1. Introduction

The idea of introducing a cache between a CPU and its main memory store was first invented in 1946 by von Neumann et. al., in their preliminary discussion on the logical design of an electronic computing instrument. Computer caches address the problem of the increasingly large discrepancy between the rate at which a CPU requires new data and the rate at which the main memory store can provide data. The fundamental reason for using a computer cache is summarised in the following quote: "Ideally one would desire an indefinitely large memory capacity such that any particular word would be immediately available … we are forced to recognise the possibility of constructing a hierarchy of memories, each of which has a greater capacity than the preceding but which is less quickly accessible" (von Neumann et. al.).

This fourth year computer science project has revolved around two main components: the Hierarchical computer Architecture design and Simulation Environment (HASE), and Dinero, a cache simulation tool, each of which is briefly described below.

Dinero is a command-line based trace-driven cache simulator. It provides a powerful simulation environment in which accurate cache performance statistics can be quickly produced. Cache simulation has been an area of much research in recent years, primarily due to the increasing effect that cache efficiency has on CPU performance as the relative gap between CPU bandwidth requirements and available main memory bandwidth widens. The increasing importance of computer caches is demonstrated by the fact that in 1980 most computers had no cache, whereas in 1995 computers often have two levels of cache [10]. In response to the increasing importance of efficient caches, over 1600 computer cache research papers have been released.

HASE is a C++ based simulation environment that provides two main benefits to a computer architecture designer. Firstly, HASE provides an in-built discrete simulation mechanism that can be easily used to describe hardware components and their interconnection. Secondly, HASE allows the movement of data through a computer architecture to be visualised by an animation of the design window [3]. There are a number of alternative simulation environments available, each of which offers a range of benefits and weaknesses when compared to HASE, for example:

1. **The Cadence Virtual Component Co-Design (VCC) tool.**
   VCC provides a graphical computer architecture design environment in which the emphasis is on producing realistic performance characteristics for embedded systems, as opposed to the emphasis on design validation offered by HASE [15]. VCC is very similar to HASE, with the main difference being that HASE does not restrict C++ language usage in any way, unlike VCC where a sub-set of C++ must be used, also HASE allows any C++ compatible library files to be used, whereas VCC designers are restricted to several pre-designed libraries.

2. **The SimOS simulator.**
   SimOS differs substantially from HASE in that it provides an environment capable of simulating the hardware of a computer system in enough detail to boot a commercial operating system and run realistic workloads on top of it [14].

3. **Standard programming languages.**
   For example: the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL), Verilog (a hardware description language) and C++. These programming languages can all be used to produce simulations of a similar standard to those that can be produced using HASE. However, as high-level simulations are usually written in either C or C++, HASE, which is based on C++, offers better interoperability that either of the above hardware description languages [11]. C++ could be used to write both a simulation environment and simulation directly, but the main benefit of HASE is that a powerful simulation library and animation system are already provided for the C++ programmer.

This project was initially entitled "Cache Simulations," for which the primary goal was to create a set of HASE simulation models of cache systems capable of running Dinero traces and to assess their performance, with the potential for the implementation of additional cache models in HASE if time constraints allowed. However, during the course of project implementation it became clear that the main benefit to be gained by implementing Dinero in HASE is that the animated visualisation of cache activity can assist the learning process for cache design students. As a consequence of this recognition, the project goal was revised to specify the development of an animated cache simulation tool, based on Dinero, which would allow cache design students to both learn how computer caches operate and to compare the performance of arbitrary cache configurations. The revised project goal was successfully attained.

The main methodologies and techniques employed during the course of the project are listed below.

1. The use of HASE to implement a trace-driven cache simulation.
2. The Human Computer Interaction (HCI) based design and evaluation of the simulator user interface.
3. Validation of the correctness of the cache simulator produced, using standard software testing techniques.

The remainder of this report is divided into five main chapters, each of which is briefly introduced below.

- **Chapter 2: An Overview of HASE**
  This chapter provides an introduction to the HASE simulation environment, describing both the operation of HASE and a typical computer architecture design methodology.
- **Chapter 3: An Overview of Dinero**
  Dinero is compared to alternative types of cache simulator and the main types of cache feature that can be simulated are described.
- **Chapter 4: The HASE Implementation of Dinero**
  This chapter contains the majority of information relevant to the implementation of Dinero in HASE, focusing in particular on the design decisions made together with appropriate justifications.
- **Chapter 5: The HASE Dinero User Interface**
  Detailed information is provided on the design and implementation of the HASE Dinero user interface, together with a description of the design and execution of an HCI-based evaluation of the user interface.
- **Chapter 6: Validation of the HASE Version of Dinero**
  A description is provided of the testing methodology used to evaluate the quality of the software produced during the course of the project.

2. An Overview of HASE

2.1 Introduction

This section of the report provides an insight into the Hierarchical computer Architecture design and Simulation Environment (HASE) used to implement Dinero.

Firstly I give a brief guide to the background of HASE followed by an explanation of the main benefits of HASE and details of the use and internal operation of the simulation environment. The section is concluded with an overview of the stages in a typical design methodology, as used in this project, to produce an architecture simulation in HASE.

2.2 Background

Initial development of the current version of HASE began at Edinburgh University in 1992. Several research papers are available that provide details on both the operation of HASE and its application to computer architecture design problems [2, 3, 5, 12].

HASE addresses the fourth of the "Grand Challenge Problems in Computer Architecture" identified at the 1992 Purdue Workshop [16], namely "to develop sufficient infrastructure to allow rapid prototyping of hardware ideas and the associated software in a way that permits realistic evaluation." The main goal of the HASE project is to provide computer architects with a set of tools that allow the rapid development and exploration of computer systems [3].

2.3 Features

HASE incorporates several powerful features to facilitate the rapid development and exploration of computer systems:

- A simulation can be hierarchically structured to reflect varying levels of architectural or problem abstraction. This enables the typical architecture refinement methodology whereby initial high-level behavioural descriptions of the architecture are broken down into smaller components, leaving a structural view for the high-level architecture. This approach also allows different parts of an architectural design to be simulated at different levels of detail, on a per-run basis.

- The visual verification of a model's behaviour via an animation of the simulation design window. This is a very powerful feature of HASE, and the main factor that distinguishes it from the majority of alternative simulation environments. The HASE designer can view, at any desired level of detail, the flow of data through an architecture thereby facilitating both swift debugging and easy verification of correctness.

- Provision of simulation templates and component reuse that provide ideal mechanisms for rapid prototyping and architecture exploration. The behavioural descriptions of HASE architectural components are written in a language composed of a superset of C++. Due to the object-oriented nature of C++ and the standard interface provided by the HASE simulator it is possible to reuse parts of existing simulations when creating a new design. This can assist in substantially reducing the required development time.

2.4 The Simulation Environment

HASE uses a discrete event simulation engine called HASE++, which provides a simulation language formed from a superset of C++. The basic HASE architectural unit is an entity. During simulation each entity is modelled by the simulator as a separate lightweight thread, synchronised by a central class that maintains the future and deferred event queues for each thread. The simulation continues until the future event queues for each thread are empty.

During a simulation run HASE writes out simulation events to a trace file. The volume of data written to the trace file can be restricted by choosing a trace level that specifies which types of event should be recorded, for example: state changes, message passing and array updates.

HASE provides an animation tool that allows the trace file produced during a simulation run to be loaded into memory and played back via an animation of the design window. The animation tool provides all of the facilities required to precisely control the playback of the trace file, including: pause, single-step, jump to specific location and variable playback speed.

The high-level architectural view of HASE simulations is specified by writing two files, unique to each HASE simulation created. The Entity Description Language (EDL) file specifies four main types of architectural description:

1. **Entities** are the basic architectural components supported by HASE, and can range in detail from low-level to high-level. For example, an entity could represent a CPU or a single register. HASE allows entities to be pooled under a larger entity, the **compentity**, which provides a higher level of abstraction in terms of both simulation detail and animation of the display during trace file playback. Each project written in HASE has a number of entities associated with it, found in the **entity library** section of the EDL file.

2. Each entity and compentity has a number of parameters associated with it. **Ports** specify the input and output capability of an entity, with each port being restricted to a particular link type and any number of ports per entity allowable. HASE **parameters** can also be associated with an entity thus providing a useful means of feedback during animation since the dynamic value of each parameter can be displayed as text on the screen. It is also possible to associate parameters of the enum type with a number of different picture files, thereby allowing the state of an entity to be visually displayed during trace file playback.

3. The EDL file provides a **parameter types** section for the declaration of **data types** and **link types**. Data types allow the HASE designer to create custom types for entity parameters, constructed from the basic HASE-allowed data types (for example: integer, float, string, array and enum). Link declarations allow the types of data supported by a link to be specified.

4. The final part of the EDL file is the **architecture layout** section, which both provides a way to instantiate entities in a project, and allows the connection of links between entities to be specified.

The second file required to specify a HASE project design is the Entity Layout File (ELF). The ELF describes the arrangement on-screen of entities and links specified in the architecture layout section of the EDL file. The ELF also allows the designer to specify both whether parameters associated with entities should be displayed on-screen and how they should be displayed.

Since 1997 HASE has incorporated five modes of operation, to clearly distinguish the different stages of the architecture design and evaluation cycle [3]:

1. **Design**. In this mode the number of components in the project and their layout and connection can be modified via the HASE GUI.

2. **Validate**. This mode has not yet been implemented in HASE.

3. **Build**. This mode allows the designer to instruct HASE to compile and link the HASE++ code associated with each design entity.

4. **Simulate**. Facilities are provided to instruct HASE to run the simulation, load the subsequently produced trace file back into memory, and to animate the display based on the trace file contents.

5. **Experiment**. This mode provides the capability of running the simulation multiple times, varying certain of the input parameters at each simulation iteration.

There are currently two versions of HASE in use: one written for UNIX, and the other for Windows NT. The Windows NT version of the simulator was written at a later date than the UNIX version and provides more advanced facilities, including in particular **dynamic arrays**. Dynamic arrays allow the size of a HASE array to be changed at each simulation iteration, without the need for recompilation, based on a HASE declared integer parameter. NT HASE also provides in-built support for integers of unlimited size (within reason) based on the LEDA large integer library.

2.5 Creating a HASE Simulation

1. **High-Level Architecture Design**
   In this stage the HASE designer will decide on the basic architectural form of the simulation. This will consist of deciding the level of abstraction to be used for the different parts of the architecture and then describing the architecture as a number of entities in the HASE EDL file. Concerns at this stage will typically be decisions such as: “Should a register block be modelled as a single unit or should each register be a separate entity.” The answer to such questions will depend on a combination of factors, including the level of detail required in the simulation, the resources available to run the simulation, and the observability trade-off between abstraction and detail.

2. **Data-Flow Design**
   Once the basic design of the architecture has been decided, the designer can specify in detail the flow of data through the architecture. This will include specifying, in the HASE EDL file, the types of link to be supported by the project and the actual links between instantiated entities. A high-level specification for the data processing to occur in each entity will also be produced at this stage.

3. **Implementation of Architecture Behaviour**
   This is the final stage involved in implementing the simulation, consisting of producing a HASE++ description for each of the entities created in stage one based on the data-flow description produced in stage two.

It should be stressed that the above is an idealistic model, and in this project iteration between stages was frequently required as a consequence of unexpected implementation problems being encountered during stage three.

3. An Overview of Dinero

3.1 Introduction

This section of the report describes the different classes of cache simulator that are available then goes on to explain the main cache parameters that can be modified and simulated by Dinero. The section concludes with a brief description of the problems associated with using Dinero as a teaching tool.

3.2 What Type of Cache Simulator is Dinero?

There are four main types of cache simulator in common use, each of which has many published papers describing benefits and use [1, 19, 20]:

- **Trace Driven**
  Trace driven cache simulators require a trace file consisting of a large number of two-tuples of address and access type, written out to file during the course of an application's execution. The trace file is used by a cache simulator to determine the effectiveness of a particular cache configuration for the application modelled in the trace file. The main limitations of trace driven cache simulation are as follows: firstly, the trace file used in a simulation run only provides test data for one specific application, i.e. the effects of a multi-tasking environment and the side-effects of the operating system are ignored. Secondly, the storage space required for the trace file can be a limiting factor as millions of reference addresses must be stored. This problem becomes particularly apparent if the size of cache being simulated is large (mega-bytes in size) as the size of trace file required to usefully test a cache's effectiveness linearly scales with the size of cache being simulated.

- **Trap Driven**
  In contrast to the trace driven type of simulation, trap driven simulations operate in 'real-time'. Instead of writing memory references out to a file, the performance of a cache being simulated is dynamically evaluated for each memory reference generated. This is achieved via operating system support, with a trap being generated for each memory access. Each trap is immediately handled by the cache simulator code before control is returned to the operating system. The main benefit of this approach is that the cache simulation is not limited to the trace references of just one application as the memory references generated by the entire system are captured. Hence effects such as multi-tasking and the operating system overhead are taken into account in determining cache performance. The main drawbacks to trap driven simulation are: specific operating system support is required, each simulation run will be different, and the system slowdown caused by the simulator will bias the simulation results [20].

- **Analytical Cache Models**
  The analytical cache model method of cache simulation is a variant of trace driven simulation which addresses the problem that the memory and processing requirements for trace driven simulation can be expensive. The suggested solution is to process each trace file to produce a much reduced set of trace file statistics. The trace file statistics are heuristically generated so that an estimate of the performance of particular cache configurations with the actual trace file can be produced by the cache simulator. The main problem with this approach is that the method used to process the initial trace file is problematic to justify and not guaranteed to produce results of a satisfactory accuracy. Additionally, this type of simulation has the same problem as trace driven simulation in that the cache performance for only one application is taken into account, with no allowance for multi-tasking and the operating system overhead.

- **Hardware Measurement**
  This technique requires a hardware test-bed to be produced for a cache configuration that has been implemented in hardware. The test-bed captures the dynamic performance of the cache during typical execution, with the main benefit being that the results produced are guaranteed to be 100% accurate for the particular cache being measured. The unfortunate side effect is that each cache configuration to be evaluated requires a hardware implementation and test-bed. Consequently this is a particularly expensive method of cache simulation.

Dinero is a cache simulator of the trace driven variety. The main benefits of this type of simulation over the alternatives are:

1. Simulations are repeatable and so the effects of varying cache parameters can be isolated.

2. It is an inexpensive method of cache simulation.

3. Simulation results can be obtained in many situations where analytic model solutions are intractable without questionable simplifying assumptions.

4. A trace-driven simulation is guaranteed to be representative of at least one program in execution.

3.3 Cache Configurations Supported by Dinero

The fundamentals of cache design and operation are well codified with several published works providing an explanation of the main types of cache parameter [9, 10, 17].

Dinero is a reasonably complete cache simulator, in that all of the essential cache parameters are modelled, and can be adjusted, and several advanced facilities are provided.

The following annotated diagram (figure 3.1) shows a basic overview of a typical direct-mapped cache configuration. The data read process starts with the CPU sending an address to the cache (1). The cache splits this address up into three components (2). The **block offset** component is used to index into a cache line, since each cache line can contain more than one CPU word. The **index** component is used to select a particular line in the cache. Hence the range of the index field, in binary, matches the number of lines in the cache (this is a simplification, as will become clear by the end of this section). The **tag** component is composed of whatever is left of the CPU address

after the block offset and index components have been removed. The tag is used to distinguish between items in the set of potential main memory data, with the same index field, that can be stored in the same cache line.

Lastly, the cache examines the valid bit and tag entry for the cache line specified by the index field of the CPU address (3). If the valid bit is set, and the tag entry from the cache matches the tag in the CPU address, then the data stored in the appropriate cache line is indexed with the block offset component of the CPU address (4) and the requested data sent to the CPU. If the required data is not in the cache then the required data will be loaded to the cache and CPU from the lower-level memory.

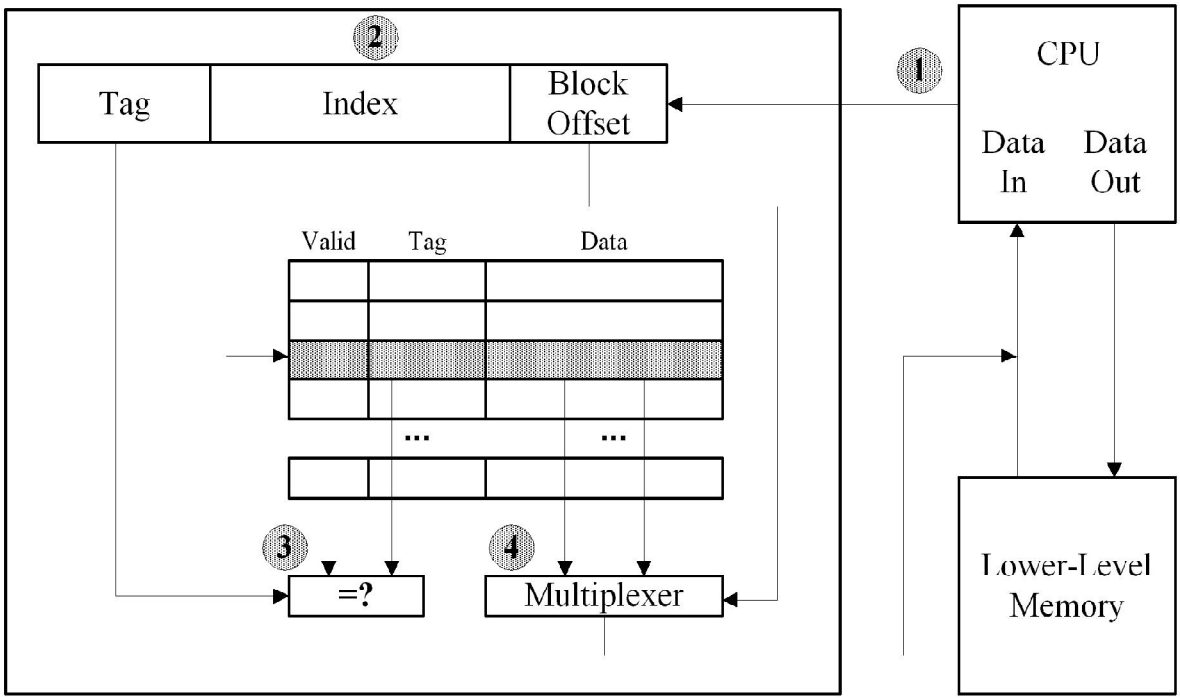


Figure 3.1: Typical Cache Configuration and Operation on a CPU Read

The following section lists and briefly describes each of the cache parameters that can be simulated and adjusted using Dinero:

1. **Variable Cache Size**
   This is naturally the most fundamental cache design parameter as it determines the amount of data that can be stored in the cache.

2. **Variable Block Size**
   Each cache line can store a certain number of CPU words. The number of CPU words per cache line is referred to as the block size. The block size determines the atomic unit of cache to main memory data transfers. Greater block size results in increased chances of a cache hit due to the principle of locality [17] (over short periods of time, a program distributes its memory references non-uniformly over its address space, and which portions of the address space are favoured remains largely the same for long periods of time) but results in higher latency when loading a block from main memory.

3. Unified or (Data and Instruction) Caches

A typical microprocessor requires two types of information to be provided in the normal course of operation. Firstly, it may need to load the instruction for the next step in the program being executed. Secondly, the instruction type may require one or more items of data to be loaded. The most important benefit of splitting the cache from unified to data and instruction is that the different types of data access do not conflict with each other by mapping to the same cache line. The other advantages of splitting the unified cache into separate data and instruction caches are, firstly, that the CPU can access both caches simultaneously. Secondly, each separate cache can have a storage capacity appropriately scaled to the different throughput requirements of the information type being stored.

4. Variable Associativity

Associativity allows the cache to be divided into discrete subsets of cache lines called **associative sets**. Each associative set is allocated a unique index that determines which subset of memory addresses are eligible to be stored in the set. There are three classes of associativity, described below:

- **Fully Associative**
  The entire cache is regarded as just one subset. Hence the index field size is zero, and any memory address can be stored at any line in the cache. Consequently the tag of every line in the cache must be checked to determine whether a cache hit has occurred at each lookup. This lookup operation occurs in parallel in hardware, but is a time-consuming process in a software simulation.

- **Set-Associative**
  The cache is divided into a number of discrete subsets, each of size *associativity* cache lines. All memory addresses sharing the same index can be stored at any line in the associative set. The index field of the CPU address is large enough to address each unique associative set within the cache. Consequently the tags of each line within a particular associative set must be checked to determine whether a cache hit has occurred at each lookup.

- **Direct Mapped**
  Each cache line is its own associative set. Hence, for a particular CPU address, only one cache line need be checked to determine whether a cache hit has occurred. The index field of the CPU address is large enough to address every line in the cache.

The larger the associative set size, the less likely it is that the problematic situation of a number of frequently used blocks being mapped to an associative set too small to contain all of the blocks will occur. If this situation does occur then the performance of the cache will be degraded as blocks in the frequently used set will need to be constantly re-loaded from main memory. As a consequence, cache designs with larger associative sets usually offer better performance.

5. Cache Replacement Policy

The cache replacement policy determines how a particular cache line is chosen to be overwritten when a new block is loaded to the cache and all of the cache lines it could be placed in are already in use. Dinero allows three different replacement policies:

- **Least Recently Used**
  The Least Recently Used (LRU) algorithm requires the cache to keep a record of the order of accesses to each cache line within each associative set. A typical method used (in a cache simulator, not hardware!) is to assign a timestamp to a cache line each time that it is read / written. When a cache line must be selected for overwriting, the LRU line in the appropriate associative set is chosen.

- **First-In, First-Out**
  The First-In, First-Out (FIFO) method of cache-line replacement is simpler than that of LRU. Each associative set is treated as a circular buffer with new cache lines entering at one end and old lines leaving at the other. This is typically implemented by simply maintaining a pointer to the next cache line to be replaced within each associative set, and so the maintenance overhead for this policy is very low.

