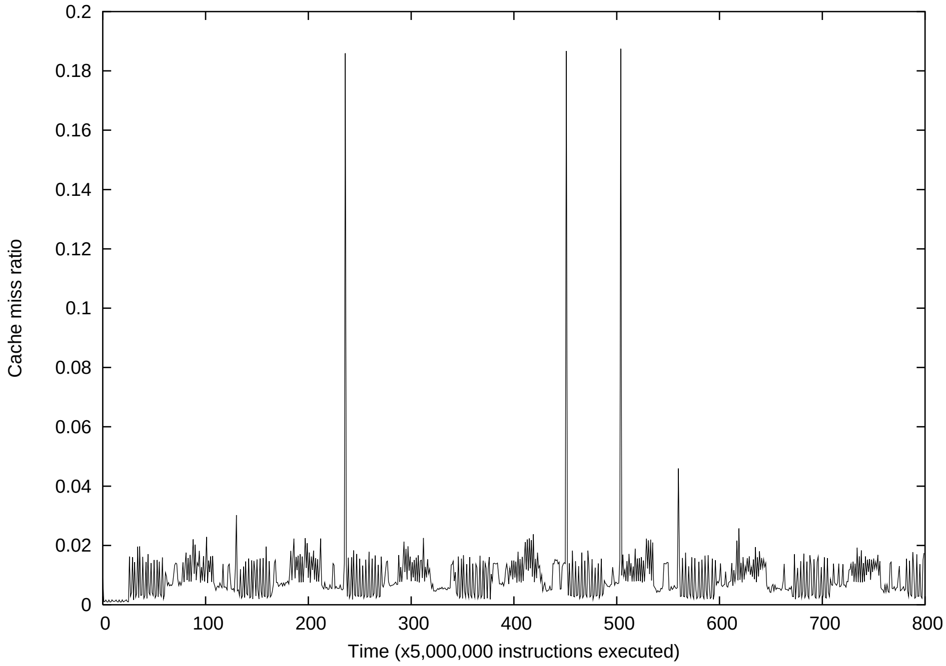

Figure A.5: Data cache miss ratio vs. time for *ijpeg*

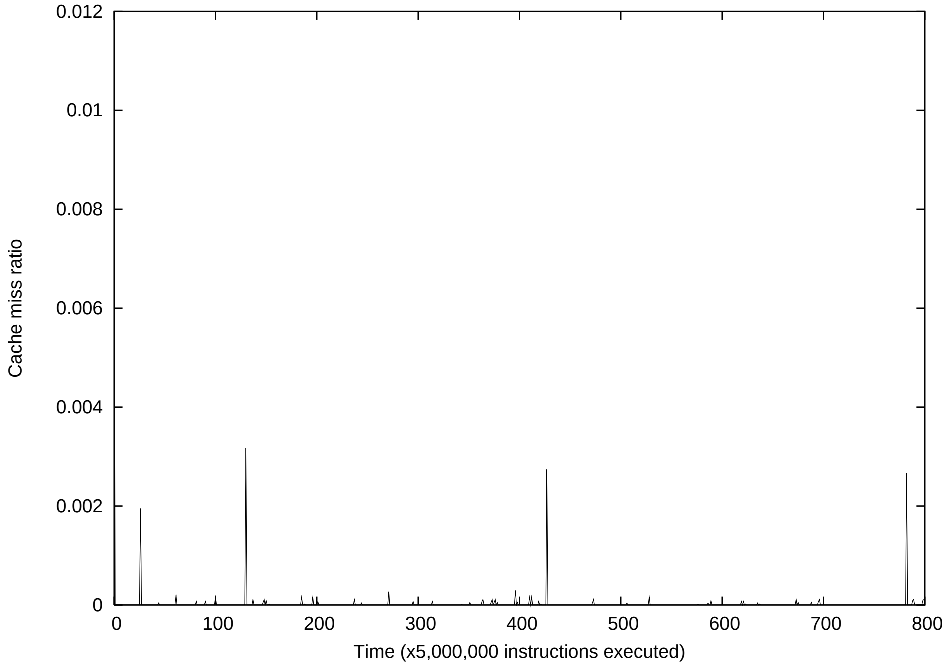

Figure A.6: Instruction cache miss ratio vs. time for *ijpeg*

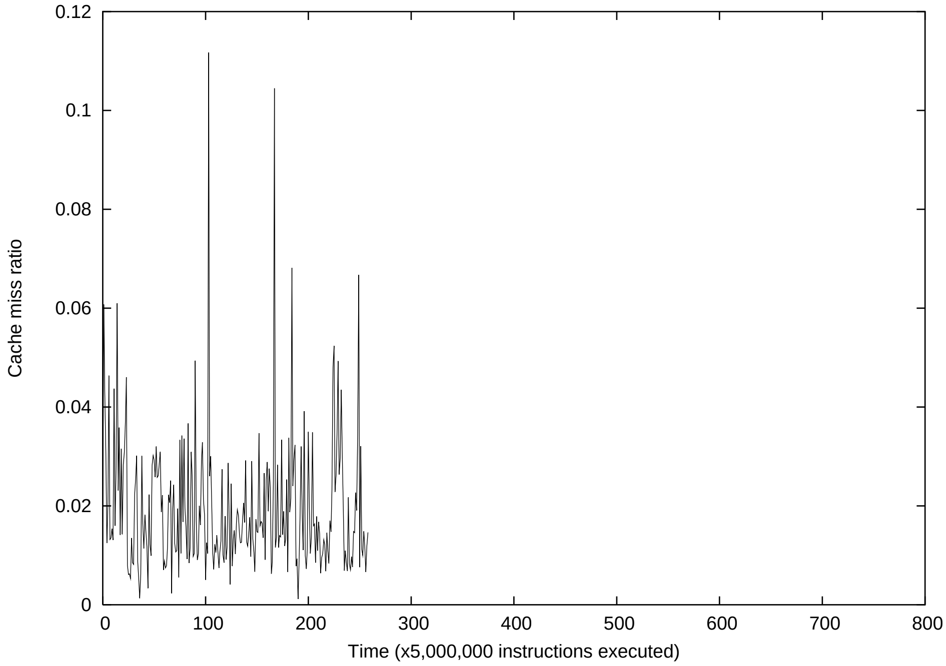

Figure A.7: Data cache miss ratio vs. time for *gcc*

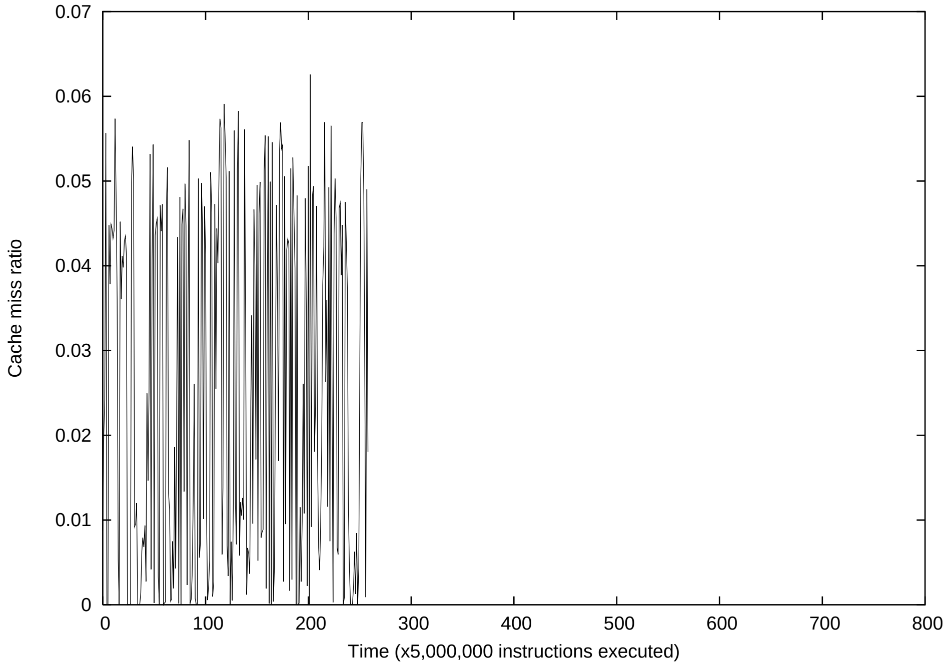

Figure A.8: Instruction cache miss ratio vs. time for *gcc*

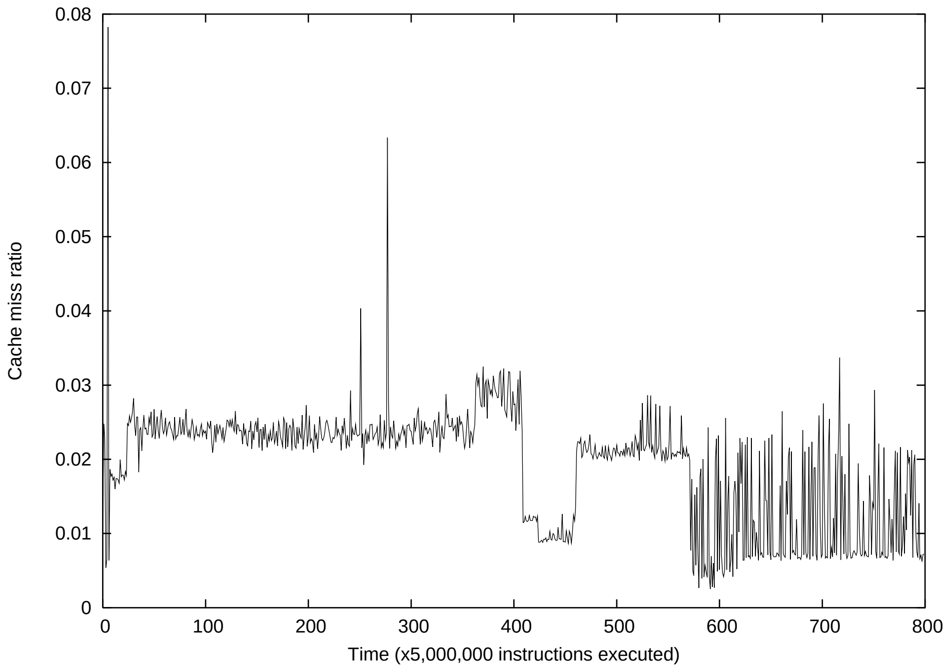

Figure A.9: Data cache miss ratio vs. time for *vortex*

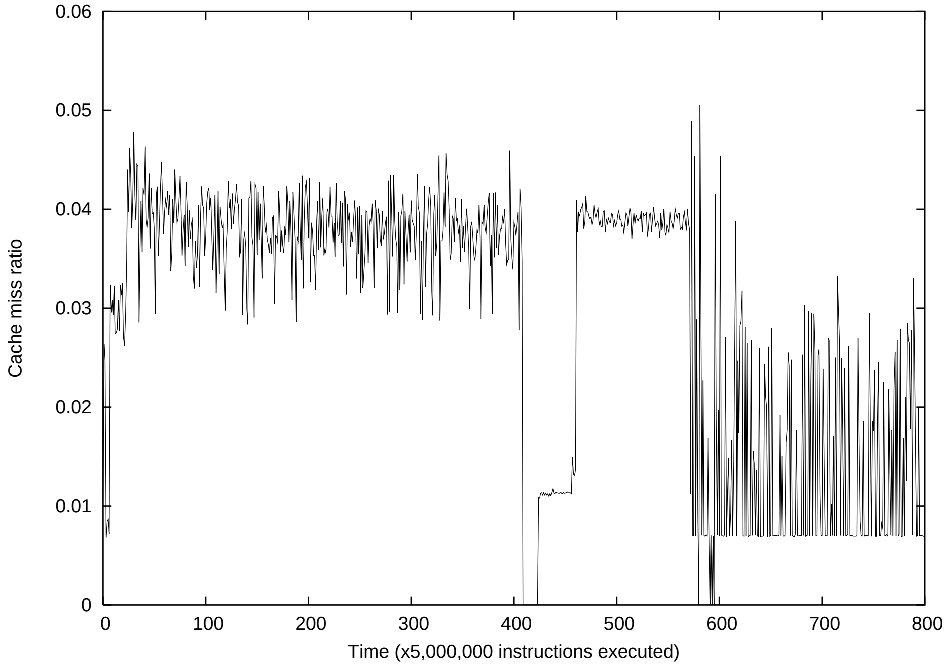

Figure A.10: Instruction cache miss ratio vs. time for *vortex*

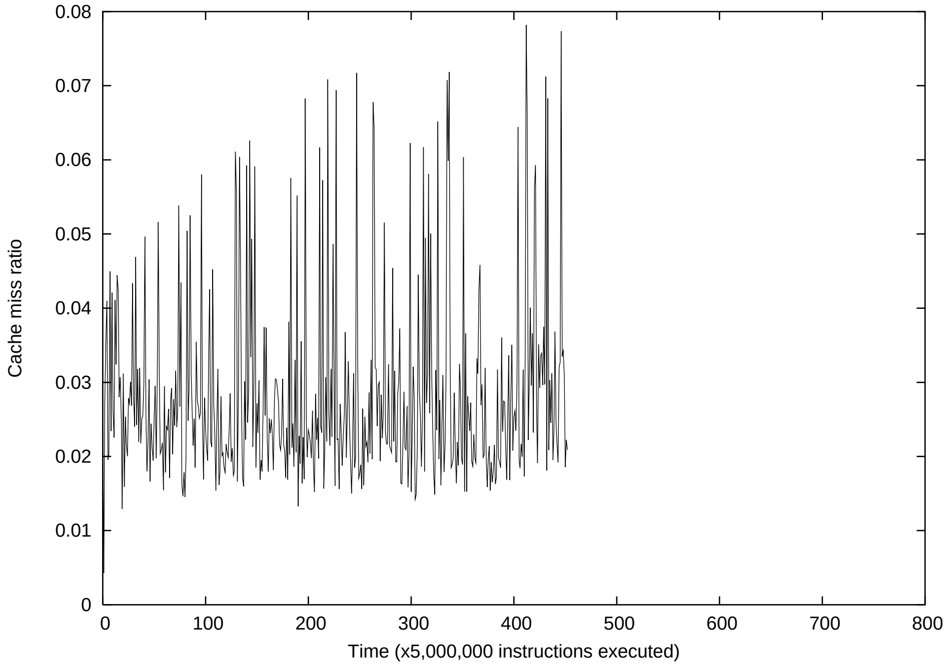