- **Random**
  This is the simplest replacement policy to simulate in software (although FIFO is the simplest replacement policy to implement in hardware). When a cache line must be selected for replacement, it is chosen randomly from the appropriate associative set.

6. Cache Write Policy

The cache write policy determines what action the cache should take on a CPU write. Two options are possible:

- **Write Through**
  If a cache hit occurs for a CPU write command then the modified data is written to both the cache and to main memory. However, if the CPU write causes a cache miss then, whilst the modified data is always written to main memory, whether the modified data is written to the cache is determined by the write allocation policy. This removes the complexity of keeping track of which lines have been updated in the cache, as all CPU writes propagate immediately to main memory, but increases the latency for a CPU write unless a main memory write buffer is used.

- **Copy Back**
  CPU writes do not immediately propagate to main memory, instead the modified data is stored in the cache and a modified bit set. If a modified cache line is chosen for replacement, or a cache flush occurs, the modified data must be written to main memory.

7. Write Allocation Policy

This policy determines the action to be taken when a CPU write causes a cache miss, in terms of the additional part of the cache block containing the data word written by the CPU:

- **Allocate**
  The block being written to is loaded from main memory to the cache. If the cache write policy is set to write-through, then the modified data from the CPU is written to the cache (and main memory).

- **No Allocate**
  The block being written to is not loaded from main memory. If the cache write policy is set to write-through, then the modified data from the CPU is not written to the cache.

Note that the cache write and write allocation policies are closely related. All combinations of the two policies are valid except for copy back and no allocate, which would result in invalid blocks being stored in the cache and later written to main memory.

8. Sub-Block Placement and Fetching

Dinero allows the unit of transfer between main memory and cache to be reduced from the normal *block* size to a size of *sub-block*. The sub-blocks allow the cache to contain blocks of which only certain sub-blocks are valid. The benefit of sub-blocks is that the cache storage requirements for the tag field of each cache line can be kept small, due to the relatively large block size, whilst allowing a smaller unit of transfer between cache and memory (and hence reduced latency / bus width requirements).

9. Hardware Prefetching

Dinero allows the simulation of hardware prefetching. Prefetching in Dinero is very straightforward, with only one (sub-) block being fetched at each prefetch, and the (sub-) block to prefetch being determined by a fixed prefetch distance. A (sub-) block loaded by a prefetch command is stored in a separate single-item buffer. There are five prefetching schemes available:

- **Demand Fetch**
  No prefetching take place.

- **Always Prefetch**
  A prefetch operation takes place after every read request from the CPU.

- **Miss Prefetch**
  A prefetch operation takes place after every read request from the CPU that results in a cache miss.

- **Load Forward Prefetch**
  This policy only works with sub-block placement and works just like always prefetch, except that no attempt is made to prefetch sub-blocks outside of the current block.

- **Sub Block Prefetch**
  This policy only works with sub-block placement and works just like always prefetch, except that attempts to prefetch sub-blocks outside of the current block are instead converted to prefetches within the current block by a simple wrapping method.

3.4 Problems with Dinero

Whilst Dinero is a very quick cache simulator, it is not well suited for use as a teaching tool. Dinero has a command line based interface and consequently the set of commands that need to be learnt in order to use the program cause usability problems for people new to the simulator. Additionally, the results from Dinero are displayed in a textual format that is not particularly clear and hard to justify, given the advanced graphical methods available today. Figure 3.2 shows the Dinero results output format, from which the clarity problems of the simulator should be clear.

```
---Dinero III by Mark D. Hill.
---Version 3.3, Released May 1989.
---Solaris version by Nigel Topham, Released January 1995.

CMDLINE: dinero -b32 -u4096 -a1 -rl -wc -Aw
CACHE (bytes): blocksize=32, sub-blocksize=0, Usize=4096, Dsize=0, Isize=0.
POLICIES: assoc=1-way, replacement=l, fetch=d(1,0), write=c, allocate=w.
CTRL: debug=0, output=0, skipcount=0, maxcount=10000000, Q=0.

 Metrics                Access Type:
 (totals,fraction)      Total   Instrn   Data    Read   Write    Misc
 -----------------      ------  ------  ------  ------  ------  ------
 Demand Fetches         1000002 757341  242661  159631   83030       0
                        1.0000  0.7573  0.2427  0.1596  0.0830  0.0000

 Demand Misses          104630   66844   37786   27139   10647       0
                        0.1046  0.0883  0.1557  0.1700  0.1282  0.0000

 Words From Memory      837040
 ( / Demand Fetches)    0.8370
 Words Copied-Back      120784
 ( / Demand Writes)     1.4547
 Total Traffic (words)  957824
 ( / Demand Fetches)    0.9578
```

Figure 3.2: The Dinero Results Output Format

Dinero is ideally suited to an experienced cache designer who wants quick simulation results, but the combination of both the lexical complexity of the Dinero interface and the lack of a cache design teaching facility result in Dinero not being suited to beginners.

4. The HASE Implementation of Dinero

4.1 Overview

This chapter of the report describes both the design issues encountered and implementation decisions made during the creation of the HASE version of Dinero. The chapter is divided into three main sections, which reflects both the different types of task undertaken and the ordering of work in the project:

- **4.2 Basic Dinero Features in HASE**
- **4.3 Advanced Dinero Features in HASE**
- **4.4 Additional Features in HASE Version of Dinero**

4.2 Basic Dinero Features in HASE

This section provides the greater part of information in the report regarding the specific details of the HASE Dinero implementation. The section begins by describing the architectural and data-flow design issues encountered and their solutions. The section concludes with details of both the implementation problems encountered and techniques employed in creating the basic features of Dinero in HASE.

4.2.1 Design of the Animated HASE Dinero Architecture

The first design issue I encountered was deciding how to represent the Dinero cache architecture as an interconnection of entities in the HASE environment. The main issue with designing a suitable architecture was that Dinero allows many different types of cache to be simulated, many of which require different hardware implementations. Unfortunately, HASE allows only one architecture design per project. As a consequence, if I chose to use a unique HASE architecture for each of the different types of cache architecture then many different completely separate HASE projects would be needed. I decided that this was not a suitable solution due to both the large number of architectures that would be required and the additional confusion this would add to the teaching environment.

The main variable parameter of Dinero that requires different architectural designs is *associativity*. Depending on which of the three levels of associativity is simulated (fully associative, set associative or direct mapped), a different mechanism is required in the cache to determine whether a cache hit has occurred. If the cache is modelled at the sub-component level as register banks, multiplexers and control logic then the layout and interconnection of these components will drastically change between different associativity classes.

There are other Dinero parameters, apart from associativity, that result in different architectural configurations if the cache is modelled at a relatively low level of detail, principally the block size, sub-block size and replacement, write allocation and prefetching policies. However, the architectural effect caused by these parameters is less pronounced than that of associativity.

In addition to the design issue of how the HASE Dinero architecture should reflect the cache configuration in use, there was the additional concern that there are as many different ways to construct a cache as there are cache designers. As HASE Dinero was designed to be a teaching tool it was a critical concern that an architecture be chosen that was easy to understand but still clearly displayed the activity occurring within the cache, even though different cache configurations would share the same cache architecture.

It was apparent that the cache could not be modelled as a series of multiplexers, registers and control logic as any particular architectural configuration chosen would not fit all of the cache parameters simulatable. As a consequence, and in line with my goal of simplifying the simulation environment as far as possible, I designed a cache architecture consisting of just a control unit and memory block (figure 4.1). The conceptual operation of the cache is very straightforward.

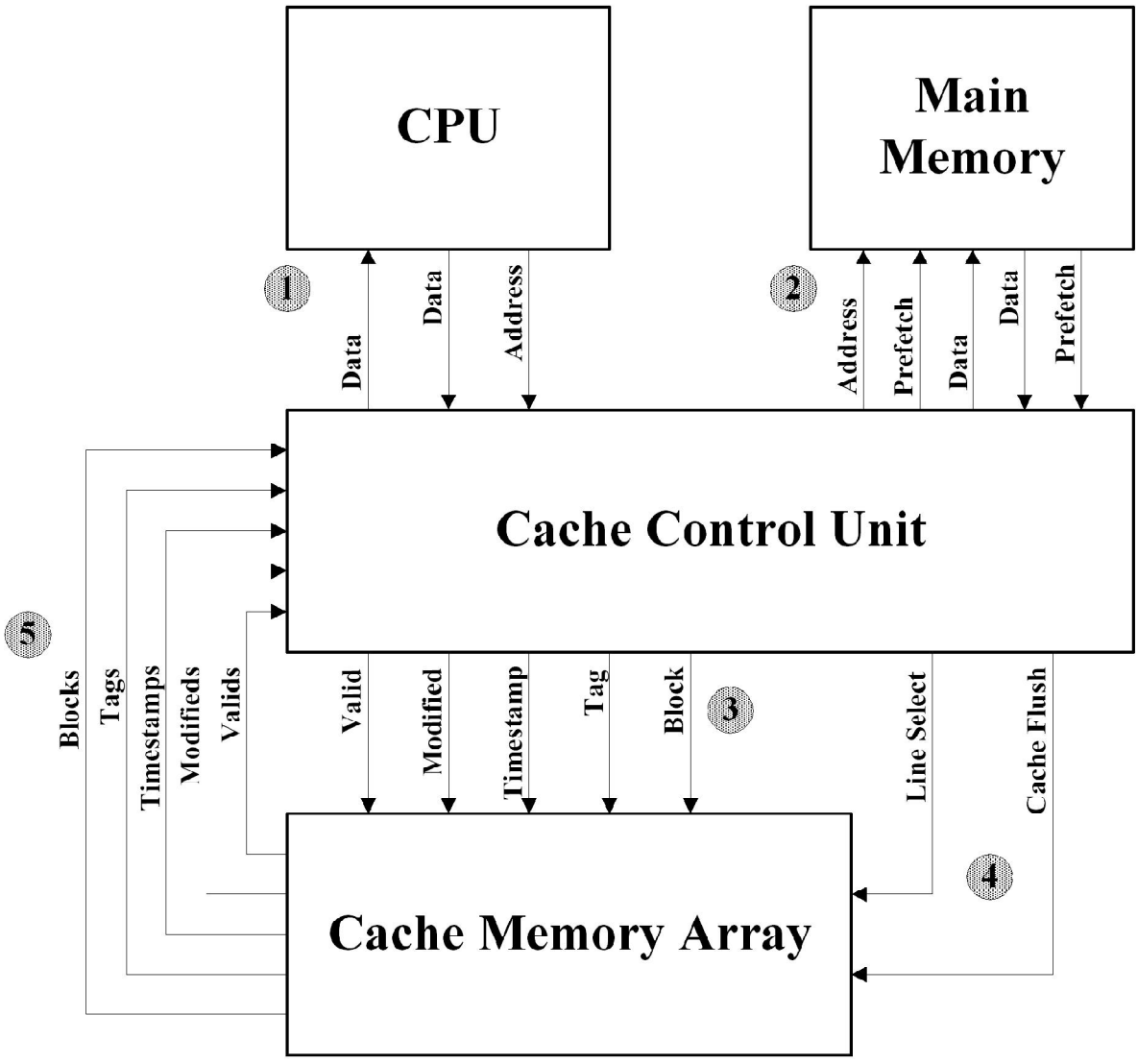


Figure 4.1: The HASE Dinero Memory Hierarchy and Cache Architecture

There are three links from the cache to the CPU (1) which allow CPU words to be sent in both directions and address values to be sent from the CPU to the cache. The five links between the cache and main memory (2) are used for: sending prefetch address and standard address informa-

tion from the cache to main memory, communication of cache blocks in both directions, and for the transmission of prefetch blocks from main memory to the cache.

The first five connections from the cache control unit to the cache memory array (3) are used for sending a valid bit, modified bit, timestamp, tag and block, respectively, from the control unit to the memory array. This provides a mechanism for the control unit to write data to the memory array. The last two links from the cache control unit to the memory array (4) are the line select link and the cache flush link. The five links from the cache memory array to the cache control unit (5) allow the contents of associative sets to be sent to the control unit. Each link corresponds to one of: valid bits, modified bits, timestamps, tags and blocks.

Whilst this design is extremely minimalist and an over-simplification, it does give a good idea of the three principal components of any cache: the control logic, the storage area and the interconnection network. Furthermore, it satisfies my main design goal of providing an architectural design that can be used as a representative for any of the cache configurations allowed by the Dinero parameters.

4.2.2 Design of the HASE Dinero Data-Flow System

The design of the data-flow specification for the Dinero simulator was the most complex aspect of the entire project. The most important issue to take into account when designing a data-flow specification for HASE is that each entity is modelled as a separate thread. Hence synchronisation between entities is non-trivial and a data-flow specification should ensure that all entities have clearly defined receive and send phases so that the undesirable situation of data unexpectedly arriving at an entity, and thus not being processed in a predictable manner, does not occur.

One possible solution to the synchronisation problem would be to use a separate clock entity to synchronise all of the cache entities. However, in the initial data-flow design stage I decided that the most suitable synchronisation method for use in the Dinero simulator would be for each entity to maintain implicit state, whereby each entity contains a number of functions to handle the arrival of particular sequences of events and does not explicitly keep a record of the current entity activity. This decision was made in order to simplify the initial data-flow design process by removing the need for either an elaborate central clock synchronisation protocol or an internal state mechanism, for which the design was not clear due to the early stage of the project.

If the cache architecture could be guaranteed to operate in a consistent manner then the problem of synchronisation would be much simplified as just one data-flow design could be used to guarantee that loss of synchronisation would not occur. However, it soon became apparent that due to the multiple cache configurations supported by Dinero it would be impractical to describe the flow of data through the system using just one model. To further complicate the situation, the data-flow specifications for different cache configurations interact with each other hence requiring awareness of many types of cache configuration to be incorporated into each specification.

The end consequence of both the initial design decision for each entity to maintain implicit state and the relatively high complexity of ensuring that interacting cache configurations could work together without loss of synchronisation resulted in each of the main entities (the cache control unit and cache memory array) having sufficiently complex data-flow charts that I was unable to devise a clear method to display the data-flow specification diagrammatically. This undesirable situation

was caused primarily by the evolution of the initial data-flow model which, as extra functionality was added to the Dinero simulation, resulted in a distinctly non-trivial synchronisation system. In retrospect I would now choose a centrally synchronised communication mechanism which would provide each entity a clear send and receive stage and also require each entity to maintain explicit state (such as a simple counter). Unfortunately by the time I had reached this conclusion the project had reached a sufficient stage of development that the time required to re-write existing (and working) code was prohibitive within the available time.

However, whilst providing a detailed data-flow specification of cache operation was (within time constraints) intractable, I can provide a textual description of cache synchronisation activity as a substitute.

4.2.2.1 An Example of the HASE Dinero Data-Flow System

As mentioned earlier, the HASE simulation control class keeps the simulation running for as long as some entity has an event scheduled in its future event queue. As a consequence my goal in designing the Dinero data-flow system has been to keep the cache in operation by ensuring that each entity maintains the operational status of the simulation by generating at least one event for each message set received, although there are occasional exceptions to this rule.

The following diagram (figure 4.2) shows an example of data-flow activity within a Dinero cache simulation during a CPU read request and is described below.

1. The simulation is initiated by the CPU entity, which sends an address request message to the cache control unit (1). This address request message can be any one of the following:

   - **Read Data**
   - **Write Data (accompanied by data to write)**
   - **Fetch Instruction**
   - **Flush Cache**

2. If the cache receives a *read address* message from the CPU, it calculates the tag, index and block offset components of the address based on the current cache configuration. The index component of the address is then sent to the cache memory array via the line select link (2).

3. The cache memory array now performs an associative set lookup for the set requested by the control unit and then sends the entire associative set to the cache control unit (3).

4. If the data requested by the CPU is present (in other words, one of the tag fields in the associative set matches the tag component of the CPU address and the relevant cache line valid bit is set) then the data is immediately sent to the CPU from the control unit and the timestamp for the cache line just read is updated by the cache control unit sending a line write message to the cache memory array, via the line select link, in conjunction with sending the modified timestamp to the memory array. However, in this example the required data is not stored on a line in the associative set and so a read request is sent by the control unit to main memory (4).

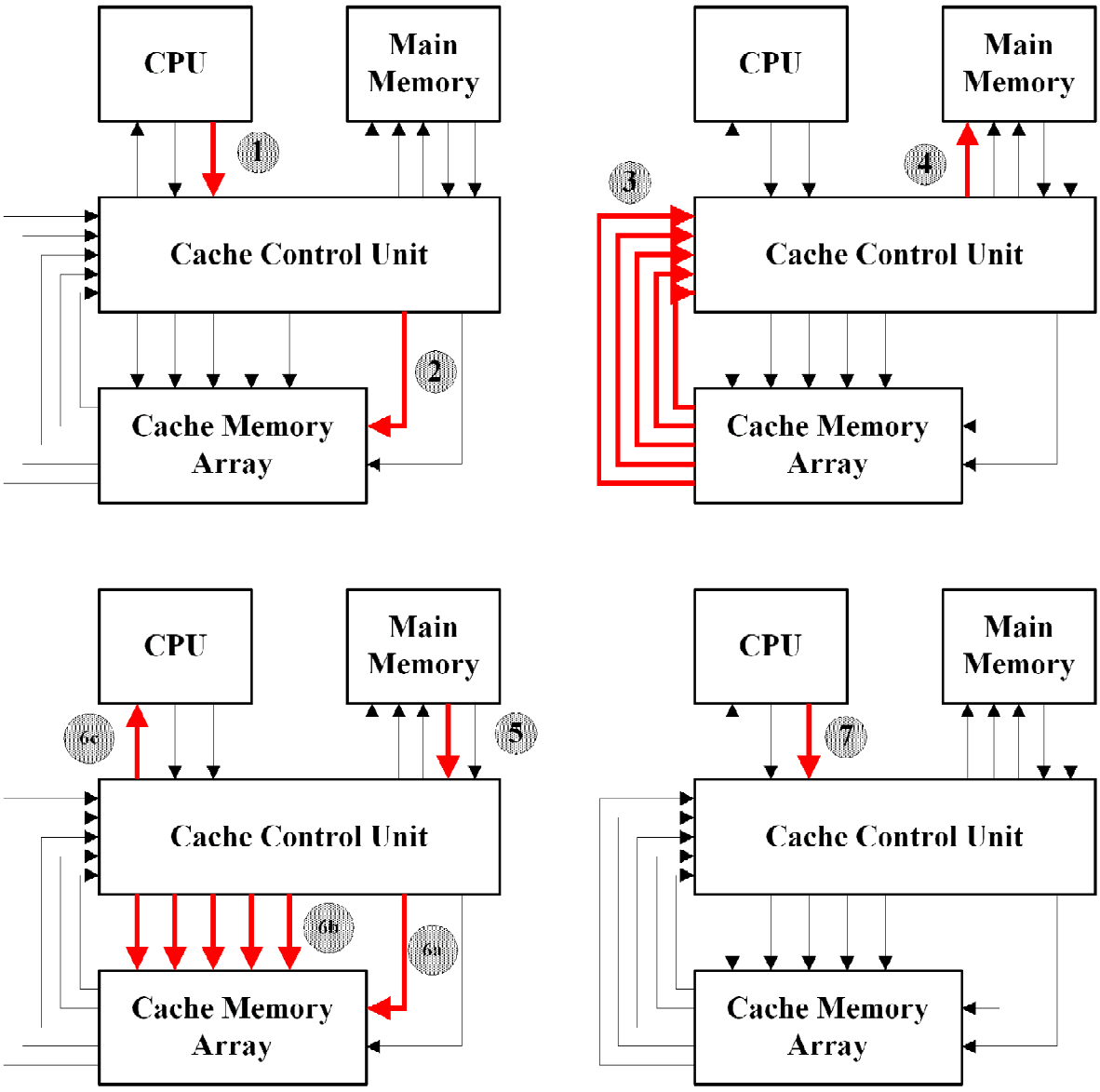


Figure 4.2: HASE Dinero Handling a CPU Read Request

5. The main memory entity services the read request and supplies the requested block to the cache control unit (5).

6. The cache control unit now chooses one of the lines in the associative set loaded from the cache memory array for overwriting. This may involve use of the cache replacement policy (if there are no unused cache lines in the associative set) and could also result in modified data being written out to main memory (if the cache line chosen for replacement contains modified data). However, in this example the cache control unit finds an unused line in the associative set and writes the new contents for the line back to the cache memory array by sending a write message to the memory array on the line select link (6a) in conjunction with sending the data fields of the line to be written to the memory array (6b). The cache control unit also sends the requested data to the CPU in this stage (6c), causing the simulation process to re-iterate.

7. The CPU responds to the control unit data message by sending a new address message, to be serviced, to the cache (7).

Note that each of the four diagrams in figure 4.2 illustrate the basic synchronisation mechanism of Dinero HASE whereby all entities generate at least one event in response to each message set received.

The above brief description provides an example of how the HASE Dinero simulator operates, but I should like to emphasise that this is a relatively trivial example and that to describe all possible cache activity in a similar manner would be a project unto itself!

4.2.2.2 The Representation of Data in the Simulation

The final design issue relevant to this stage of the report is that of the actual data stored in the cache and main memory, and the data provided for writing by the CPU.

My project goal was to implement Dinero as a cache simulator only (not as a modular HASE cache component) and so HASE Dinero does not have any 'real' data handling capability. In other words, it is not currently possible to plug HASE Dinero into a HASE CPU simulation to act as the memory hierarchy component because the Dinero cache and main memory do not have external data throughput capability.

Instead, I decided to use sequence numbers to represent words of memory. Words written by the CPU are always represented by negative sequence numbers in the cache, whereas words loaded from main memory are shown as positive sequence numbers. In addition to this, the CPU and main memory entities have viewable *data in* and *data out* parameters. The combination of these features will allow a cache architecture student to track the movement of data through the memory hierarchy.

4.2.3 HASE Dinero Implementation Details

In this section of the report I shall list and describe the implementation techniques used to achieve a working model of Dinero in HASE.

4.2.3.1 Trace Files

Whilst my primary design goal for HASE Dinero was to produce a cache architecture teaching environment, the secondary objective has been to provide a way of accurately experimenting with different cache combinations. As Dinero is a trace file based simulator an essential design feature of the HASE implementation was to allow the user to either choose from a set of characteristic trace files, or to specify a custom trace file to be used.

Part of last year's Edinburgh University computer architecture course involved a coursework module, using Dinero, for which a selection of three trace files was provided, each modelling a different but representative type of application: **GCC**, **Spice** and **Latex**. Each of these trace files contain around one million memory references and so are large enough for realistic cache sizes to be simulated (the larger the cache, the more trace file references required to average out any discrepancies in cache performance). These three trace files have been made available to all HASE Dinero simulations.

At this point in the report, I shall briefly introduce a concept to be examined in much more detail later (see **section 4.2.4**). During the course of the project I found it necessary to split HASE Dinero into two modes of operation: Animated mode and Fast mode. Each of the two modes has different trace file requirements based on whether the simulation goal is to show how the cache operates (Animated mode) or quickly produce accurate results (Fast mode).

In order to load the trace file references into the Animated HASE simulation, I decided to use an array capable of holding 1024 trace file lines. This was an arbitrarily chosen number since the maximum storage capacity of the array is unlikely to ever be reached due to the lengthy amount of time required to both run and view an Animated HASE Dinero simulation. Moreover, a five hundred line trace file processing limit is currently implemented in the Animated Dinero HASE simulator to prevent the simulation results trace file from consuming vast amounts of hard disk space.

This 1024 element array is initialised at the start of each simulation by the CPU entity, which fills the array (up to a user-specified limit) before sending the first address reference to the cache. In the event that a simulation requires more than 1024 references (which is not currently possible) then the CPU entity fills the trace array on the first attempt, then re-fills the array as it empties.

As the name implies, the Fast mode of Dinero is designed to determine cache results as quickly as possible. As a consequence, a different simulation method is used to that of Animated mode and there is effectively no restriction on the number of trace references that can be processed during a simulation run. The same technique as Animated mode is used, in that a trace array is periodically refilled as it empties. **Section 4.2.4** provides a detailed description of the design and implementation procedure for the HASE Dinero Fast mode.

For both Animated mode and Fast mode, a facility is provided for the user to specify a custom trace file to be used. This trace file is treated exactly the same as one of the three standard trace files with the exception that several checks are performed on the trace file, before simulation starts, to test that the file contains valid data (see appendix B).

4.2.3.2 Variable Cache Size

Dinero allows the size of the cache to be specified for each simulation run. If I had used the UNIX version of HASE this would have presented a problem since each array must be statically declared. In order to change the size of a UNIX HASE array a re-compilation must occur, which is obviously not ideal. Fortunately, NT HASE allows the use of dynamic arrays whereby the array size can be specified by a variable parameter thus allowing the array size to be changed without code re-compilation.

The cache memory is modelled as an array of a structure type corresponding to a cache line. Each cache line contains storage for the valid bit, modified bit, timestamp, tag and block.

The following sample of code, taken from the Dinero EDL file, shows the declaration of the cache line struct followed by the declaration of an array of type cache_line (used to model the cache memory). Note that the Cache_Contents array declaration has a parameter called UCACHE_LINES, which enables the size of the array to be dynamically modified.

```
STRUCT( cache_line, [RSTRUCT( valid_packet, VP),
                     RSTRUCT( modified_packet, MP),
                     RSTRUCT( timestamp_packet, TIP),
                     RSTRUCT( tag_packet, TAP),
                     RARRAY( Cache_Block_Line, CBL)]
);

ARRAY( Cache_Contents, UCACHE_LINES, cache_line);
```

4.2.3.3 Variable Block Size

In addition to allowing the cache size to be varied, Dinero also allows the number of CPU words per cache line to be specified (the block size). Variable block size is once again achieved in HASE Dinero by using dynamic arrays. The only difference from the previous example is that the dynamic array exists within another dynamic array (the cache line).

The cache line structure in the previous example contains the following element which models the block component of the cache line:

```
RARRAY( Cache_Block_Line, CBL)]);
```

Cache_Block_Line is in fact a dynamic array previously declared in the Dinero EDL file as follows:

```
ARRAY( Cache_Block_Line, BLOCK_LINE, int);
```

The BLOCK_LINE parameter determines the number of elements in instances of the Cache_Block_Line array.

Hence both variable cache size and variable block size are achieved through the relatively straightforward use of dynamic arrays.

4.2.3.4 Unified or (Data and Instruction) Cache Model

The cache architecture described in **section 4.2.1** does not provide an explicit facility to represent a data / instruction cache model, although it does intuitively represent a unified cache model. This was a design decision justified by a desire to keep the cache architecture as simple and versatile as possible. If a facility had been provided to view the cache as separate data and instruction entities, the facility could not have been masked if the cache being simulated was of the unified variety, hence being a cause of confusion for the cache design student.

The solution I used was to add a HASE parameter to the cache control unit that indicates whether the control unit is currently accessing the unified, data or instruction cache. This parameter is displayed on screen during animated trace file playback and so provides an easy way to determine which is the active cache. In order to complete the cache split facility, the cache memory array entity contains three separate cache arrays, one for each of the three possible cache types (although only one of either unified cache or data cache / instruction cache is ever used in any particular simulation run). When the cache memory array receives a read or write request, it links to the cache control unit entity and reads the current value of the *active_cache* parameter so that the correct cache array is read or written.

4.2.3.5 Variable Associativity

My initial design plan to implement variable associativity was for the cache memory to send the entire requested associative set to the cache control unit if the level of associativity was set to direct-mapped or set-associative. If the cache was fully associative then the cache memory would only appear to send the entire associative set (which in this case would be the entire cache!) to the control unit, but in fact only send an empty data packet. As the fully-associative set was not to be sent to the control unit, it would be necessary for the cache memory array to perform the following functions (normally carried out by the control unit): cache lookup, replacement policy enforcement and data swapping (in the event of a modified line being overwritten).

A key point to make at this time is that the above separation of functionality between the control unit and the memory array, depending on the associativity class simulated, is transparent to the simulation user as the animation of the display is not effected.

Unfortunately, after I had implemented the above scheme a problem associated with the set-associative policy was discovered. The actual transfer of information between the separate threads was not a concern as HASE uses an efficient pointer mechanism for entity communication. What was a concern is that HASE writes the contents of all messages to the results trace file. In the event that the associativity is fairly high (of size eight and above) then the volume of data written to the simulation results trace file becomes unacceptable, with trace files of over double the expected size being produced.

At first I tried to disable the HASE feature whereby the data content of all messages is traced. Unfortunately, HASE has only two options concerning the tracing of messages: on or off.

The solution I implemented was to extend the empty message feature of the fully-associative model to include both set-associative and direct-mapped caches. The data is instead communicated from the memory array to the control unit by means of an internally declared dynamic array in the cache memory array entity. This array is initialised to be the size of the associative set at the start of each simulation, and is concurrently read by the control unit as the memory array sends the empty data packet to the control unit.

4.2.3.6 Cache Replacement Policy

The cache replacement policies were trivial to implement, being achieved as follows:

- **LRU**
  Every time a cache line is read or written, the cache control unit writes the current simulation time to the timestamp field of the cache line. When an LRU replacement is to be made (because there are no invalid lines in the relevant associative set), the cache line with the oldest timestamp is selected for replacement.

- **FIFO**
  Each associative set is simulated as a set of parallel shift registers. Every time a new line is written to the set, the content of all existing lines is moved down by one position, resulting in lines being 'pushed off' the end of the associative set and hence being removed from the cache. A FIFO replacement policy implemented in hardware would not use this mechanism, instead using one extra bit per cache line, with only one of the extra bits ever being set at the same time. The cache line with the extra bit set would be chosen for replacement each time. However, the parallel shift mechanism I have used in HASE Dinero is a useful teaching aid as simulation users can easily observe the principles of FIFO operation.

- **Random**
  Every time a replacement is required, a line is pseudo-randomly chosen from the candidate associative set, using the C *rand* function.

The only other concern with implementing the replacement policies is ensuring that if the cache uses a copy-back policy and a line is chosen to be overwritten that has its modified bit set then the old block must be written out to main memory.

4.2.3.7 Cache Write Policy

The implementation of this policy primarily concerns the cache control unit. If the write policy is set to copy-back then all data write operations from the CPU must result in a block being loaded from main memory and combined with the data word written by the CPU. The cache line used to store the modified block must have its modified bit set.

If the write policy is set to write-through then data writes from the CPU must always be written immediately to main memory. If the CPU write command causes a cache hit, then the cache must be updated with the CPU write data. However, if a cache miss occurs for the CPU write then the write allocation policy determines whether the CPU write data should be stored in the cache.

After handling a write request, the cache control unit signals that it is ready to continue by using a feature of HASE to send an invisible data packet (from the simulation user's perspective) to the CPU. The CPU responds by sending the next address packet to the cache (assuming that all requested trace file references have not been processed).

4.2.3.8 Write Allocation Policy

The write allocation policy is closely related to the cache write policy.

If the cache write policy is set to copy-back then the appropriate block must be loaded from main memory on a CPU write (allocate), assuming the relevant block is not currently in the cache, before the modified CPU word can be written to the cache.

If the cache write policy is write-through and a cache miss occurs for a CPU write operation, then the write allocation policy determines whether the block modified by a CPU write is stored in the cache (allocate) or not (no allocate).

Note that if the write policy is write-through then no cache line will ever have its modified bit set, since all CPU writes propagate to main memory immediately.

4.2.4 HASE Dinero: A Fast Simulation Mode

During the process of implementing Dinero it became apparent that HASE was causing a number of simulation efficiency problems. As a consequence I decided to split the simulator into two modes of operation, each completely distinct from the other. A Fast simulation mode was created to allow the quick evaluation of a cache configuration. The existing work completed in HASE was incorporated, unchanged, into the second simulation mode: Animated. The Animated simulation mode allows a cache design student to observe the operation of a cache configuration during simulation.

The primary cause for concern with the efficiency of HASE was that all simulation activity is written to a results trace file during simulation. The HASE results trace file quickly becomes very large as the number of trace file references processed increases. Figure 4.3 indicates the significance of this problem.

The cache configuration used to produce the results in figure 4.3 was arbitrarily chosen, as whichever cache parameter combination is simulated, the size of the results trace file scales in the same pattern. The cache configuration used to generate the results in figure 4.3 was as follows (provided to allow the results to be replicated):

- Unified Cache Size: 128 bytes
- Block Size: 8 bytes
- Associativity: Full
- Trace File: GCC
- Fetch Policy: Demand

Figure 4.3: The HASE Results Trace File Problem

The significance of the results in figure 4.3 is that the size of the results trace file linearly scales with the number of Dinero trace file references processed. When this fact is combined with knowledge that the number of trace file references required to produce accurate simulation results is typically about one million [17], the results trace file size problem becomes clear. Based on the linear trend shown in figure 4.3, the projected results trace file size for one million processed references is approximately three gigabytes!

Another problem I discovered whilst using HASE is that each results trace file line has a 256 character limit. Because HASE writes all array updates to the results trace file (in order to update array contents during animated playback), when a block size higher than eight bytes is used the length of certain results trace file lines exceed 256 characters. Consequently Animated mode cannot be used to simulate caches with a block size higher than eight bytes. This does not cause a significant problem as it is unlikely a cache design student would want to simulate a cache with more than an eight byte block size due to the difficulty this would cause in understanding the HASE display during animation.

4.2.4.1 Loss of Performance Attributable to HASE

The Animated mode of HASE Dinero (in which all simulation activity is traced) cannot be used to realistically determine the performance of a cache configuration and can only be used as a teaching tool. This is primarily due to the extremely large results trace file storage requirement for processing the necessary number of trace file references to produce accurate simulation results.

In addition to the problematic storage requirement for the results trace file, the time required to run a simulation on a large scale using Animated mode is also prohibitive due to the bottleneck caused by writing the results trace file to disk.

It was clear that the Animated mode would not be used for processing a large number of trace references not just because of the problems described above but also because it is highly unlikely that anyone would want to watch HASE Dinero animate the processing of one million trace references, which would probably take several weeks.

An obvious solution to this problem would be to disable the results file tracing of all simulation activity, apart from the final simulation results, when executing a simulation using a large input trace file. Unfortunately, the Animated mode suffers from some additional problems that will not allow for high-speed trace file processing, as follows:

1. The Dinero cache is effectively split into three architectural components (the memory array, control unit and CPU) which means that for each trace file reference processed many communications must take place between the separate threads modelling the three entities. Clearly this is going to cause a significant efficiency problem. Additionally, the split of the cache into three completely separate bodies of code prohibits the use of optimisation techniques which could otherwise be used to speedup the processing of each trace reference.

2. Memory usage by the HASE data types required for animation of the display is not efficient. This problem is examined in more detail later in this section.

3. The HASE simulation environment is not designed to be fast. Any function call made to access simulation control functionality results in multiple levels of recursive function call nesting due to the emphasis on code reuse throughout the body of HASE simulation code. When this fact is combined with the overhead of communication between separate HASE threads it is clear that HASE does not provide an efficient simulation environment.

As has been mentioned previously, I decided to split HASE Dinero into two modes of operation as a consequence of the above problems. Animated mode allows simulation activity to be animated in the HASE display window and was implemented using all of the techniques described in the earlier parts of this chapter. Fast mode was created using just one HASE entity, invisible to the simulation user. The Fast mode entity does not make any calls to the simulation control class. Instead the entire simulation takes place in the space of just one simulation event: the writing of the final results to the trace file.

The Fast mode of HASE Dinero clearly diverges from the primary teaching goal of this project, but is provided so that the secondary project goal is met: the provision of a way for cache design students to experiment with cache design techniques. As this is not possible in Animated mode due to the limit on the number of trace references that can be processed, Fast mode provides an alternative solution.

The following sections within this part of the report describe the techniques used to ensure that the Fast mode of HASE Dinero has good performance metrics.

4.2.4.2 The Cache Lookup Problem and Possible Solutions

The main time consuming activity within any software-based cache simulation is determining whether a cache hit has occurred for each lookup processed. Clearly for a direct mapped cache it is very easy to determine if a hit has occurred as just one cache line needs to be checked. However, if a fully associative cache is simulated and a required block is not contained in the cache, every cache line must nonetheless be checked, which is clearly a time consuming process. For example, if a one kilobyte cache with a block size of sixteen bytes is simulated with one million trace file references, then an upper bound on the number of cache line checks to be performed is in the order of tens of millions.

Several techniques to speedup the time required to determine whether a cache hit has occurred, compared to a linear search of the tag array, have been proposed in the literature [13]. Of these, two techniques are most recommended:

The Preferred Location Scheme [13]

This scheme attempts to store each block in the same location it would be if the cache were direct-mapped. When determining whether a cache hit has occurred, the direct mapped location of the desired block is checked first (this is achieved by treating the first part of the tag field of each address as the last part of the index field). If the block is not found at its direct mapped location then a sequential search of the associative set is performed. If the sequential search discovers the required block then it is swapped with the block currently stored in its direct mapped location. If the required block is not found by the sequential search (and so a cache miss has occurred), the block is loaded to its direct mapped location and the existing block (if any) is moved to the LRU location in the set.

Clearly this scheme requires an efficient way to maintain LRU information for each associative set. This is achieved by the use of a Most Recently Used (MRU) encoding method. The MRU encoding scheme employed requires a unique binary encoding for all possible orderings (in terms of the access order of the lines) within an associative set to be pre-computed. For example, if the associativity is set to four, then there are four factorial possible combinations (which is twenty four required binary encodings). This scheme then takes an additional step by storing the next MRU encoding reached after an access to any of the four set lines, in terms of each of the twenty four possible current encodings, in a statically declared table. This table allows the next binary MRU state to be computed on a line access, based on the current state, by a simple *or* operation.

The MRU Algorithm [18]

This scheme is much simpler than the preferred location scheme. Every time a cache line is accessed, it is moved to a ‘first location’ slot within its associative set by a simple swap with the replaced line. Every time a cache lookup occurs, a sequential search of the appropriate associative set occurs, with the first location slot always being checked first.

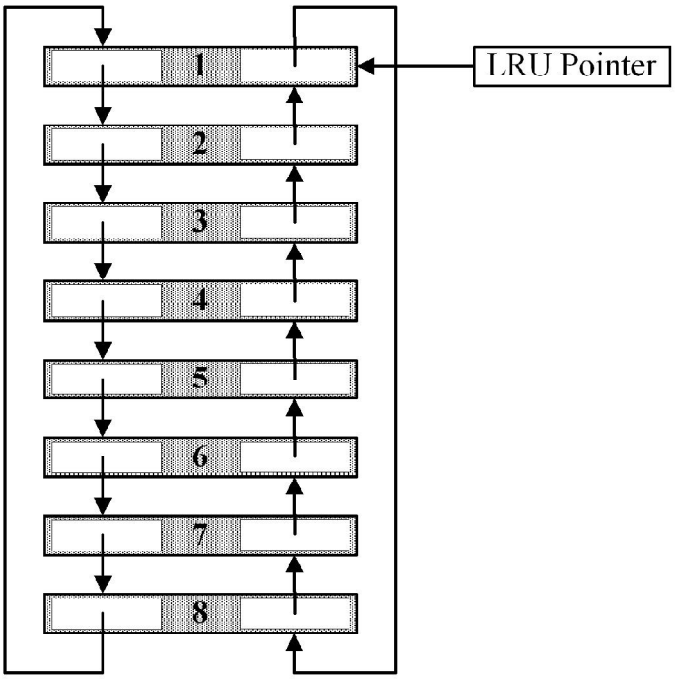
4.2.4.3 An Alternative Fast Cache Lookup Solution

Of the two algorithms outlined opposite, the preferred location scheme has much better performance [13]. Unfortunately, this scheme is not well suited for use within the HASE Dinero simulation primarily because the associative set size can be changed within an arbitrary range by the user. The problem of writing a function to generate the required MRU table, based on the associativity level chosen for the simulation, would be non-trivial. An extra consideration associated with the preferred location algorithm is that the MRU table size scales exponentially with the associative set size. As HASE Dinero allows any level of associativity (for example: 1024), the MRU table computation and storage could clearly become an issue.

As a consequence, I decided to design an alternative fast cache lookup algorithm which would not require the complex pre-computation of an MRU table, as in the preferred location scheme, but would offer better performance that the MRU algorithm described above.

Storing LRU Information

It is not desirable to store LRU information in the form of a HASE timestamp due to the high storage requirements (each Animated mode cache line requires a 32-bit integer to store the simulation time during which it was last accessed). As the MRU table employed by the preferred location scheme is also not suitable for HASE due to the variable, and potentially very large, associativity level allowed I decided to design an alternative scheme, described below.

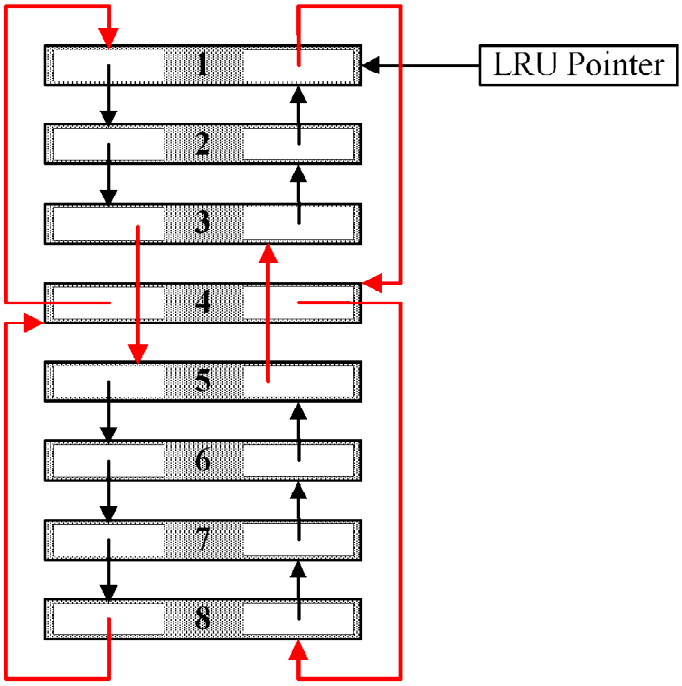


The technique I designed is based around the use of doubly-linked lists of cache lines within each associative set. The structure of each cache line includes a pointer to both the logically previous and logically next items in the LRU ordering of the set. In addition, a separate pointer is stored for each associative set, which points to the LRU item.

Figure 4.4 shows the state of the pointers within an example associative set of eight cache lines at the start of a simulation.

Figure 4.4: The Initial State of the Doubly-Linked LRU Pointer List

When an associative set is empty the cache lines are sequentially loaded until the set is full. There are no pointer updates required to maintain LRU information for this initial filling operation as no out-of-order accesses occur. However, if a cache hit occurs then the LRU information needs to be updated. This is achieved by six pointer updates. Rather than attempt to explain this fairly complex operation textually I have produced an example, shown in Figure 4.5, indicating the pointer updates that occur when a cache hit occurs to line four.



As the LRU pointer for each associative set points to the least recently used location, LRU replacement operations are very straightforward with the only extra work required being to point the LRU pointer to the logically next cache line indicated by the cache line it currently points to. Hence the LRU pointer follows the LRU chain established by the location of accesses to the associative set.

Figure 4.5: Maintaining LRU Ordering in the Associative Set

This LRU scheme is an ideal solution to the fast simulation requirements of Dinero for the following two reasons:

1. The storage requirements for the scheme linearly scale with the size of cache being simulated. Additionally, the storage requirements are low as the only overhead incurred is two pointers per cache line. This compares favourably with the preferred location scheme, for which the storage requirements scale exponentially with the associativity level and the storage requirements are relatively high.

2. The CPU overhead required to maintain the LRU information is constant, being fixed at six pointer update operations, hence allowing the associativity level to be varied with relative impunity compared to the preferred location scheme described earlier.