Figure A.11: Data cache miss ratio vs. time for *go*

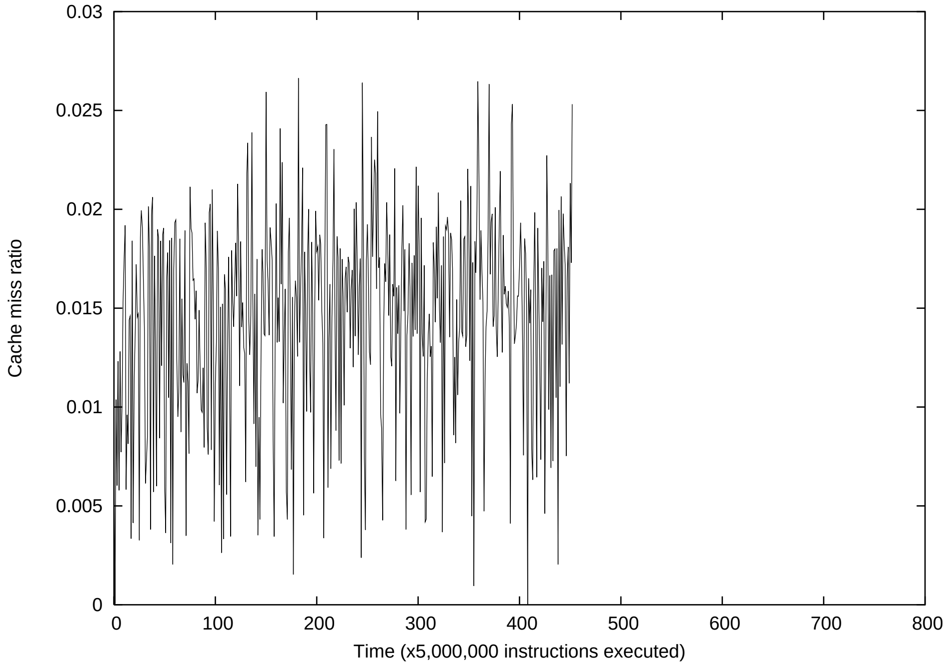

Figure A.12: Instruction cache miss ratio vs. time for *go*

Bibliography

[1] Micron mobile Synchronous DRAM 128Mb: x16, x32. `http://www.micron.com/products/datasheet.jsp?path=/DRAM/Mobile&fileID=5%81`.

[2] *Motorola MC68030 Enhanced 32-bit Microprocessor User's Manual*. Third edition, 1992.

[3] *Intel StrongARM SA-1100 Microprocessor Developer's Manual*. August 1999.

[4] *ARM Architecture Reference Manual*, chapter B3. ARM Limited, June 2000.

[5] Mark Atkins. Performance and the i860 microprocessor. *IEEE Micro*, 11(5):24–27, 72–78, October 1991.