Reducing Lookup Time

The second major area for performance optimisation in the Fast mode HASE Dinero simulator is that of determining whether a cache hit has occurred for each trace file reference.

I designed a scheme which, theoretically, offers superior performance to that of the MRU algorithm previously described. The technique used is very similar to that of the MRU scheme but takes advantage of locality of reference to optimise the time taken to determine a cache hit, if the required block is contained in the cache. The scheme is targeted only at fully-associative caches. I decided that the additional work required to port the scheme to set associative caches was not required as the simulation time difference between direct-mapped and typical set-associative caches is minimal (see **section 6.4**).

The simulator maintains three external pointers indicating the MRU data read, data write and instruction fetch locations. Each time the fully associative cache is searched for a block, a sequential search starts at the most recently accessed location of the appropriate type.

If the cache is using an LRU replacement policy then the LRU pointers are used to order the sequential search based on order of access relative to the MRU location. If the replacement policy is FIFO or random, the cache is searched in a standard slot-by-slot manner.

The key performance point about the sequential search technique used is that the search is split into two components which scan the set in opposite directions. This is achieved by alternatively scheduling the two searches, on a per cache line basis, with the performance gain exploiting spatial locality. Alternatively, if the cache is using an LRU replacement scheme then the temporal locality of the accesses to the cache is exploited. The exploitation of locality of access is only possible due to the maintenance of a separate pointer for each type of access to the cache, which is the main distinguishing factor from the previously described MRU algorithm.

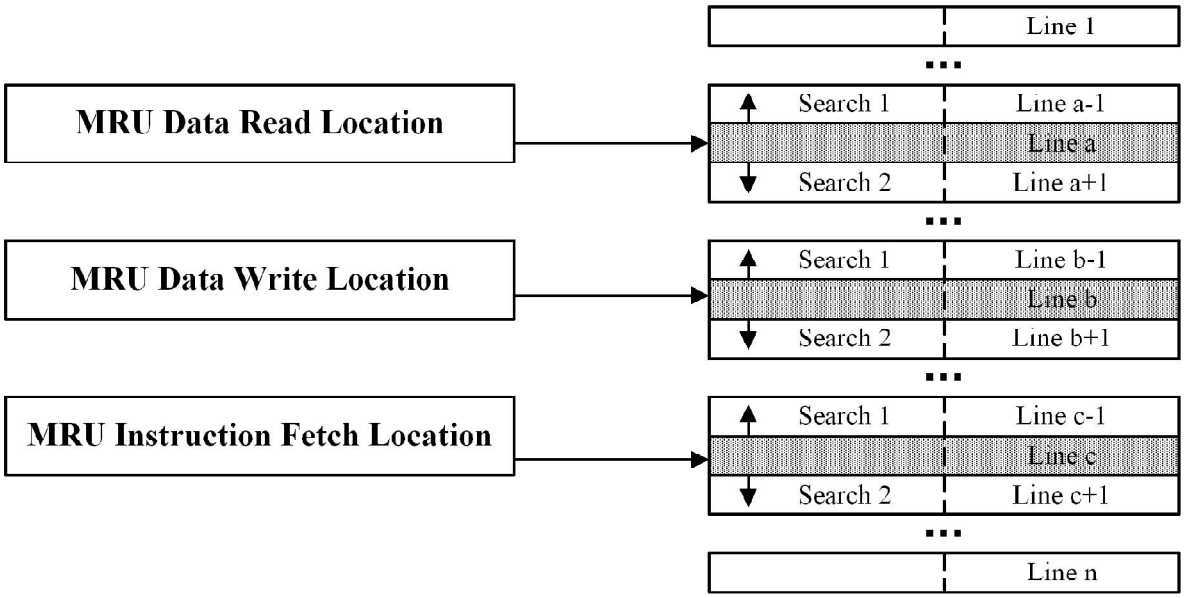


Figure 4.6: Lookup-Time Reduction Scheme

4.2.4.4 Reducing Cache Array Storage Requirements

During the later stages of implementing the Animated mode of HASE Dinero mode it became clear that the main-memory storage space used for representing cache contents during simulation was higher than expected. Upon closer investigation I discovered this was due partly to the inefficient use of data storage in the EDL structure representing a cache line, and partly due to the overhead of HASE related state associated with each HASE variable (this is particularly noticeable for large arrays). The extra state for each HASE variable is used to store data required by HASE when writing results to the trace file.

The implementation of Fast mode does not use HASE variables, instead using standard C++ variables. In addition, the structure used for storing the simulated cache in memory is optimised for minimal storage space. The storage requirements of a cache line in both Animated mode and Fast mode HASE Dinero are compared below:

Storage Requirement:	**Fast mode:**	**Animated mode:**
Valid bit	1 bit	1 bit
Modified bit	1 bit	1 bit
Timestamp	**Nil**	**1 word**
Tag	1 word	1 word
Data	**Nil**	**1 word per byte in block**
Accesses	1 word	1 word

As can be seen, the Fast mode storage requirement is substantially less than that of Animated mode, with at least two less words per cache line being required, and with the possibility of much more relative space-saving if a cache with a large block size is being simulated.

However, by far the greatest cause of memory space-saving in the Fast mode of HASE Dinero is that whilst dynamic arrays are used for the same effect as Animated mode, they are not declared using a HASE construct instead being defined directly in the C++ code using the *new* memory allocation function. Whilst I did not attempt to discover exactly why HASE does not handle dynamic arrays efficiently, in terms of storage space, I took care to ensure that the cache structure used in the Fast mode of Dinero was as memory efficient as possible. The following extract of code shows the C++ structure declaration for a Fast mode Dinero cache line:

```
typedef struct cache_line{
    bool *valid;
    bool modified;
    int tag;
    int accesses;
    struct cache_line *next;
    struct cache_line *prev;
}cache_line_t;
```

Whilst the precise use of each component in the cache_line structure is not of relevance here, the important point to note is that the storage requirement of the structure is low.

4.2.4.5 Evaluating the Performance Improvements

This section of the report compares the simulation execution times and memory requirements of the Animated mode and Fast mode of HASE Dinero.

Simulation Speed

The objective of the simulation speed tests is to illustrate the performance gains achieved over Animated mode by the Fast simulation mode.

The main speed-determining factor in any simulation execution is the number of trace file references processed. As a consequence, the cache configuration is fixed for each of the tests executed in this section. The variance of Fast mode simulation execution time due to varying the cache configuration is examined in **section 6.4**.

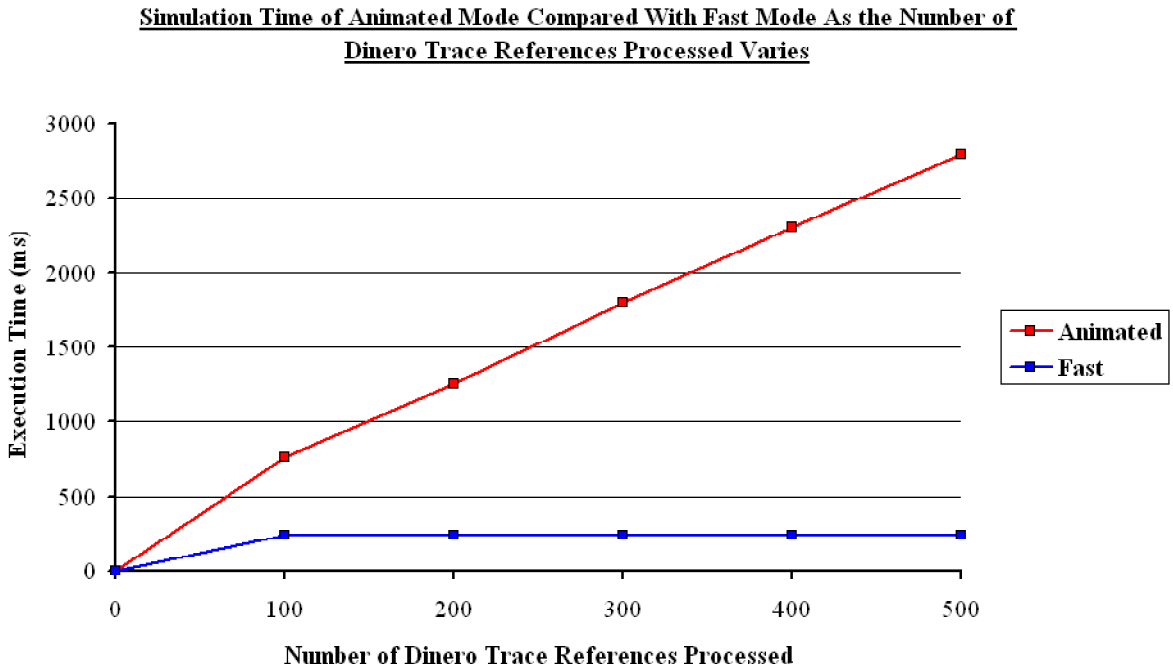


Figure 4.7: Execution Time Differences Between Animated Mode and Fast Mode

The following cache configuration was used to produce the results in figure 4.7 (provided so that the results can be replicated):

- Unified Cache Size: 128 bytes
- Block Size: 8 bytes
- Associativity: Full
- Trace File: GCC
- Fetch Policy: Demand

Figure 4.7 clearly shows that whilst the execution time for Animated mode linearly scales with the number of trace references processed, the execution time for Fast mode does not exceed 240ms. In fact, using the same cache configuration, the Fast mode simulation requires just 6760ms to process one million trace file references, which is just over double the time required for the Animated mode simulation to process five hundred trace file references. This is an important performance metric which clearly indicates that the Fast simulation mode is genuinely much faster then the Animated mode.

The results shown in figure 4.7 were accurately obtained using an in-built HASE function.

Simulation Memory Requirements

When using either the Fast or Animated mode of HASE Dinero, the memory requirement of the simulation does not scale with the number of trace file references processed, which is as expected. Instead, the memory requirements scale with the size of cache simulated. As a consequence, figure 4.8 shows how the memory requirements of Animated and Fast mode compare with different simulated cache sizes.

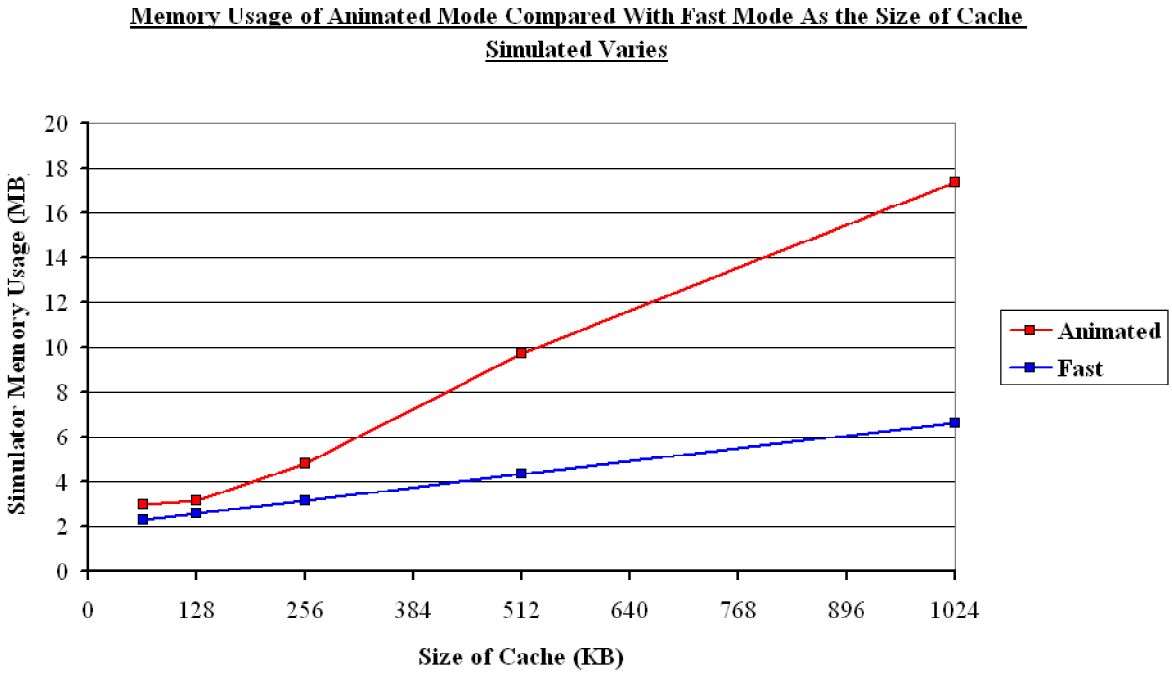


Figure 4.8: A Comparison Between Animated Mode and Fast Mode Memory Usage

The following cache configuration was used to produce the results in figure 4.8 (provided so that the results can be replicated):

- Block Size: 8 bytes
- Associativity: Full
- Trace File: GCC
- Fetch Policy: Demand
- References Processed: Animated mode (500), Fast mode (1000000)

The memory requirements of the different simulation modes were captured using the Windows NT Task Manager. Hence the number of trace references processed was scaled to each simulation mode so that a reasonable length of time was provided to observe the memory usage, bearing in mind that the number of trace references processed does not affect the simulation memory usage.

The results in figure 4.8 demonstrate that the Fast mode of HASE Dinero is suitable for simulating large cache sizes, with less than 50% of the memory requirement of Animated mode for a one megabyte cache. Furthermore, the memory requirement trend lines for Animated mode and Fast mode can be seen to be diverging as the cache size increases. This is due to the more efficient use of memory in the Fast version of the simulator.

4.2.4.6 HASE Dinero Fast Mode Conclusion

The objective of splitting HASE Dinero into two modes of operation was to allow cache design students to experiment with realistic cache configurations using at least one million trace file references. This was not possible using the Animated mode of HASE Dinero due to both the slow simulation time and the high memory requirement.

The Fast mode of HASE Dinero was designed to address both of the main problems with Animated mode and has been shown to be a success by the much higher performance metrics demonstrated in the previous section.

Whilst the poor performance of the Animated mode is obviously of concern, it does not cause a problem for the intended use for which HASE Dinero was designed: providing a cache architecture teaching and evaluation tool. Animated mode will be used by cache design students to discover both how the cache operates and how simulation results are determined. Due to the visual and mental complexity of understanding the animation of a realistically sized cache it is possible to restrict the size and complexity of cache that can be simulated using the Animated mode without effecting the intended use of the simulator.

4.3 Advanced Dinero Features in HASE

The original Dinero simulator provides three main types of advanced cache simulation feature, each related to either the cache architecture or the simulation environment. The succeeding sections in this part of the report describe the implementation techniques used to implement the advanced Dinero features in HASE:

- **4.3.1 Sub-Blocks**
- **4.3.2 Prefetching**
- **4.3.3 Cache Flushing**

The section concludes with a description of two deviations from the implementation of the original Dinero in the HASE Dinero implementation.

4.3.1 Sub-Blocks

Sub-blocks were described in detail in **section 3.3** and so only a brief overview of the technique is supplied here.

Sub-blocks are a cache architecture optimisation technique that allow the atomic unit of cache to main-memory transfers to be reduced from the usual size of *block*, to a smaller *sub-block* size. Each block in the cache is able to store a fixed number of sub-blocks. This provides two main benefits to a cache design:

1. The block size of a cache can be set relatively large, resulting in a reduced tag storage requirement for each cache line.

2. The atomic unit of cache to main-memory transfer is reduced in size, thus resulting in both lower latency and reduced bus bandwidth requirements.

The Implementation

By this stage of development, the HASE Dinero simulator had been split into the two separate modes of operation: Fast and Animated, each of which required a separate implementation of the Dinero sub-block facilities.

There were a number of modifications to the Animated mode implementation that would have been required in order to facilitate sub-block functionality. Firstly, the user-visible cache arrays that display the current content of each cache line would need to be updated so that instead of having just one valid bit, indicating whether the cache line block is valid, there would be a number of valid bits equal to the number of sub-blocks in each block. Secondly, the unit of cache to main memory transfer would need to be reduced from a size of block, to a size of sub-block. This would be straightforward to implement for demand fetches, as the packet currently labelled ‘block’ could simply be re-labelled ‘sub-block’, and the amount of data transferred adjusted to match the relevant sub-block size appropriately. However, the issue of writing modified cache lines to main memory would be more problematic as, for example, if the cache line contained four sub-blocks, each of which was valid at the time the modified cache line was to be discarded, then four sub-block packets would have to be shown being sent from the cache control unit to the main memory.

This multiple packet send was a cause of concern for two main reasons. Firstly, it would be difficult to update the synchronisation model used for the Animated simulation mode so that all possible cache configurations would work correctly with sub-blocks. Secondly, the benefit to a cache design student of being able to view the animation of a sub-block cache configuration is dubious. This is primarily due to the length of time required to animate the transmission of multiple sub-blocks from cache to main memory, each component of which would appear the same to the simulation user. This problem would become particularly apparent if, for example, the sub-block size was set to one byte and the block size set to eight bytes, in which case eight distinct animated packet transfers could potentially be required.

There are obviously various techniques that could have been used to speed-up this process such as instead of sending multiple sub-block packets, sending a single ‘multiple sub-block’ packet or limiting the number of sub-blocks allowed per block.

However, I decided that due to the difficulty involved with implementing sub-blocks, and the dubious benefit this would bring to the cache design student, I would not implement sub-blocks in Animated mode.

Fortunately, implementing sub-blocks in the Fast simulation mode proved to be much easier. This was primarily because the external synchronisation forced by HASE in the Animated mode was not required, but also because as the Fast mode was designed at a later date than the Animated mode it was easier to incorporate sub-blocks into the overall Fast mode simulation design.

As the storage of data sequence numbers in Fast mode was not a requirement, the only cache line array modification required was to convert the valid bit storage element to a dynamic array. This dynamic array can be initialised to a size of one boolean element if sub-blocks are disabled, and otherwise initialised to a size matching the number of sub-blocks per block. An example of the alternative cache line storage requirement of a cache configuration using blocks and a configuration using four sub-blocks per block is shown in figure 4.9.

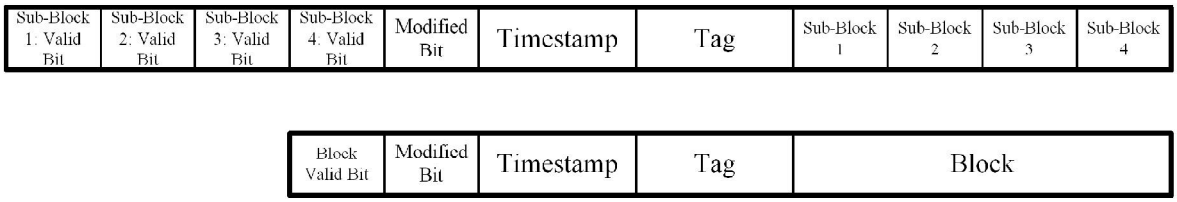


Figure 4.9: Comparing the Storage Requirement of Blocks and Sub-Blocks

Another Fast mode modification required was to only load the sub-block containing the requested CPU word, instead of the entire block, for both demand fetching and the loading of sub-blocks for copy-back writing. It was also important to ensure that cache line sub-block valid bits were set appropriately. This modification is combined with the need to check the appropriate sub-block valid bit when a cache lookup operation is being performed.

Perhaps the most important sub-block implementation detail was to ensure that if a sub-block miss occurs, but some other sub-blocks in the block that should contain the missing sub-block are present in the cache, then the only action required is to set the appropriate sub-block valid bit. In contrast, the block placement scheme always requires a cache line to be loaded with a new block in the event of a cache miss, potentially resulting in an existing valid block in the cache being overwritten.

4.3.2 Prefetching

Dinero prefetching has already been described in **section 3.3**, and so only a brief overview is supplied here.

Prefetching in Dinero is entirely hardware based and uses a very simple fixed stride offset mechanism. Prefetching can be specified to occur for every trace reference or only for trace references that cause a cache miss. If a prefetch operation is executed, then the block with an address *stride* blocks higher in main memory is fetched and stored in a single-element prefetch buffer. Dinero prefetching can be used with both sub-blocks and blocks, with the stride being interpreted in terms of sub-blocks if sub-block placement is enabled.

The prefetching scheme used in the original Dinero is very basic, and I decided that improvements could be made that would allow a cache architecture student to experiment more freely with the range of fixed-stride prefetching techniques.

There is clearly room for optimisation of the hardware prefetching scheme used, on a per cache configuration basis, as was shown in studies by Jouppi from 1990 to 1994 [10]. Jouppi found that a prefetch buffer with a storage capacity of one block caught 15% to 25% of the misses from a four kilobyte, direct-mapped instruction cache with sixteen-byte blocks. However, if the prefetch buffer storage capacity was increased to four blocks, the hit rate improved to 50%, and with sixteen bytes to 72%. Jouppi conducted a similar study with a data cache and found broadly similar results, in that the percentage of cache misses caught by the prefetch buffer scales with the prefetch buffer size.

The modified prefetching scheme I implemented in HASE Dinero allows the prefetch stride for the unified, data and instruction caches to be individually specified. In addition, the size of the prefetch buffer in (sub-) blocks can be specified. This scheme will allow a cache design student to customise the prefetching scheme used for each cache configuration. The benefit of this is that alternative hardware prefetching schemes can be experimented with in order to optimise the prefetch buffer hit rate with the particular cache configuration and trace file in use.

The principal improvements over the original Dinero prefetching scheme are as follows: data and instruction prefetch strides can be individually specified, allowing the different memory access patterns of the two data types (instruction and data) to be exploited, and the size of the prefetch buffer can be modified, allowing a cache design student to experiment with the benefit of increasing the buffer size.

The HASE Dinero prefetching scheme is implemented in both Animated and Fast mode. The remainder of this section is divided into two parts, devoted to the separate implementation details of each of the two simulation modes.

Animated Mode

The Animated mode prefetching implementation was much simplified due to the earlier design decision to not model sub-blocks in Animated mode. However, the code modification required to facilitate prefetching was not a simple task due to the difficulty of ensuring that non-prefetching modes of operation were not adversely affected by altering the complex HASE synchronisation model.

I decided to increase the complexity of the cache architecture slightly by incorporating a separate prefetch address link and prefetch data link between the cache control unit and main memory (see figure 4.1). This additional complexity was easily justified due to the conceptual benefit it brings to the cache design student by separating cache prefetch activity from the usual main memory access mechanism and also allowing the concurrent nature of prefetching to be visualised.

Furthermore, the completely separate prefetch ports simplified the modifications required to the HASE Dinero synchronisation model by isolating the additions made to the simulation.

A separate circular FIFO prefetch buffer array is provided for viewing during simulation animation, accessible from the cache control unit. This buffer array is updated as prefetch data is loaded from main memory, and is checked by the control unit every time a cache lookup operation is performed. If a hit occurs to the prefetch buffer, then the control unit treats the data as though it had been loaded from main memory, loading the required block to a cache line and also sending the requested data word to the CPU.

An important implementation detail incorporated into the Animated prefetching scheme is that a block stored in the prefetch buffer must be invalidated if it is written to by the CPU. If a cache miss occurs then the required block can be loaded from the prefetch buffer, but the prefetch buffer line must be invalidated afterwards as the block is no longer valid. It is also possible for the same block to be stored in both the main cache array and the prefetch buffer, in which case the prefetch buffer copy of the block must be invalidated if the CPU writes to the version stored in the main cache array.

The following structure was used to define the prefetch buffer (copied from the Dinero EDL file):

```
STRUCT( buffer_line, [RINT (cache_type, 0),
                      RINT (valid, 0),
                      RINT (index, 0),
                      RINT (tag, 0),
                      RARRAY(Cache_Block_Line, data)]
);

ARRAY( prefetch_buffer, P_SIZE, buffer_line);
```

The *prefetch_buffer* is a dynamic array, being dependent on the *P_SIZE* HASE variable. Each prefetch buffer line contains a valid bit (to allow an invalid state on initialisation and after certain CPU writes), index field, tag field and data storage area, *Cache_Block_Line*. The index and tag fields are used to uniquely identify the address of each line stored in the prefetch buffer. I chose to store the prefetch address as index and tag components as I felt this would be clearer to the cache design student than a relatively cryptic hexadecimal address string.

Fast Mode

Prefetching in Fast mode was implemented in a manner very similar to Animated mode, in that a separate prefetch buffer is provided, and this prefetch buffer is checked every time a miss occurs to the main cache array. The conceptual implementation is in fact nearly identical to that of Animated mode, with the exception being that the prefetching of sub-blocks can be handled.

Apart from the ability to handle sub-blocks, the only other difference between the implementation of Animated mode and Fast mode is the structure used to represent the prefetch buffer, shown below:

```
typedef struct buffer{
    short cache;
    bool *valid;
    long address;
    int tag;
    int index;
    int block_offset;
}buffer_t;

buffer_t *prefetch_buffer;
```

Efficiency of memory usage is not a concern with the prefetch buffer, as it will always be very small (the benefit from larger prefetch buffers reduces exponentially after approximately sixteen slots). Consequently, the buffer structure shown above contains the different types of storage element required for a block-based prefetch buffer, a sub-block-based prefetch buffer and a victim cache.

The *cache* component of the prefetch buffer structure indicates which cache type (unified, data or instruction) the prefetch data belongs to. The *valid* pointer is initialised using the C++ *new* function, as a dynamic array, to point to an array of boolean values of size one for block-based prefetching, or to a boolean array of size equal to the number of sub-blocks per block.

The *address* component is used to store the numerical address of the prefetch data contained in the prefetch line. The alternative *tag* and *index* components are used if the *buffer_t* is modelling a victim cache (introduced in **section 4.4.2**).

The *block_offset* component is used if sub-block prefetching is enabled, and specifies which sub-block within the block *address* is actually stored in the relevant prefetch buffer slot.

4.3.3 Cache Flushing

Cache flushing is a feature implemented in the original version of Dinero to “crudely simulate multi-programming.” In other words the cache flush feature is an attempt to address one of the main problems with trace file based cache simulation, in that the effect of the multi-tasking environment is not taken into account. Multi-tasking can result in a requirement for frequent cache flushes if the cache stores virtual, as opposed to physical, addresses. If the cache stores virtual addresses then different programs executing on the CPU may have certain virtual addresses in common. This could result in the data from one program being incorrectly interpreted as data belonging to another program if a task switch is allowed to occur without first invalidating all of the cache contents, referred to as flushing the cache.

The cache flush feature in Dinero is very simple. A flush count is specified by the user in terms of how many trace references are to be processed between cache flushes. Hence the frequency of cache flushes is guaranteed, but is not necessarily very indicative of the cache flushes that are actually performed in a 'real' multitasking environment, where the cache flushes will not occur with regular frequency due to the unpredictable requirement of task-switching.

The Animated mode implementation of cache flushing works as follows. When a cache flush is due to occur the CPU entity sends a visible cache flush packet to the cache control unit. The cache control unit then sends a visible cache flush control packet to the memory array, which causes the valid bit of all cache lines in all cache arrays (unified, data and instruction) to be set to false. The valid bits for entries in the prefetch buffer and victim cache are also un-set. The cache control unit responds to the CPU's flush packet with an invisible response packet, thus causing the simulation to continue. In the event of modified cache lines being invalidated during a cache flush, visualisation is not provided of blocks being written to main memory. I decided to not implement such visualisation due to the large number of modified blocks that might potentially need to be copied out of the cache.

The principle behind the cache flush implementation for Fast mode is exactly the same as for Animated mode.

4.3.4 Differences between HASE Dinero and the Original Dinero

There are two main differences between the design features of the original version of Dinero and the HASE version:

- **Abort Prefetch Percent**
- **Cache Hits Breakdown by Hit Type**

Abort Prefetch Percent

The original version of Dinero is documented to include a feature termed 'abort prefetch percent' which, as the name implies, allows the percentage of prefetch operations that should occur but do not, in fact, occur to be specified. The abort prefetch percent feature is intended to provide a way of simulating the effect of data references blocking the prefetching of data from main memory.

When I initially implemented this feature, I was surprised to discover that the prefetch buffer hit rate actually increases as the abort prefetch percent increases. This is obviously contrary to what should be expected. I implemented abort prefetch percent in both Animated mode and Fast mode and thoroughly checked the relevant code to ensure it was operating correctly. Both versions of the simulator still produced the same strange results, in that increasing the abort prefetch percent raised the hit rate to the prefetch buffer. This was particularly noticeable in Fast mode when one million trace references were processed, with a peak prefetch buffer hit ratio being achieved when the abort prefetch percent was set to approximately 90%!

I decided to test the results produced by the original version of Dinero on an identical cache configuration to that used in Fast mode HASE Dinero, with abort prefetch percent enabled, and observed a strange feature of the original Dinero: as soon as the abort prefetch percent feature is enabled, the prefetch mechanism stops working.

My conclusion from this is that the designer of the original version of Dinero observed the same abort prefetch percent behaviour as myself and therefore decided to disable the feature. Whilst I do not know why the apparently illogical prefetch buffer hit pattern is observed as the abort prefetch percent varies, a possible theory is that the number of conflicts caused by prefetch data overwriting existing data in the prefetch buffer is reduced as the abort prefetch percent rises. Of course, there are many other possible theories including that the data loaded on a prefetch buffer hit is frequently required again, shortly afterwards, due to an intermediary demand fetch overwriting the cache line containing the prefetch buffer data, or that the typically prefetched data item is frequently required for the first time just after it would normally have been overwritten in the prefetch buffer.

As the abort prefetch percent feature is a decidedly minor objective in terms of my project goals, I did not spend any further time investigating the unforeseen prefetch buffer hit ratio pattern observed and decided to omit the abort prefetch percent feature from HASE Dinero.

Cache Hits Breakdown by Hit Type

The original version of Dinero provides a breakdown of the location of cache hits, in terms of which of the main cache and the prefetch buffer supplied a hit for each trace reference.

As a consequence, the total number of hits reported by the prefetch buffer and main cache exceeds the number of hits actually supplied by the cache configuration. This is because if both the prefetch buffer and main cache contain the data required to service a trace reference then the number of hits for **both** the prefetch buffer and the main cache are incremented.

I decided that this method of breaking down the number of hits to each cache component was potentially confusing to the cache design student both due to the inconsistency between the total number of hits and the total of the breakdown and because the actual number of hits supplied by the prefetch buffer is not clearly indicated. As a consequence I decided to report the breakdown of hits to different parts of the cache in a hierarchical method which ensures that the total of the breakdown equals the total number of cache hits.

The hit breakdown scheme I have used in HASE Dinero works as follows. Every time a cache lookup operation is performed, the main cache is checked first. This is representative of a realistic cache implementation as the main cache is likely to be highly optimised and thus the quickest cache component to check for a hit. If the main cache provides a hit then its hit count is incremented and no further checks are performed. However, if a miss occurs to the main cache then the prefetch buffer is checked, with the prefetch buffer hit total being incremented if a hit occurs. Should a miss also occur to the prefetch buffer, then the victim cache will finally be checked (see **section 4.4.2** for more information on the victim cache).

Using the scheme described above, the cache design student will be able to easily observe the actual benefit gained from implementing a cache optimisation feature, such as the prefetch buffer or victim cache, as opposed to being shown a list of numbers that does not clearly indicate which architectural feature had the most effect on the hit ratio.

4.4 Additional Features in the HASE Version of Dinero

The HASE version of Dinero incorporates some features that are not included in the original version. These additional features were included either to increase the range of cache configurations that can be simulated or to provide an additional method of results feedback to a cache design student. The following additional features are included in HASE Dinero, each of which will be examined in more detail later in this section:

- **4.4.2 Victim Cache**
- **4.4.3 Breakdown of Cache Utilisation**
- **4.4.4 Breakdown of Cache Misses**

4.4.1 Cache Optimisation Techniques

There are many advanced optimisation techniques available for cache design which attempt to reduce one of: the number of cache misses, the cache miss penalty, or the time required to determine if a cache hit has occurred [10, 17].

Of these advanced techniques, many would be suitable for simulation using the original Dinero. Of the techniques not suited to the original Dinero many require the cache simulation to either provide a realistic timing model or to be part of an out-of-order CPU simulation.

I have listed below the cache optimisation techniques that would make a useful addition to a cache simulator, but have not been implemented in HASE Dinero:

- **Write-Through Buffers**
  This technique requires the cache to be using a write-through write policy. A buffer with a much lower access latency than main memory is placed between the CPU and main memory. When a CPU write operation occurs the data is stored in the write buffer, which passes the data on to main memory as soon as the latency of the main memory allows. The advantage of this technique is that the CPU is only stalled for the amount of time required to write to the write buffer, which is much shorter than the time required to write to main memory. HASE Dinero cannot simulate this technique because it does not have a timing model capable of demonstrating the performance gain of the write buffer.

- **Compiler Controlled Prefetching**
  This technique requires a compiler to generate assembler instructions to control the cache prefetch strategy. For example, the compiler can both indicate when prefetching should occur and the size of the prefetch stride for each prefetch instance. This technique is not suitable for simulation in HASE Dinero as there is no mechanism to supply CPU commands to the cache due to the trace-driven nature of the simulator.

- **Compiler Optimisations**
  This technique involves a cache architecture aware compiler optimising the sequence of commands in a program so that an optimal number of hits to the cache occur. For example, the compiler will try to avoid the undesirable situation of two frequently used blocks being mapped to the same location in a direct-mapped cache (and hence replacing each other when loaded) by arranging the ordering of data storage in an appropriate manner. This technique is not suitable for use in HASE Dinero because the simulation takes place outside of the compiler optimisation application area.

- **Read Miss Priority Over Write**
  This technique requires a cache to always load the data required from main memory to service CPU reads before the data required for CPU writes. This technique recognises that CPU writes do not cause the CPU to stall as once a data write command is issued, the CPU can carry on with normal operation. However, if the CPU issues a data read command it must often wait for the data to be loaded before it can continue. This technique cannot be implemented in a meaningful way in the HASE Dinero simulator because the cache architecture used does not provide a propagation delay model and so it would not be possible to represent the performance gain.

- **Early Restart and Critical Word First**
  This technique recognises that the CPU only requires one word from the block loaded from main memory to service a demand miss. *Early restart* involves passing the part of the block required by the CPU on to the CPU as soon as it arrives at the cache, hence blocking the CPU for less time than the usual method of waiting for the entire block to arrive before sending the requested word to the CPU. *Critical word first* involves requesting the required block to service a CPU request as normal, but specifying which part of the block should be sent first. In this way the actual word requested by the CPU is received first and passed to the CPU, leaving the CPU free to operate while the remainder of the block is loaded to the cache. This technique cannot be simulated in HASE Dinero for the same reason as read miss priority over write, in that propagation delays are not modelled.

- **Non-Blocking Caches to Reduce Stall on Cache Miss**
  This technique requires the cache to immediately return control to the CPU once a demand miss has occurred. The cache will notify the CPU once the required data has been loaded from main memory. The benefit of this technique is that the CPU is blocked for as short an amount of time as possible. This technique cannot be modelled in HASE Dinero as it requires the cache to be simulated as part of an out of order CPU simulation in order to be demonstrated.

- **Second Level Caches**
  This technique allows a second level of cache to be placed between the primary CPU cache(s) and main memory. The second level cache is always larger than the primary cache and has a higher access latency as a consequence, although the latency will be much lower than the main memory access time. The benefit of this technique is that there is a greater chance of a reduced latency requirement in servicing a primary cache demand miss, whilst the hardware cost is relatively low compared to increasing the size of the primary cache to match the size of the secondary cache. This technique is suitable for simulation in HASE Dinero but has not been implemented due to project time constraints.

4.4.2 Victim Cache

Whilst it could be beneficial to a cache design student to be able to experiment with the optimisation techniques outlined in the previous section, it is not possible to implement the techniques using Dinero due to a combination of the fact that it is a trace-driven cache simulation, and that it does not have a propagation delay model that can be used to provide further information on cache performance.

However, there is one useful cache optimisation technique that I was able to implement in HASE Dinero: a victim cache.

A victim cache is a very small, fully associative, FIFO cache that is used to store all blocks discarded as a result of a conflict in the main cache. Every time a main cache miss occurs, the victim cache is checked. In the event that the victim cache contains the required data, the victim cache data is swapped with the data stored at its location in the main cache. Jouppi (1990) found that a four-entry victim cache removed 20% to 95% of the conflict misses in a four kilobyte direct-mapped data cache [10].

The victim cache optimisation technique has been implemented in both simulation modes of HASE Dinero. A user-variable parameter allows the number of blocks that can be stored in the victim cache to be specified before each simulation run.

The victim cache buffer is implemented as a dynamic array and is accessible from the cache control unit in animated mode. The victim cache is checked for a hit only after a miss to both the main cache and prefetch buffer. This method allows the additional benefit of a victim cache over prefetching to be examined and also, due to the consistency of trace driven simulation, allows the separate effects of the victim cache and prefetch buffer to be checked by simply disabling one of the two features.

Unlike the prefetch buffer, it is not ever necessary to invalidate data in the victim cache on a CPU write, as the same block cannot be contained in both the main cache and the victim cache at the same time due to the swap that occurs each time a victim cache item is used. The only situation requiring the victim cache contents to be invalidated is a cache flush.

4.4.3 Breakdown of Cache Utilisation

Cache utilisation is a measure of how efficiently the resources available in the cache are utilised. In the context of the HASE Dinero cache simulation, cache utilisation is expressed as a percentage indicating how well distributed the location of new line writes within the cache have been. A new line write is caused by a demand miss requiring a (sub-) block to be loaded to the cache from main memory. Each line in the cache has a counter that is incremented every time a (sub-) block is loaded to the line.

As an example of how the cache utilisation is calculated, consider a direct-mapped cache with ten lines that services one hundred trace file references, each of which requires a new block to be loaded from main memory. If each cache line is loaded with a new block ten times during the simulation then the cache utilisation will be 100% as the loads to the cache are evenly distributed. However, if just one cache line was loaded with a new block one hundred times then the cache utilisation would be 10% as just one cache line receives the entire load.

Whilst the above example is theoretically possible, in practice calculating the cache utilisation is more complex. An additional factor to be taken into account when calculating the cache utilisation is the replacement policy. Whilst the LRU and random replacement policies have no effect on the simple algorithm outlined above, if the replacement policy is FIFO then, if the above method was used, it would always appear that the cache has low utilisation because only the first line of each associative set will ever be loaded with a new block (in Animated mode HASE Dinero). The solution to this problem is to treat each associative set as just one cache line, for the purposes of calculating the cache utilisation. Hence the cache utilisation calculation for caches using a FIFO replacement policy indicates the distribution of block-loads between the different associative sets, as opposed to different cache lines. As a consequence, the utilisation for a FIFO fully associative cache is always 100%.

The formula used to calculate cache utilisation for a direct mapped cache in HASE Dinero is shown below:

```
sum = 0;
avg = (Number_of_Trace_References/Number_of_Cache_Lines);
for(i=0; i<Number_of_Cache_Lines; ++i){
    if((Number_of_Block_Loads[Line_i]-avg)<=0){
        sum += Number_of_Block_Loads[i];
    }
    else sum += avg;
}
return((sum/Number_of_Trace_References)*100));
```

4.4.4 Breakdown of Cache Misses

The breakdown of cache misses provides a useful way to observe how the causes of cache misses vary with different cache configurations. The standard method used to represent a cache miss breakdown is for each miss category to be shown as a percentage of the total misses.

The textbook recommended for students studying the computer architecture module at Edinburgh University [10], provides a classification method for distinguishing classes of cache miss, as follows:

- **Compulsory** misses are caused by the cache being completely empty after program initialisation. Compulsory misses are distinguished from other classes of miss by the loaded block having not been loaded to the cache before.

- **Capacity** misses are caused because a cache is not large enough to hold all of the blocks in the working set of the current program. Capacity misses are classified as those blocks that are loaded to the cache again after being discarded at some previous time.

- **Conflict** cache misses are very similar to capacity misses, but are distinguished by the need for the cache to be direct-mapped or set-associative. Conflict misses are caused by a block being discarded from its associative set and later being loaded again. Hence conflict misses cannot occur in a fully-associative cache. Conflict misses are classified as those misses that occur as a consequence of the level of cache associativity.

There is an alternative cache miss classification system proposed in a research paper on analytical cache model simulation techniques [1], described below:

- **Start-Up Effects** are caused by the initial process of loading the working set of a program to the cache.

- **Non-Stationary Behaviour** is caused by the slow change of a program's working set over time, recognisable by the loading of a block for the first time.

- **Intrinsic Interference** is caused by a finite cache size, where multiple program blocks compete for a cache set and collide.

- **Extrinsic Interference** is a side-effect of multiprogramming, causing the invalidation of cache contents as a process switch occurs.

Whilst I was in the process of evaluating which of the above classification techniques would be best suited for use in HASE Dinero, it became apparent that both techniques require a list of previously loaded blocks to be maintained. This represented a significant problem as the miss classification system would need to operate in the Fast simulation mode for realistic results to be produced. The storage space required and the processing overhead would clearly be prohibitive in implementing a block-address storage mechanism for the Fast simulation mode in which at least one million trace file references can be expected to be simulated. I estimated that the Fast mode processing time would be increased by a factor of three due to the overhead of searching and maintaining the previously-used block list.

As a consequence of the previously-used block problem, I decided to design an alternative cache miss breakdown mechanism that would approximate the results of the Hennessy & Patterson cache miss classification scheme but would not require a list of each previous cache block reference to be maintained, as follows:

- **Compulsory** cache misses are regarded as all misses to the cache that are loaded to a cache line which has not yet had its valid bit set.

- **Conflict** cache misses are caused by any miss that occurs after every line in the cache has had its valid bit set.

- **Capacity** misses are caused by any miss that both occurs before every line in the cache has its valid bit set, and also causes valid cache data to be overwritten.

Clearly, the performance overhead associated with the above scheme is minimal as no list of previously loaded blocks has to be maintained and the classification of each cache miss can be calculated based solely on how many compulsory misses have already occurred and the status of the valid bit on the cache line the block is loaded to.

An unfortunate side-effect of the low-overhead implementation of the miss breakdown scheme is that once the cache is full no more compulsory or conflict misses can occur. As a consequence, in order to gain observable ratios in the miss breakdown, the number of trace file references processed must be limited to a number proportional to the number of lines in the cache being simulated. Whilst this number can be arbitrarily chosen, I found that processing 150 trace file lines for each

cache line produced a characteristic miss ratio breakdown, with the miss ratio components being modified in a predictable manner by the alternative cache architectures simulated. Note that as long as the same number of trace file lines are processed for each cache line, the miss breakdowns of any cache configurations with the same number of cache lines can be compared.

If a comparison of the miss breakdown between cache configurations with a varying number of cache lines is to be performed then the number of trace file references processed should remain constant. I found that setting the number of input trace file lines to be processed to 150 for each cache line in the cache with the largest number of cache lines provided characteristic miss breakdown ratios as the cache size and block size varied.

Sample results produced by varying the associativity of a fixed-size cache configuration are shown in figure 4.10.