[6] T. Austin, E. Larson, and D. Ernst. SimpleScalar: an infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, February 2002.

[7] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill. Translation lookaside buffer consistency: a software approach. In *Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 113–122. ACM Press, 1989.

[8] M. Cekleov and M. Dubois. Virtual-address caches part 1: Problems and solutions in uniprocessors. *IEEE Micro*, pages 64–71, September 1997.

[9] M. Cekleov and M. Dubois. Virtual-address caches part 2: Multiprocessor issues. *IEEE Micro*, pages 69–74, November 1997.

[10] Chia Chao, Milon Mackey, and Bart Sears. Mach on a virtually addressed cache architecture. Technical Report HPL-90-67, HP Laboratories, June 1990.

[11] D. R. Cheriton, G. A. Slavenburg, and P. D. Boyle. Software-controlled caches in the VMP multiprocessor. In *Proceedings of the 13th annual international symposium on Computer architecture*, pages 366–374. IEEE Computer Society Press, 1986.

[12] Edward H. Frank. The SBus: Sun's high performance system bus for RISC workstations. In *Digest of Papers. Thirty-Fifth IEEE Computer Society International Conference*, pages 184–188, 1990.

[13] Erik Hagersten, Anders Landin, and Seif Haridi. DDM - a cache-only memory architecture. *IEEE Computer*, 25(9):44–54, 1992.

[14] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Software: Practice and Experience*, 28(9):901–928, July 1998.

[15] J. L. Henning. SPEC CPU2000: measuring CPU performance in the New Millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[16] Patrick Hicks, Matthew Walnock, and Robert Michael Owens. Analysis of power consumption in memory hierarchies. In *Proceedings of the 1997 international symposium on Low power electronics and design*, pages 239–242. ACM Press, 1997.

[17] J. Huck and J. Hays. Architectural support for translation table management in large address space machines. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 39–50, May 1993.

[18] B. Jacob and T. Mudge. Uniprocessor Virtual Memory without TLBs. *IEEE Transactions on Computers*, 50(5), May 2001.

[19] Bruce L. Jacob and Trevor N. Mudge. A look at several memory management units, TLB-refill mechanisms, and page table organizations. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 295–306. ACM Press, 1998.

[20] Eric E. Johnson, Jiheng Ha, and M. Baqar Zaidi. Lossless trace compression. *IEEE Transactions on Computers*, 50(2), February 2001.

[21] Toni Juan, Tomas Lang, and Juan J. Navarro. Reducing TLB power requirements. In *Proceedings of the 1997 international symposium on Low power electronics and design*, pages 196–201. ACM Press, 1997.

[22] Gokul B. Kandiraju and Anand Sivasubramaniam. Characterizing the d-TLB behavior of SPEC CPU2000 benchmarks. In *Proceedings of the international conference on Measurement and modeling of computer systems*, pages 129–139. ACM Press, 2002.

[23] Gokul B. Kandiraju and Anand Sivasubramaniam. Going the distance for TLB prefetching: An application-driven study. In *International Symposium on Computer Architecture (ISCA)*, Anchorage, USA, May 2002.

[24] Les Kohn and Neal Margulis. Introducing the Intel i860 64-bit Microprocessor. *IEEE Micro*, 9(4):15–30, August 1989.

[25] Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers. Architectural Support for Single Address Space Operating Systems. Technical Report 92-03-10, University of Washington, July 1992.

[26] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the thirtieth annual IEEE/ACM international symposium on Microarchitecture*, pages 330–335. IEEE Computer Society Press, 1997.

[27] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, February 2002.

[28] Cathy May, Ed Silha, Rick Simpson, and Hank Warren, editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann, 1994.

[29] James Montanaro, Richard T. Witek, Krishna Anne, Andrew J. Black, Elizabeth M. Cooper, Daniel W. Dobberpuhl, Paul M. Donahue, Jim Eno, Gregory W. Hoeppner, David Kruckemyer, Thomas H. Lee, Peter C. M. Lin, Liam Madden, Daniel Murray, Mark H. Pearce, Sribalan Santhanam, Kathryn J. Snyder, Ray Stephany, and Stephen C. Thierauf. A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *IEEE Journal of Solid-State Circuits*, 31(11):1703–1714, November 1996.

[30] Piyush Patel and Diane Douglass. Architectural features of the i860-microprocessor RISC core and on-chip caches. In *Proceedings of the 1989 IEEE International Conference on Computer Design*, pages 385 – 390, October 1989.