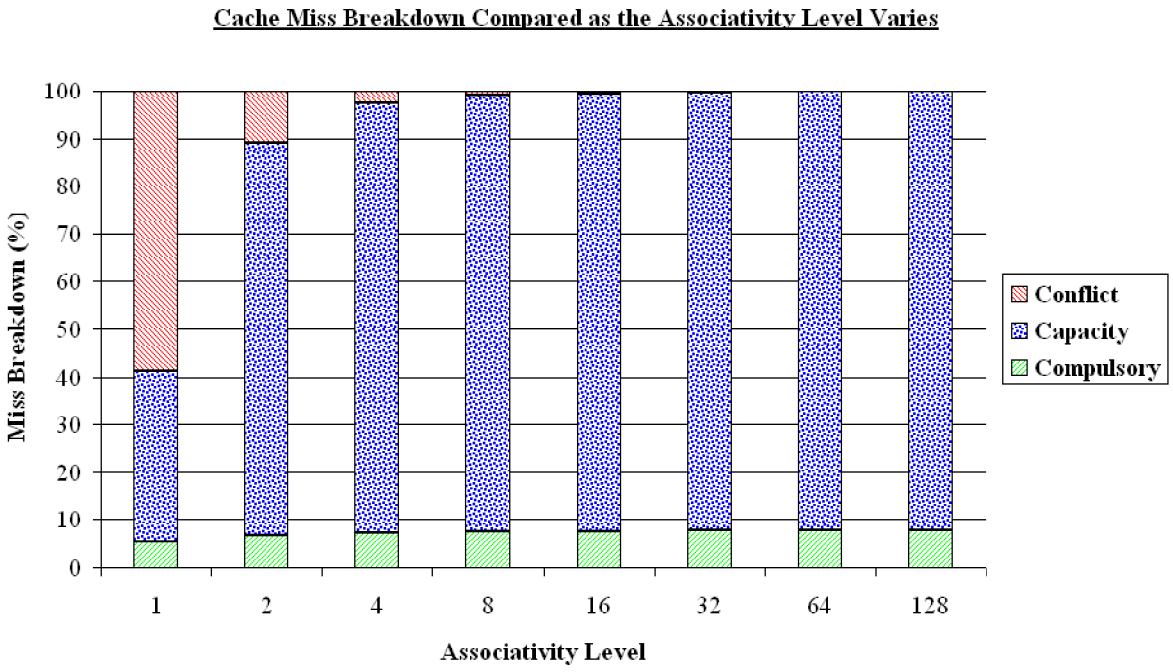


Figure 4.10: How the Miss Breakdown Ratio Varies With Associativity

The following cache configuration was used to produce the results in figure 4.10:

- Unified Cache Size: 4096 bytes
- Block Size: 32 bytes
- Trace File: GCC, 19200 lines processed
- Replacement Policy: LRU
- Fetch Policy: Demand

Figure 4.10 displays the miss breakdown ratios that should be seen due to varying the cache associativity. The compulsory cache misses account for approximately 6% of the misses and the split between capacity and conflict misses depends on the level of associativity in the cache. The larger the associative set is, the less likely it is that a valid cache line will be overwritten and so the proportion of conflict cache misses is relatively lower.

In fact the only complication introduced by the miss breakdown scheme used in HASE Dinero is the need to make clear to users of the simulation both how the scheme operates, together with recommendations for use, and how it differs from the system described by Hennessy & Patterson, in order to avoid potential confusion.

5. The HASE Dinero User Interface

5.1 Overview

This chapter of the report begins by describing the design and implementation of the HASE Dinero user-guide and tutorials, which are two features that have been added to the project to make it more approachable for students with no prior cache architecture knowledge. A description of the HASE Dinero user interface is then provided, together with the design guidelines used during the interface implementation. Finally, a description of the user interface evaluation is provided, together with the evaluation results and remedial action taken to improve the usability of the user interface.

5.2 The HASE Dinero User-Guide and Tutorials

The main goal of the HASE Dinero project was to produce an environment in which computer science students unfamiliar with the concepts of cache architecture could both learn the principles of cache design and experiment with the effectiveness of customised cache configurations. It was clear that an important requirement of the learning environment would be to develop a mechanism that would gradually introduce the concepts of cache architecture in easily understandable phases, and then to ensure that the mechanism produced actually achieved its goal. A large part of the required mechanism, which would demonstrate the principles of cache architecture, was already provided by the Animated mode of HASE Dinero, in which students can observe dynamic details of cache operation at a relatively low level of detail. However, the Animated mode of HASE Dinero by itself is certainly not simple enough that a person with no knowledge of computer caches could be shown the simulator and rapidly develop an understanding of cache operation.

Due to the complexity of both cache operation and understanding the HASE Dinero cache model, it was clear that I would need to develop an easily understandable introduction to HASE Dinero, which would act both as an interface between the general computer science knowledge assumed of the student, and the fairly complex cache architecture animation supplied by the simulator. I decided that the most powerful method available to achieve this goal would be to incorporate a tutorial mode of operation into HASE Dinero.

The tutorial mode of operation would use a very simple cache configuration to demonstrate both the principles of cache operation and how the HASE Dinero simulator operates. The main benefit gained from producing the tutorial mode in HASE Dinero itself, as opposed to writing a purely textual guide, is that HASE can be used to display text on screen during the course of running each tutorial. The tutorial text can be changed during each simulation time unit so that it describes the current cache activity being animated on screen. The ability to display text on screen that matches the current animated activity is a powerful feature which allows tutorials to be developed that are both easy to use and provide an excellent way of gradually introducing the concepts of both cache architecture and the operation of HASE Dinero.

However, in order to meet the primary teaching goal of the project it was not sufficient to just add a tutorial mode to HASE Dinero. It is assumed that students using HASE Dinero do not, initially, have any knowledge of cache architecture or operation and so simply showing such students an

animation of cache activity is not going to answer the most fundamental cache architecture questions. In addition, HASE Dinero required a manual of operation which would both explain how to use the simulator and also provide detailed information on the HASE Dinero implementation.

To address the above concerns I have produced an HTML user-guide for HASE Dinero. The user-guide is primarily concerned with explaining the fundamentals of cache use and architecture such as the principle of locality and how the cache divides a CPU address into tag, index and block-offset components. The other main purpose of the user-guide is to provide clear instructions on how to use the tutorial modes of HASE Dinero, as this will be the first feature of the simulator to be used by new students. In addition to the above two features, the user-guide also provides detailed information on how to use HASE Dinero in both Animated and Fast mode, details of how the cache utilisation and miss breakdown are calculated, and a brief description of the background of HASE Dinero.

The following two sections of the report describe the design and implementation of both the HASE Dinero tutorials and the HTML user-guide.

5.2.1 Tutorial Design

The final implementation of HASE Dinero incorporates fourteen tutorials, each of which demonstrates a different aspect of cache architecture. Before commencing the design of each tutorial, I produced a set of tutorial design guidelines that I felt would have to be met in order to ensure that the tutorials provided an easy-to-use, well structured introduction to the HASE Dinero simulator:

- The tutorials should gradually introduce new concepts, with each tutorial building on concepts introduced in previous tutorials.

- The tutorials should provide an introduction to all of the important cache architecture parameters available in HASE Dinero.

- The tutorials should not only describe the cache architecture itself and the benefits / trade-offs for each of the available cache architecture features, but also demonstrate how the HASE Dinero simulator describes the current cache activity.

- The size of cache simulated in tutorial modes should be kept very small, so that the features demonstrated in the tutorials are as clear as possible.

By using the above guidelines it was clear that the tutorials produced would need to start with the most basic cache design features and then go on to introduce the more advanced features of cache operation, such as the different replacement policies and write policies.

The HASE Dinero user-guide provides a description of both the principle of locality and how the cache divides the CPU address into tag, index and block-offset components, so it is assumed that the student understands both the reason for introducing a cache between the CPU and main memory and how the CPU address is mapped to a location in the cache.

The following part of this section provides an overview of the structure of the tutorials, and how the various design features are demonstrated.

- **Tutorial 1: A Direct-Mapped Cache**
- **Tutorial 2: A Set-Associative Cache**
- **Tutorial 3: A Fully-Associative Cache**

The first three tutorials demonstrate the difference between each of the above three methods of mapping CPU addresses to cache line(s). In addition, the first tutorial highlights how HASE Dinero provides feedback on the current cache activity.

The cache configurations used in the first three tutorials are identical (except for the associativity level) so that the differences caused by varying the associativity level can be easily demonstrated. The modelled cache is eight bytes in size, with each block able to hold one byte (a CPU word is one byte in HASE Dinero). The small cache and block sizes allow the difference between the associativity policies to be clearly demonstrated whilst introducing minimal confusion due to unnecessary information being shown on screen.

The input trace file used is specific to the first three tutorials and has been designed to clearly demonstrate the differences caused by varying the level of associativity. The trace file is four lines long and demonstrates both how the same addresses map to different cache lines depending on which associativity level is chosen, and also the benefit to be gained from increasing the associativity level. The trace file is shown below:

Line 1: Data Read to address 0
Line 2: Instruction Fetch from address 1
Line 3: Data Read from address 0
Line 4: Data Read from address 8

The above trace file has been carefully designed to demonstrate four main features:

1. Line three results in a cache hit as the required data has already been loaded to the cache for line one.

2. Line two causes a block to be loaded to cache line one (cache line numbering starts from zero) for a direct-mapped cache, line two for a two-way set-associative cache, and line one for a fully-associative cache (because cache line zero is already in use).

3. Line four causes the block in line one to be overwritten for a direct-mapped cache, but does not for a fully-associative or two-way set-associative cache as the loaded block can be stored in an empty cache line.

4. The trace file references cause the cache lines to be sequentially loaded in a fully-associative cache, whereas the mapping scheme used for direct-mapped and two-way set-associative caches results in the cache lines being loaded in a non-sequential order.

Instead of describing the detailed implementation of each tutorial, I shall only describe the detailed implementation of the first tutorial, as the same implementation techniques were used for all of the tutorials. Although a detailed description of all of the tutorials is not provided here, it should be clear that the tutorial mode is one of the most important features of the HASE Dinero simulator. The level of detail supplied in the tutorials is sufficient to clearly introduce all of the important cache architectural configurations that can be simulated in HASE Dinero, with the tutorials containing over 6500 words describing simulation activity over more than two hundred discrete simulation time-units.

The Implementation of the First Tutorial

The tutorial modes are implemented as standard simulations in HASE Dinero. This means that in order to view a tutorial, the user must first run the simulation corresponding to the desired tutorial. I decided that this was the best way to implement tutorials on the basis that the benefit of familiarising the user with the fairly complex syntactical methodology required to run a simulation in NT HASE is more important than the problematic repetitive requirement of the need to run a simulation for each tutorial (as opposed to simply loading a pre-computed results trace file).

When a tutorial simulation is executed, HASE Dinero automatically sets all of the cache configuration parameters and specifies which input trace file should be used. As mentioned previously, HASE can provide descriptive text that matches the current simulation activity. This is not an intended use of HASE functionality and was achieved by creating a special tutorial entity, the state of which is displayed on screen. The tutorial entity has eighty-five states, each of which is associated with a different picture to be displayed on screen. Each of these pictures contains descriptive text relevant to a certain sub-set of simulation activity during the course of tutorial animation.

The HASE++ source code for the tutorial entity specifies the simulation time-step delays that should be introduced between changes of state for the tutorial entity during the course of each tutorial animation.

Using the techniques described above it was possible to allow HASE Dinero to completely control the status of each tutorial run, guaranteeing that each tutorial run is identical and that the appropriate descriptive text is always displayed on screen.

Now that the techniques used to implement the tutorials have been described, the actual activity during a tutorial simulation run can be shown. Up to this point in the report the user interface has not been described and it is only introduced at this point so that a description of tutorial activity can be given. **Section 5.3** provides complete information on the HASE Dinero user interface.

Throughout the course of the first tutorial, there are four relevant HASE Dinero screens. These screens are shown opposite in their initial states at the start of the tutorial.

Tutorial 1: Simulation Time-Step 1

The cache architecture display (figure 5.1) indicates that the simulation is idle at the moment; the CPU, main memory, cache control unit and cache memory array are all in an idle state.

The status screen (figure 5.2, left) is only active during Animated mode playback, at which time it provides feedback on current cache activity. The status screen is not currently displaying any useful information as the simulation has not yet started. The unified cache contents window (figure 5.2, right) has been initialised to display the contents of the unified cache. There are eight rows in the unified cache contents window. Each corresponds to a cache line, with the left-to-right ordering of the contents of each row being as follows:

- Valid Bit
- Modified Bit
- Timestamp
- Tag
- Byte Sequence Number (the block-size is set to just one byte)

During the course of the simulation, the unified cache contents window will be dynamically updated to reflect the current contents of the unified cache.

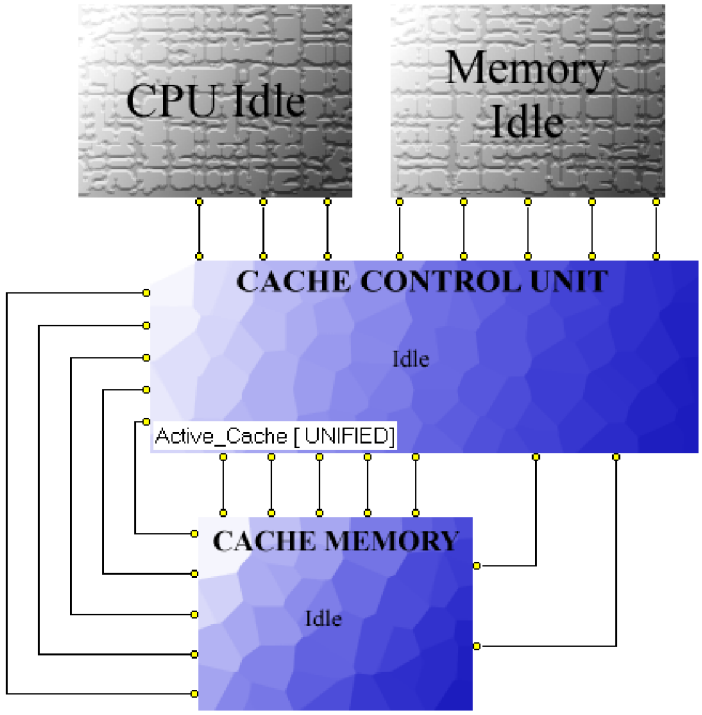


Figure 5.1: The Cache Architecture at Simulation Time-Step 1

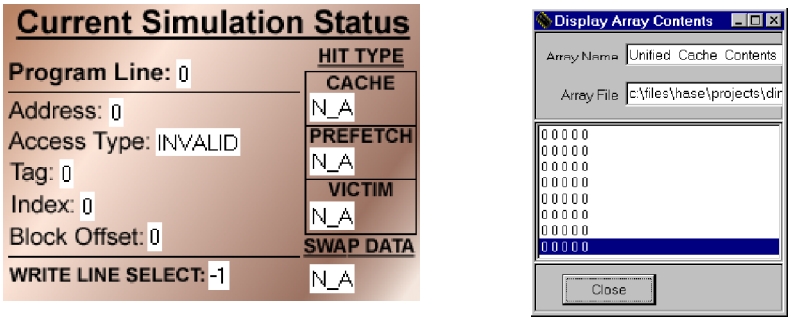


Figure 5.2: The Status Window and Unified Cache Contents Window at Simulation Time-Step 1

The tutorial entity is displaying a message introducing the features to be demonstrated in the first tutorial and ensuring that the unified cache contents window has been opened by the user.

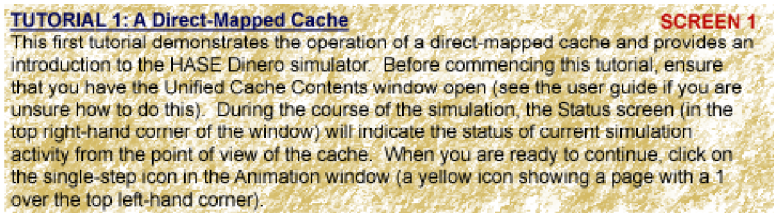


Figure 5.3: The Tutorial Entity Screen at Simulation Time-Step 1

Tutorial one demonstrates not only the operation of a direct-mapped cache but also how HASE Dinero describes current simulation activity. There are twenty-four discrete simulation states in the first tutorial and twelve separate tutorial entity states, thus a complete description of the tutorial would require more space than can be justified for this section of the report. Instead, only the first two steps of the tutorial are shown so that an overview of tutorial operation can be gained.

Tutorial 1: Simulation Time-Step 2

The simulation time has been advanced by one unit in response to the user clicking on the single-step button on the simulation Animator window (see **section 5.3.1.3**).

The screen dump of simulation activity in figure 5.4 was taken whilst the CPU was in the process of sending a data read request to the cache control unit. Note that the CPU state has changed to indicate the current activity.

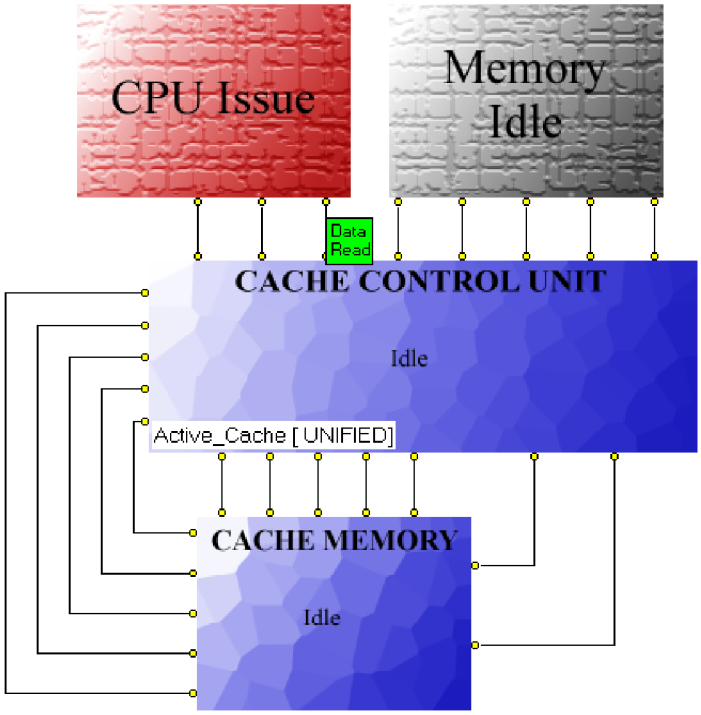


Figure 5.4: The Cache Architecture at Simulation Time-Step 2

The only change that has taken place for the status window (figure 5.5, left) is that the status of the cache hit field (top-right) has changed from N_A (not applicable) to WAIT. The cache hit field will remain in the WAIT state until the cache control unit entity has determined whether a cache hit has occurred for the current CPU address request, at which point either YES or NO will be displayed.

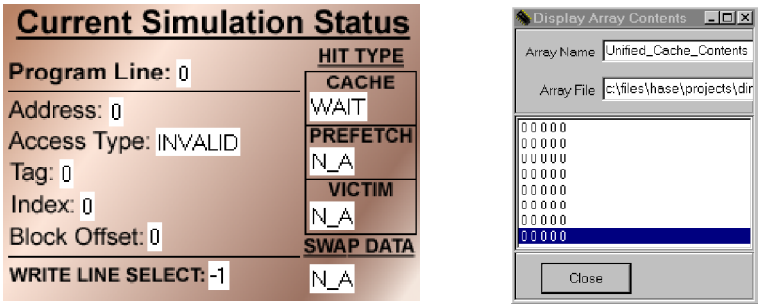


Figure 5.5: The Status Window and Unified Cache Contents Window at Simulation Time-Step 2

The tutorial entity is now displaying screen two (figure 5.6), which provides a textual description of the simulation activity that has just occurred.

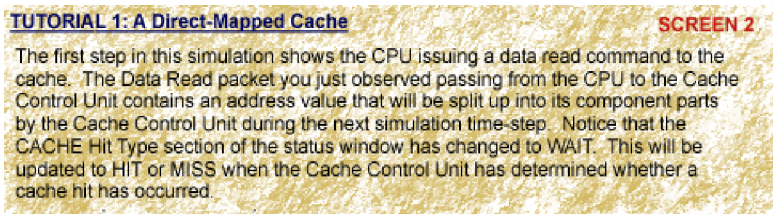


Figure 5.6: The Tutorial Entity Screen at Simulation Time-Step 2

The remaining eleven tutorials each demonstrate a specific feature of cache architecture. Wherever possible, the same input trace file is used for similar tutorials (for example: the tutorials demonstrating LRU, FIFO and random replacement), so that the student can clearly observe the differences caused by varying the relevant cache policy. The additional tutorials implemented are listed below:

- **Tutorial 4: The Benefit of Increased Block Size**
- **Tutorial 5: The Operation of a Split Instruction / Data Cache**
- **Tutorial 6: A Demonstration of LRU Replacement**
- **Tutorial 7: A Demonstration of FIFO Replacement**
- **Tutorial 8: A Demonstration of Random Replacement**
- **Tutorial 9: The Copy-Back Write Policy**
- **Tutorial 10: The Write-Through, Write-Allocate Policy**
- **Tutorial 11: The Write-Through, No-Write-Allocate Policy**
- **Tutorial 12: The Always-Prefetch Fetch Policy**
- **Tutorial 13: The Miss-Prefetch Policy**
- **Tutorial 14: The Benefit of a Victim Cache**

There are certain features of HASE Dinero that are not mentioned during the course of the tutorials, but these features have been purposefully omitted for the reasons discussed below.

The most important omission from the tutorials is an explanation of the simulation results screen (**figure 5.9**). This explanation was omitted because I felt that the results screen itself is sufficiently clear that to explain the results which are meaningful in the context of the tutorials (for example: the cache hit ratio and breakdown of access type) would be unnecessary. The simulation results

that are not easy to understand (for example: the cache utilisation and miss breakdown) would be difficult to describe in a tutorial and so I decided to include the description of such results in the user-guide.

In addition, the following cache features were not demonstrated in a tutorial as I felt that that the features were sufficiently conceptually simple that an animated demonstration was not required (all of the features listed below are introduced in the tutorials, but the effect of varying the parameter is not described):

- Variable Cache Size
- Variable Prefetch Stride
- Variable Prefetch Buffer Size
- Variable Victim Cache Size

It was not possible to produce a tutorial to demonstrate sub-block placement as sub-blocks have not been implemented in Animated mode, and also I decided that a demonstration of the *flush count* feature of HASE Dinero was not required as it is both easy to understand and unlikely that a student would use the feature.

5.2.2 User-Guide Design

The HASE Dinero user-guide has been implemented in HTML and can be found on-line at internet address **http://www.dcs.ed.ac.uk/~hase/projects/dinero/index.html**. The user-guide was designed to address two main concerns. Firstly, to provide an introduction to HASE Dinero together with details on how to use the HASE interface. Secondly, to provide an introduction to the field of cache design so that computer science students with limited knowledge of computer caches can use HASE Dinero to gradually build up a detailed understanding.

There are four main areas in the user-guide, each of which is described below.

The Tutorial Section

The HASE Dinero tutorials are designed to be the first part of the cache simulation system to be used by an inexperienced cache design student. As such, the tutorial section provides details not only on how to use the HASE interface to run the tutorials, but also provides hyperlinks to other sections of the user-guide containing information crucial to the understanding of the tutorial cache simulation:

- The format of data in the cache contents windows.
- The format of data in the prefetch buffer and victim cache windows.
- An explanation of how each CPU address is split into separate tag, index and block-offset components depending on the current cache size, associativity and block size.

In addition, a hyperlink to a description of the principle of locality is provided, to ensure that students understand the reason why computer caches are used in the memory hierarchy.

The tutorial section of the HASE Dinero user-guide has been included in appendix A of this report.

The Point and Click Guide

This section of the website is intended to assist a student who is unclear on some aspect of the HASE Dinero simulator. A screen-dump of the HASE Dinero user interface is provided. Each region of the user interface picture has a hyperlink pointing to information describing all relevant information about the appropriate section.

The point and click guide was designed to be an easy-to-use help guide that would allow students to quickly access the desired information by an intuitive point and click mechanism. The information provided for each section contains, for example, details on the input parsing limits imposed by HASE Dinero and the format of data produced during the course of simulation.

The Guide to Use Section

This section of the website provides detailed information on how to install HASE Dinero onto a Windows NT system, and describes how to run a sample cache simulation. In addition, four subsections are provided to give useful background information on both the operation of HASE Dinero and computer caches in general:

- How the CPU address is split into separate components.
- The principle of locality.
- How the cache utilisation is calculated in HASE Dinero.
- How the miss breakdown is calculated in HASE Dinero.

Clearly, each of the above sections addresses comprehensibility issues with HASE Dinero that have been introduced during the course of this report.

The Background Section

Finally, the background section of the user-guide provides an overview of the history of HASE, Dinero, and HASE Dinero.

5.3 The Final User Interface Model

This section of the report introduces the user interface of the HASE Dinero simulator. Firstly, a description of the interface is given followed by an assessment of the quality of the interface using Human Computer Interaction (HCI) evaluation techniques. The final part of this section describes the modifications made to the interface as a consequence of the feedback obtained during testing.

HCI can be defined as “the discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them” [6]. HCI is a relatively new field in computer science; although it has existed as a professional community since 1982 the first steps in codifying the field have only begun in the past few years.

5.3.1 User Interface Design

Throughout the process of designing the HASE Dinero user interface, HCI usability principles were used to guide the design decisions taken. There are three major usability principles that have been codified in the HCI field, these are listed below:

- **Learnability**: the ease with which new users can begin effective interaction and achieve maximal performance.

- **Flexibility**: the multiplicity of ways the user and system exchange information.

- **Robustness**: the level of support provided to the user in determining successful achievement and assessment of goals.

Of the three main classes of usability principle described above, the flexibility guidelines could not be applied to the HASE Dinero user interface design due to the constraints imposed by the HASE interface. Examples of specific flexibility guidelines include:

- **Dialogue Initiative**: allowing the user freedom from artificial constraints on the input dialogue imposed by the system.

- **Multi-Threading**: the ability of the system to support user interaction pertaining to more than one task at a time.

Clearly, the above two guidelines apply to the actual HASE interface and the operating system, respectively, not the actual HASE Dinero part of the user interface.

However, the two other main classes of usability guidelines (learnability and robustness) each contain several guidelines that were relevant to the HASE Dinero user interface design. The guidelines that I used to assist in designing the user interface are listed below:

- **Learnability**

    1. **Predictability**: support for the user to determine the effect of future action based on past interaction history.

    2. **Synthesizability**: support for the user to assess the effect of past operations on the current state.

    3. **Familiarity**: the extent to which a user's knowledge and experience in other real-world or computer-based domains can be applied when interacting with a new system.

- **Robustness**

    1. **Observability**: ability of the user to evaluate the internal state of the system from its perceivable representation.

    2. **Recoverability**: ability of the user to take corrective action once an error has been recognised.

HASE itself enforced the use of a direct-manipulation WIMP interface together with all associated pre-designed HASE menus. As a consequence of this, the main elements of the HASE Dinero user interface that could be customised to provide good usability characteristics were as follows:

- **The arrangement of controls and displays on-screen.**

- **The use of colour in the interface.**

The remainder of this part of the report describes the actual user interface used in HASE Dinero. Figure 5.7 shows the user interface in its initial state when HASE Dinero is first loaded.

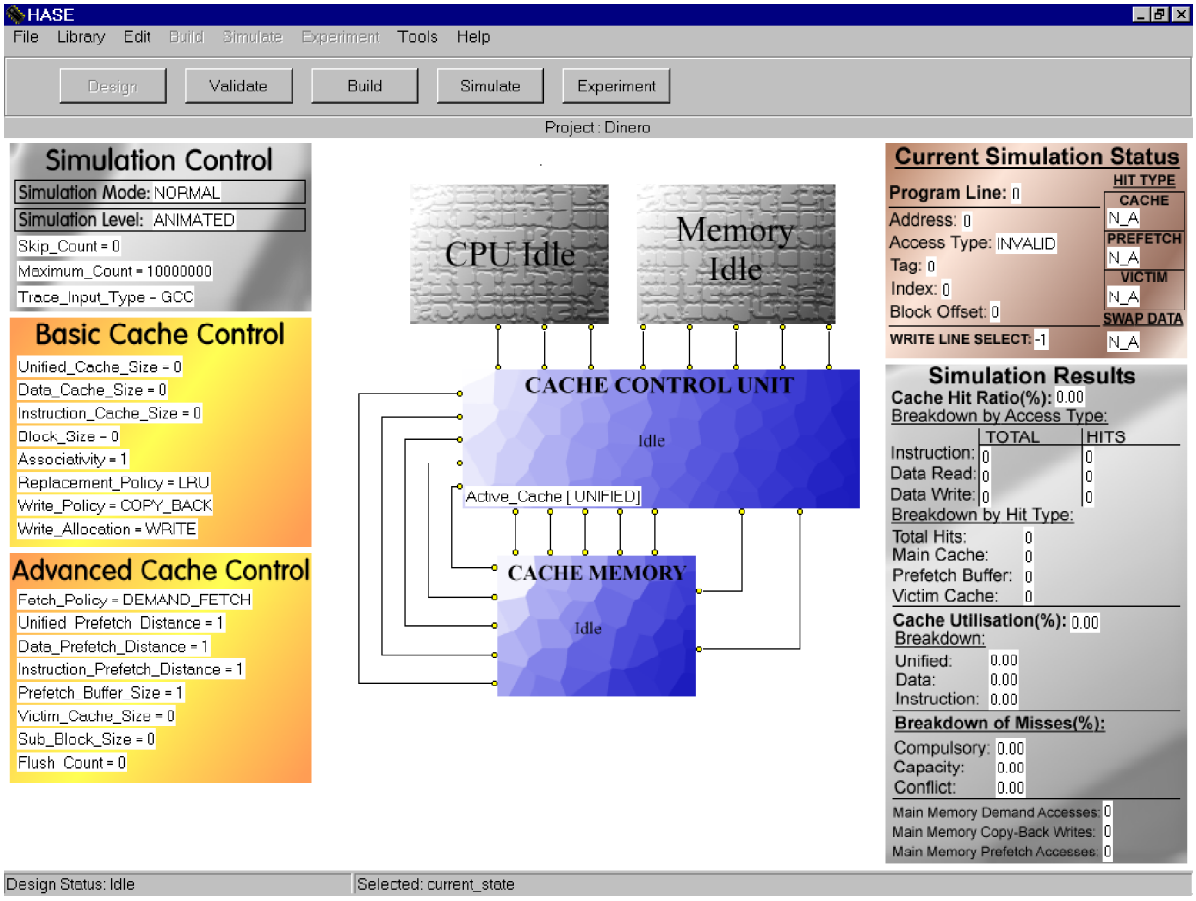


Figure 5.7: The Initial State of the HASE Dinero User Interface

5.3.1.1 User Interface Input Sections

1. **Simulation Control Screen:**
   The user interface simulation control screen allows parameters controlling the simulator itself to be modified by the user, for example: the number of trace file references to process, and which input trace file to use.

2. **Basic Cache Control Screen:**
   The basic cache control screen allows modification of the basic HASE Dinero cache configuration parameters (as described in **section 4.2**).

3. **Advanced Cache Control Screen:**
   The advanced cache control screen allows modification of the advanced HASE Dinero cache configuration parameters (as described in **section 4.3**), as well as the victim cache parameter.

The colour coding of the user interface input sections was chosen to clearly distinguish between the different type of input parameters that can be modified.

5.3.1.2 User Interface Output Sections

The HASE Dinero output screens are only active during the playback of a simulation results trace file. Each output screen provides information on a specific aspect of the cache simulation.

1. The Current Simulation Status Screen

The simulation status screen is only active if the simulator is playing back a results trace file produced using Animated mode. Figure 5.8 shows an example of the contents of the current simulation status window during simulation playback.

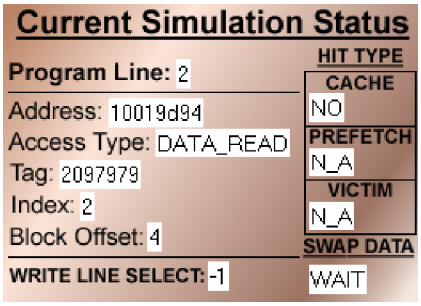


Figure 5.8: The Current Simulation Status Screen

The content of the current simulation status screen is described below:

1. **Program Line**: the line number of the current trace file reference being processed.

2. **Address**: a hexadecimal representation of the current CPU address being serviced by the cache.

3. **Access Type**: a string representing the type of the current CPU access, which can be any of:
   - **Invalid** (for example: a corrupt custom trace file)
   - **Data Read**
   - **Data Write**
   - **Instruction Fetch**
   - **Flush Cache**

4. **Tag, Index, Block-Offset**: a decimal representation of the breakdown of the current CPU address (performed by the cache control unit).

5. **Write Line Select**: when the cache control unit is in the process of writing a new block to the cache memory array, or updating a cache line, this field indicates which cache line will be modified.

6. **Hit Type (Cache, Prefetch, Victim)**: each of these fields is initialised to one of N_A (not applicable) or WAIT at the start of the processing of each CPU trace file reference, depending on whether the relevant cache architectural feature is enabled in the current cache configuration (the *cache* component is obviously always active). When the cache has determined whether a cache hit has occurred, one of the fields will be updated to show YES, and any other fields showing WAIT will be updated to show NO.

7. **Swap Data**: this field is initialised to WAIT if the cache write policy is set to copy-back, and N_A otherwise. If the *swap data* field is current displaying WAIT and a modified cache line is chosen for replacement when a new block is loaded to the cache, then the field is updated to show YES. If a cache replacement is not required, or a non-modified line is chosen for replacement then the field is updated to show NO.

2. The Simulation Results Screen

The simulation results screen provides dynamic feedback on cache performance during playback of a results trace file produced using Animated mode. If a Fast mode trace file is played back then the simulation results screen shows only the final results produced at the end of simulation.

Figure 5.9 shows an example of the simulation results screen produced from a Fast mode simulation execution.

One of the main usability features of the results screen is that the information displayed is easy to understand. As a consequence, an explanation of the results screen contents is not provided here. The only results screen information that is not intuitive is both the cache utilisation and the breakdown of misses, for which an explanation is provided in **section 4.4**.

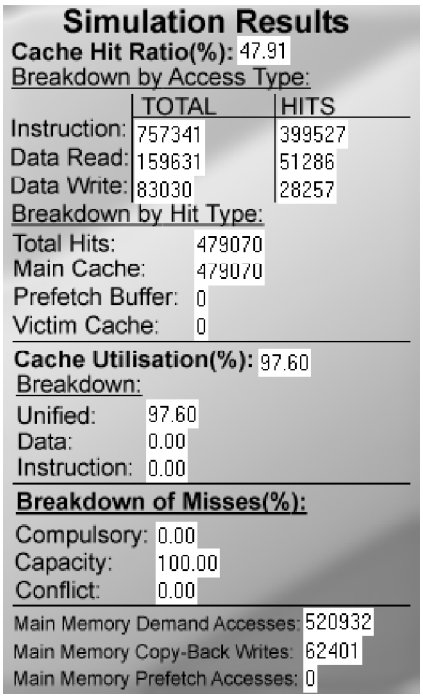


Figure 5.9: An Example of Information Displayed in the Simulation Results Screen

In addition to the five main input and output screens, the HASE Dinero user interface also contains three other features relevant to the interface usability:

- The HASE Dinero Animation System
- The HASE Array Contents Windows
- The Feedback Mechanism

Each of the above three features is described in more detail in the ensuing sections.

5.3.1.3 The HASE Dinero Animation System

There are two aspects to the animation system in HASE Dinero. Firstly, HASE provides an animation window which provides the simulation user with a high degree of control over the playback of the results trace file. The Animator window is shown in figure 5.10.

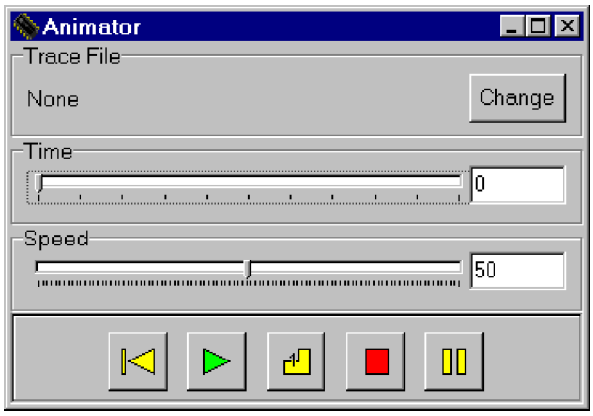
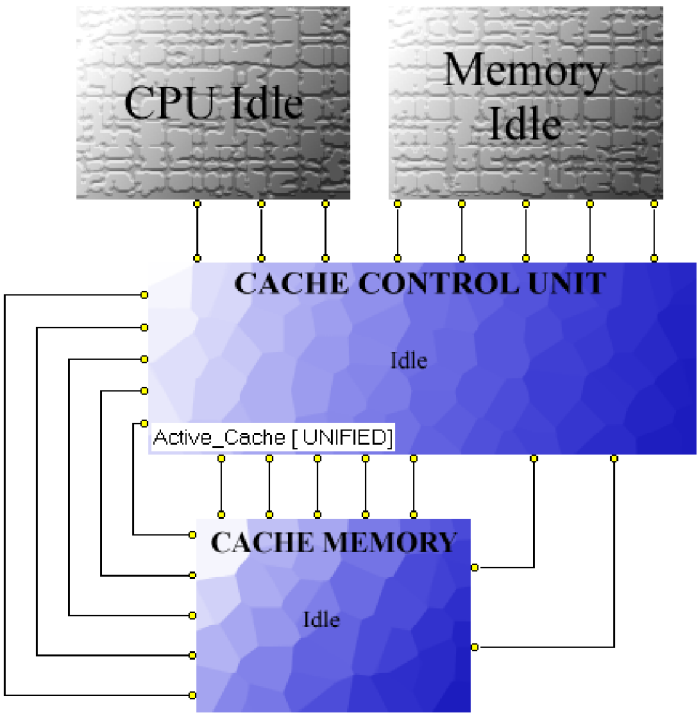


Figure 5.10: The HASE Animator Window

From left to right, the functionality of the Animator window buttons is as follows:

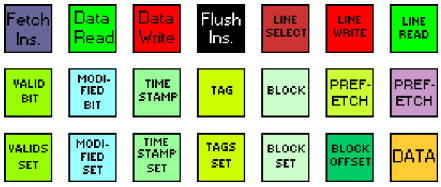
- **Rewind**: resets the current simulation playback to simulation time zero.
- **Play**: instructs HASE to playback the simulation at the speed specified by the *speed* slider bar.
- **Single-Step**: instructs HASE to playback all animation occurring during the next simulation time-step, and then pause the simulation.
- **Stop**: instructs HASE to stop processing the results trace file at the end of the current simulation time-step.
- **Pause**: instructs HASE to freeze all screen activity in its current state.



In addition to the above functionality, the Animator window also provides a simulation time slider, which allows the user to freely move though the available simulation time-steps.

The Animator window is an in-built part of the HASE interface and therefore not part of the HASE Dinero user interface design. However, the animation shown on screen during Animated mode results trace file playback is clearly one of the most important aspects of the HASE Dinero project. Figure 5.11 shows the on-screen cache architecture representation used in HASE Dinero.

Figure 5.11: The User Interface Cache Architecture Representation



During the course of display animation, the flow of data through the cache architecture is represented by the movement of packets. HASE Dinero has a wide range of packets used during the course of animation, shown in Figure 5.12.

Figure 5.12: The HASE Dinero Animation Packets

There is an extremely large number of types of animation that can be visualised by the HASE Dinero cache architecture. As a consequence, I shall provide an example of animated HASE Dinero activity that can be used to understand how the visualisation of the animation system has been implemented, and thus applied to all other possible cache activity by an inductive process. The example given below shows how a CPU data read request is animated using the HASE Dinero cache architecture. Figure 5.13 shows the CPU sending a data read packet to the cache control unit. Note that the CPU status has changed from *idle* to *issue*, clearly indicating the current activity. Figure 5.14 shows the cache control unit sending a line read request to the cache memory array. Note that the cache control unit image has vividly changed state to indicate the current activity. Figure 5.15 shows the cache memory array sending the contents of the requested associative set to the cache control unit. During the course of simulation playback the simulation status screen provides information on the current cache status, displaying all of the additional information required by the simulation user to understand current cache activity.

In this example, the cache memory array does not contain the required data and so a cache miss occurs. As a consequence, the cache control unit requests the required block from main memory (figure 5.16). Main memory sends the required block to the cache control unit in the next simulation time step (figure 5.17). The cache control unit now sends the requested data to the CPU and writes the newly loaded block to a free cache line (figure 5.18).

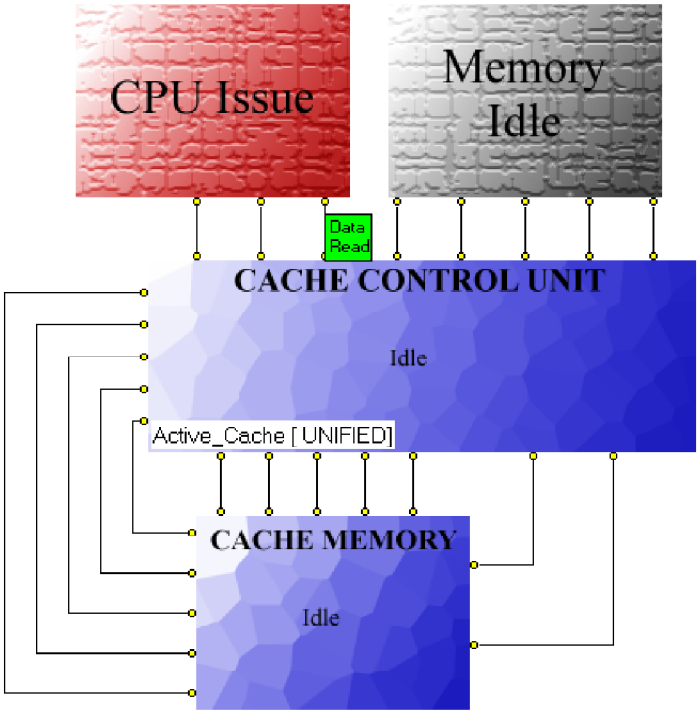


Figure 5.13: The CPU Issuing a Data Read Request

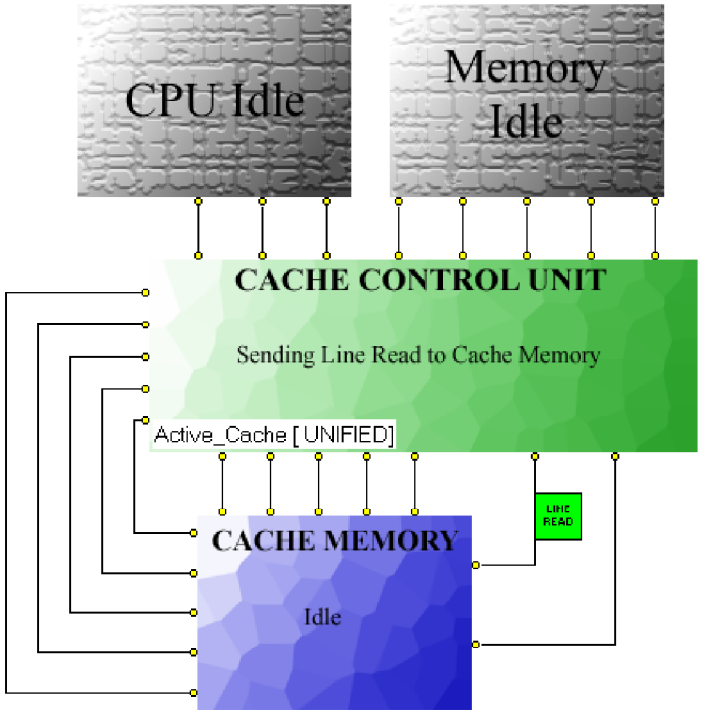


Figure 5.14: The Cache Control Unit Issuing a Line Read Request to the Cache Memory Array