[31] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design: The hardware/software interface*, chapter 7. Morgan Kaufmann Publishers, Inc., second edition, 1998.

[32] Xiaogang Qiu and Michel Dubois. Options for dynamic address translation in COMAs. In *Proceedings of the 25th annual international symposium on Computer architecture*, pages 214–225. IEEE Press, 1998.

[33] Glen Reinman and Norman P. Jouppi. CACTI 2.0: An integrated cache timing and power model. Technical Report 2000.7, Compaq Western Research Laboratory, February 2000.

[34] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 285–298. ACM Press, 1995.

[35] Mendel Rosenblum, Stephen Alan Herrod, Emmett Witchel, and Anoop Gupta. Complete computer simulation: The SimOS approach. *IEEE Parallel and Distributed Technology*, 3(4):34–43, 1995.

[36] Adel S. Sedra and Kenneth C. Smith. *Microelectronic Circuits*. Oxford University Press, fourth edition, 1998.

[37] P. Shivakumar and Norman P. Jouppi. CACTI 3.0: An integrated cache timing, power, and area model. Technical Report 2001.2, Compaq Western Research Laboratory, August 2001.

[38] Kevin Skadron, Pritpal S. Ahuja, Margaret Martonosi, and Douglas W. Clark. Branch prediction, instruction-window size, and cache size: performance tradeoffs and simulation techniques. *IEEE Transactions on Computers*, 48(11):1260–1281, November 1999.

[39] Alan C. Skousen, Donald S. Miller, and Ronald G. Feigen. The Sombrero operating system: An operating system for a distributed single very large address space — general introduction. Technical Report TR-96-005, Computer Science and Engineering Department, Arizona State University, Tempe, AZ 85287-5406, April 1996.

[40] Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[41] Per Stenstrom, Truman Joe, and Anoop Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. In *Proceedings of the 19th annual international symposium on Computer architecture*, pages 80–91. ACM Press, 1992.

[42] Madhusudhan Talluri. *Use of superpages and subblocking in the address translation hierarchy*. PhD thesis, University of Wisconsin-Madison, June 1995.

[43] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. Tradeoffs in supporting two page sizes. In *Proceedings of the 19th annual international symposium on Computer architecture*, pages 415–424. ACM Press, 1992.

[44] Andrew S. Tanenbaum. *Structured Computer Organization*, chapter 2. Prentice-Hall, Inc., fourth edition, 1999.

[45] George Taylor, Peter Davies, and Michael Farmwald. The TLB slice - a low-cost high-speed address translation mechanism. In *Proceedings of the 17th annual international symposium on Computer Architecture*, pages 355–363. ACM Press, 1990.

[46] Patricia J. Teller. Translation-lookaside buffer consistency. *IEEE Computer*, 23(6):26–36, June 1990.

[47] Leon Tietz. Virtual memory within embedded systems - marketing hype or engineering reality. *Intel PCA Developer Network Handheld and Wireless Solutions Journal*, 2001.

[48] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys*, 29(2):128–170, 1997.

[49] Tim Wilkinson, Tom Stiemerling, Peter E. Osmon, Ashley Saulsbury, and Paul Kelly. Angel: A proposed multiprocessor operating system kernel. In *European Workshop on Parallel Computing*, pages 316–319, Barcelona, Spain, 1992.

[50] Steven J. E. Wilton and Norman P. Jouppi. CACTI: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, May 1996.

[51] D. A. Wood. *The Design and Evaluation of In-Cache Address Translation*. PhD thesis, Department of Computer Science, University of California, Berkeley, February 1990.

[52] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, J. M. Pendleton, S. A. Ritchie, G. S. Taylor, R. H. Katz, and D. A. Patterson. An in-cache address translation mechanism. In *Proceedings of the 13th annual international symposium on Computer architecture*, 1986.

[53] D.A. Wood and R. H. Katz. Supporting Reference And Dirty Bits In SPUR's Virtual Address Cache. In *Proceedings of the 16th annual international symposium on Computer Architecture*, pages 122–130, 1989.

[54] Patryk Zadarnowski. The design and implementation of an extendible instruction-set simulator. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 2000.

[55] Michael Zhang and Krste Asanovic. Highly-associative caches for low-power processors. In *Kool Chips Workshop, MICRO-33, Monterey, CA*, December 2000.