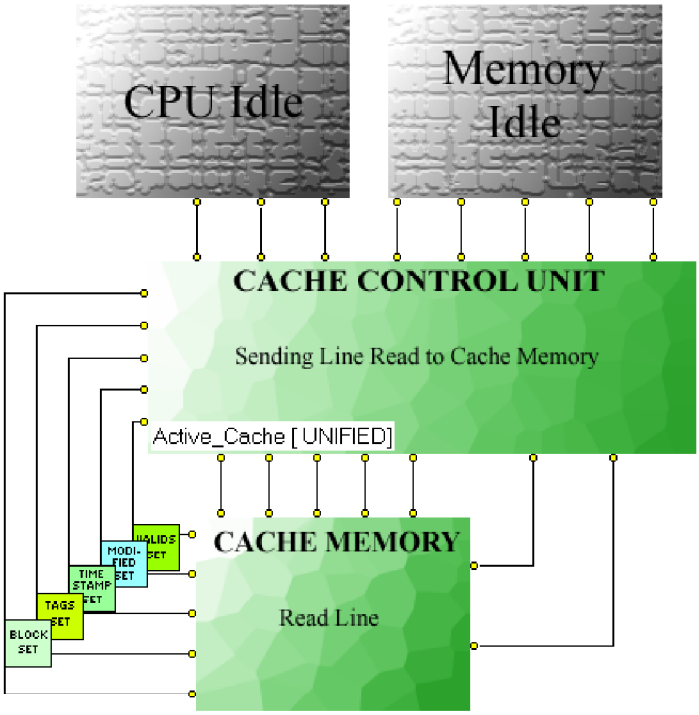


Figure 5.15: The Cache Memory Array Sending the Contents of an Associative Set to the Cache Control Unit

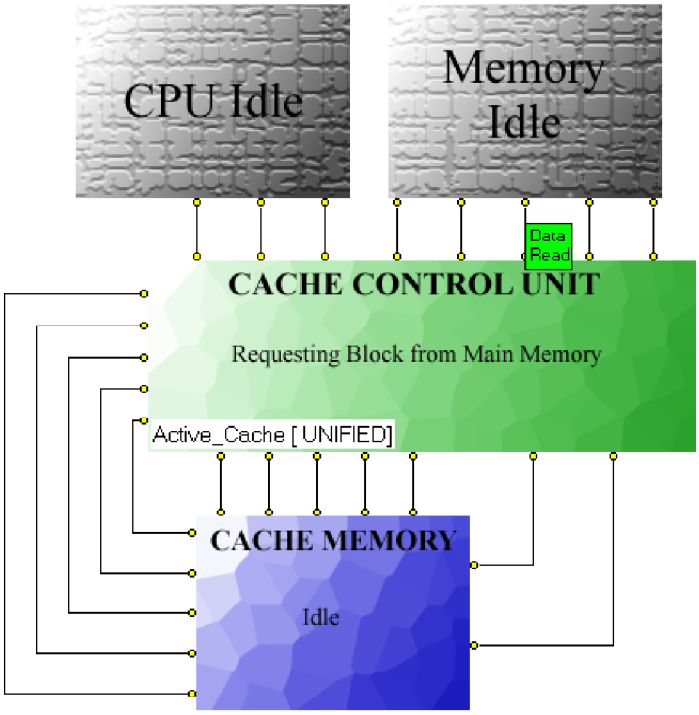


Figure 5.16: The Cache Control Unit Issuing a Block Read Request to Main Memory

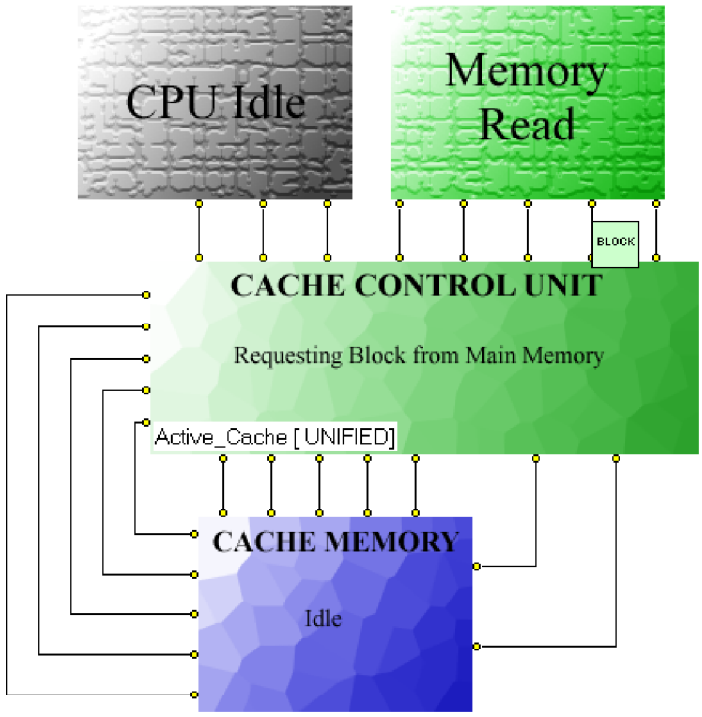


Figure 5.17: Main Memory Sending a Requested Block to the Cache Control Unit

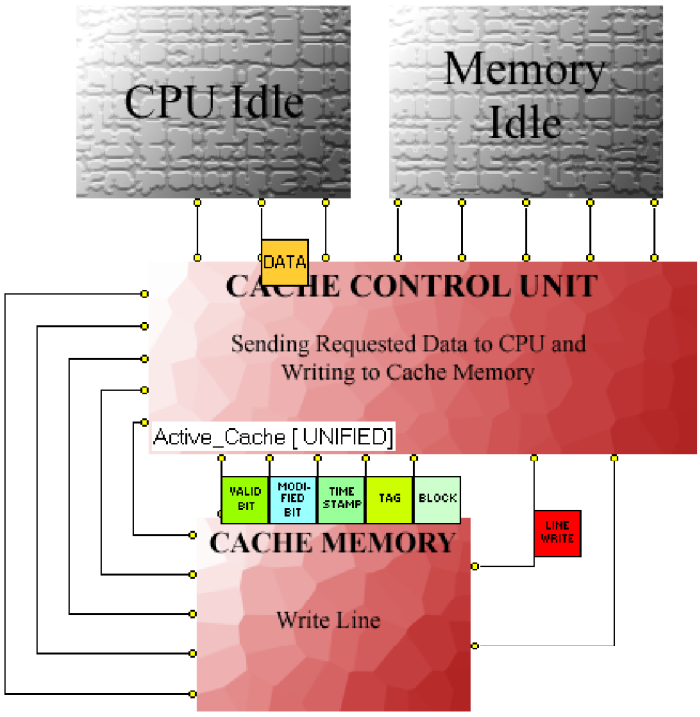


Figure 5.18: The Cache Control Unit Sending the Requested Data to the CPU and Writing a Block to the Cache Control Unit

A final point to mention here is that the cache control unit has twenty five possible different states, each of which is vividly represented in colour to assist the simulation user in understanding current cache activity; as an example, two possible states are shown below.

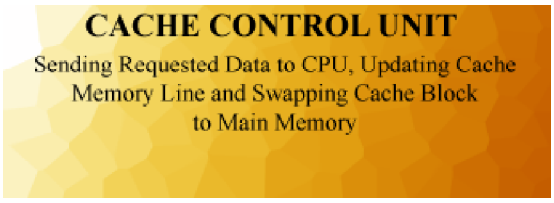


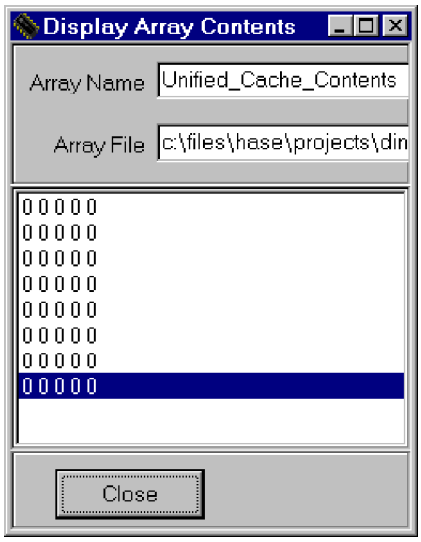
Figure 5.19: Cache Control Unit State Example 1

Figure 5.20: Cache Control Unit State Example 2

The example just given, describing how cache activity is animated, can be applied to any situation encountered during a HASE Dinero cache simulation animation.

The next section explains why the HASE array contents windows are essential for an understanding of cache activity to be gained.

5.3.1.4 The HASE Array Contents Windows



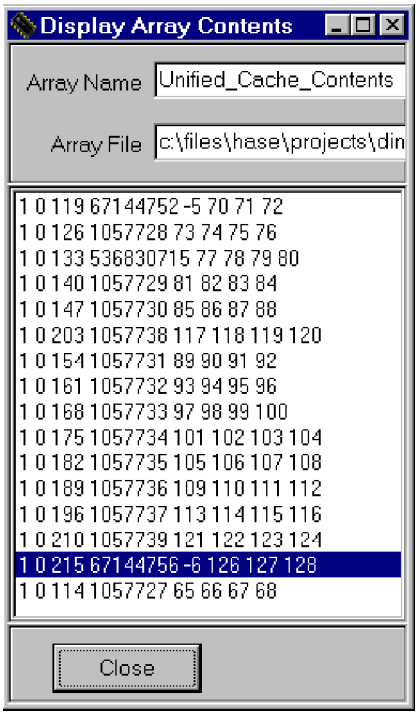
HASE allows the updates applied to arrays during the course of a simulation execution to be written to the results trace file. In addition, HASE provides array contents windows that can be selectively opened by the user for the course of a simulation animation. An example of a HASE array contents window for a unified cache with a block size of one byte is shown in figure 5.21.

An example of the HASE array contents window for a small cache that has been completely filled during the course of simulation is shown in figure 5.22.

Figure 5.21: The HASE Array Contents Window for a Unified Cache

There are five array contents windows available in HASE Dinero:

- Unified Cache Contents
- Data Cache Contents
- Instruction Cache Contents
- Prefetch Buffer Contents
- Victim Cache Contents



The three cache array windows each have the same data format as figure 5.21, however the prefetch buffer and victim cache each have a different data format to reflect the altered storage requirement.

The HASE array contents windows are essential for an understanding of simulation activity to be gained. By observing the contents of the array content windows and the current simulation activity it is possible to both exactly trace the current cache activity and, when familiar with the simulation environment, to predict the outcome of the next few simulation steps.

Figure 5.22: The HASE Array Contents Window for a Full Unified Cache

An additional feature of HASE that makes following simulation activity easier is that each HASE array window highlights the last line accessed by the simulator and, if the array contents require more space than the array contents window size, the window will automatically scroll to the latest accessed location before highlighting the appropriate line.

5.3.1.5 The User Interface Feedback Mechanism

HASE Dinero incorporates a simple feedback mechanism which is used for two main purposes. Firstly, to provide error feedback to a user if the cache configuration that has been specified for simulation is invalid, and secondly to indicate the current simulation status during animation of the display window.

The feedback mechanism was implemented as a HASE entity for which the current state is displayed on screen as a picture. The pictorial representations of all possible states for the feedback entity are shown in figure 5.23.

Figure 5.23: All Possible States for the Feedback Entity

Each time a simulation is executed, the first step taken is to check that the cache configuration specified by the user is valid. If an invalid cache configuration was allowed to be simulated then not only would the results produced be meaningless but it is quite likely that illegal memory accesses would occur during the course of simulation. An example of an illegal cache configuration would be to specify a unified cache size of 64 bytes together with a block size of 128 bytes.

A complete listing of the validity checks that are carried out before simulation execution is provided in appendix B. In the event that an error is detected then the simulation is cancelled and an error message is written to the results trace file. Unfortunately, HASE does not allow spaces to be inserted into text messages in the results trace file, and so spaces are represented by underscores in error messages. When the results trace file is animated, the error message will be displayed in the feedback entity screen. An example of an error message is shown in figure 5.24.

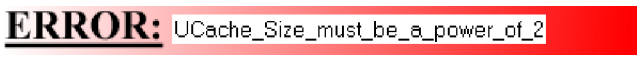


Figure 5.24: An Example Feedback Entity Error Message

5.3.2 User Interface Evaluation

HCI-based user interface evaluation usually has three main goals [6]:

1. **To assess the extent of the system's functionality.**

2. **To asses the effect of the interface on the user.**

3. **To identify any specific problems with the system.**

HASE Dinero was not developed in response to a task-oriented user need and, as a consequence, it is not possible to asses the extent of the system's functionality: what metrics could be used to evaluate the extent to which the vague goal of "implementing an effective cache learning environment" has been achieved? It would have been possible to perform a comparative evaluation between HASE Dinero and the original Dinero, but I saw little point to this as the original Dinero was developed to provide a quick simulation tool, not a teaching aid, and so a comparison would not provide useful information.

However, the remaining two evaluation goals are clearly applicable to HASE Dinero and so could be taken into account whilst I was developing the user interface testing plan. A major concern in the design of the user interface testing plan was to separate the HASE user interface from the Dinero interface actually implemented during the course of the project. At an early stage in the project I decided to not directly modify the HASE interface (which would have made it easier to use HASE Dinero). If I had decided to modify the HASE interface then I could have completely removed the distinction between HASE and the HASE Dinero project, instead producing a complete Dinero simulation application, based on the initial HASE design. This would not have required a great deal of development effort as NT HASE was developed using Borland C++, a graphical programming tool, which would have facilitated rapid modification of the HASE interface to a completely customised form. I decided not to modify the HASE interface during the course of the project because this would have completely isolated HASE Dinero from the current set of simulations that have been written as HASE projects. As a consequence, HASE Dinero can be used by anyone with a standard installation of NT / Linux HASE.

In terms of the user interface evaluation, HASE imposed three main features onto the HASE Dinero interface that needed to be carefully excluded from the user interface testing plan. These three features are listed below:

- **The HASE control interface**: includes features such as the methodologies used to load projects into HASE, run simulations and load the results trace file into memory.

- **The HASE Animator window**.

- **The HASE array contents windows**.

It was clear that the testing plan developed should consider only the Animated mode of HASE Dinero not only because, in terms of the user interface, Animated mode provides a superset of the functionality of Fast mode, but also because I intended to use computer science students without prior knowledge of cache architecture as experimental subjects. I decided to use students without detailed subject area knowledge so that I could assess both the teaching capability of HASE Dinero,

which would not have been possible if subjects with detailed cache architecture knowledge had been chosen, and any effects that the user interface might have on the teaching environment.

Taking into account the points previously described, I produced the following goal for the HASE Dinero user interface evaluation:

The test plan should aim to determine two usability factors of the HASE Dinero user interface:

1. How useful HASE Dinero is as a teaching tool, in terms of the information displayed on screen, and specifically considering the animation system used.

2. The effect that the HASE Dinero colour scheme has on the learning environment: does it enhance or detract from the user interface clarity?

5.3.2.1 The HASE Dinero User Interface Test Plan

As the HASE Dinero user interface is fairly unique, I did not have the benefit of knowing in advance precisely which usability aspects I would need to test. As a consequence, the main testing technique I chose was a form of interview-based evaluation, **critical incidents and breakdowns**, in which test subjects are observed using the system and any interface problems encountered are classified into two categories:

1. A **critical incident** occurs when a test subject is unsure which action is required in order to achieve the current goal, or perhaps performs an incorrect action in attempting to achieve the goal.

2. A **breakdown** happens if a test subject becomes conscious of the system interface during the course of performing an action, perhaps because the interface is not as intuitive as it might be.

The primary test plan consisted of a forty five minute interview for each test subject, during which time the subject was introduced to the field of computer caches and shown how to use HASE Dinero. Following the introduction and demonstration, the subject was asked to simulate a cache configuration, different from the configuration previously demonstrated, and then asked to provide a verbal description of simulation activity during the course of results trace file playback.

As test subjects were selected on the basis of being fourth year computer science students and not having detailed knowledge of computer caches, the test plan outlined above provided a good way to evaluate the benefit of HASE Dinero as a teaching tool.

The demonstration stage of the interview consisted of providing the subject with a verbal description of how CPU addresses are split into tag, index and block-offset components, together with a description of how the cache associativity affects this split. The verbal explanation was accompanied by a printed copy of the CPU address breakdown description from the HASE Dinero user guide.

When I had decided that the test subject had a satisfactory knowledge of the basic operation of computer caches, I demonstrated how HASE Dinero could be used to perform a simple Animated mode cache simulation. During animation of the results trace file playback for the first three input trace file references I provided a verbal explanation of cache activity.

Once the initial demonstration stage of the interview was completed, I provided the subject with an experiment sheet which both specified a cache configuration to be simulated and asked the subject to verbally describe the simulation activity as the results trace file was played back. This allowed me to perform a critical incidents and breakdowns evaluation of the HASE Dinero input screens, output screens and animation system. The experiment sheet provided to the test subjects is shown in appendix C, whilst the results of the critical incidents and breakdowns evaluation are discussed in the next section.

The second stage of the user interface testing plan was to provide each test subject with a questionnaire to be completed after the interview. The questionnaire provided a mechanism to ask specific questions, the results for which could be quantitatively analysed once all experiments were complete. The questionnaire also allowed me to target specific aspects of the HASE Dinero user interface that I was particularly interested in evaluating. The questionnaire given to test subjects is shown in appendix D together with the questionnaire results.

I decided to perform the user interface test procedure with five test subjects. I would have liked to perform the experiment with more subjects, but time constraints and the availability of willing candidates did not allow for this.

The combination of both a critical incidents and breakdowns interview and a questionnaire resulted in a test plan for which both unexpected problems with the interface could be discovered and specific areas of the interface could be specifically targeted for evaluation.

5.3.2.2 The Results of the User Interface Evaluation

I found that the feedback provided from both the questionnaires and the interviews was, for the most part, favourable and believe that it can be concluded from this that HASE Dinero has achieved its primary design goal of providing a computer cache teaching environment.

However, there were eight specific problem areas with the HASE Dinero user interface that were identified during the course of the evaluation. These user interface issues are listed below, together with the proposed remedial action:

1. The most significant problem that caused difficulty with understanding the HASE Dinero screen animation was that the HASE array contents windows are not easy to understand. There were two reasons given for this: firstly because each column of data does not have a title, and secondly because the columns of data become misaligned as data values with varying numbers of digits are stored in each row (see figure 5.22). As this is a problem with the HASE interface itself, I did not attempt to solve this usability problem.

2. Two of the test subjects found that it was difficult to read the text in the CPU fetch instruction packet. As this is primarily due to the choice of background colour for the packet, a simple solution was to change the background colour so that there is more contrast with the foreground text. The original and revised instruction fetch packets are shown in figure 5.25.

Figure 5.25: The Original and Modified Instruction Fetch Packet

3. Whilst all of the test subjects found that the vivid colour changes of the cache control unit assisted in their understanding of the simulation animation, it was commented that the scheme would be more beneficial if the meaning of the colour of each state was explained. As I had in fact used a colour coding scheme for cache control unit states (where, for example, green represents that data is being read, and red that data is being written), it required little effort to provide a guide to the meaning of the cache control unit colours in the HASE Dinero user-guide.

4. Several test subjects commented that the choice of texture for the CPU and main memory entities caused a clarity problem for the interface as it was felt the texture was completely unrelated to the actual components represented. The solution to this problem was to replace the CPU and main memory texture with textures that do represent the functionality of each component. The original and revised CPU and main memory textures are shown in figure 5.26.

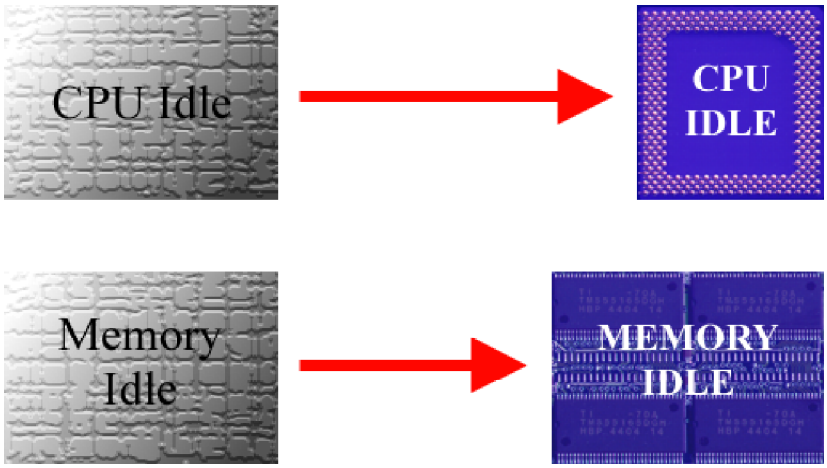


Figure 5.26: The Original and Revised CPU and Main Memory Textures

5. The basic and advanced cache control screens are always visible in the HASE Dinero window, even when results trace file playback is occurring. Several test subjects mentioned that the complexity of the interface would be reduced if these windows were only shown when required (at the stage when cache configuration parameters are specified). It would be possible to move the entities for the basic and advanced cache control screens into a HASE *compentity*. This *compentity* could be expanded and contracted by the simulation user as desired, with the expanded view showing the two cache control screens, and the contracted view showing an expandable box (without any visible parameters). However, I decided that it would not be beneficial to most HASE Dinero users for this modification to be made for two main reasons. Firstly, new users might be confused by the HASE methodology required to expand and contract the *compentity* and thus a new usability problem could be introduced. Secondly, the cache configuration being simulated is always written to the results trace file, and so whenever a simulation is animated, the cache control screens indicate precisely which type of cache has been simulated.

6. Another issue that was raised with relation to the basic and advanced cache control screens was that it was found to be unclear which simulation control screen any given cache parameter is located in. Although it would be possible to place all of the cache parameters in just one entity, I found two main problems with this approach. Firstly, the sheer number of parameters on-screen in one control screen is difficult to understand. Secondly, the HASE parameter modification window is too large to fit onto the virtual desktop of the display (at a 1280 by 1024 resolu-

tion) if all of the parameters are modifiable from just one entity. Additionally, once a cache design student is familiar with the HASE Dinero environment, it should be clear which control window to look in to find any desired parameter.

7. An aesthetic issue raised by several test subjects was that the background textures chosen for the simulation control screen and the simulation results screen are the same, thus falsely suggesting that there is a link between the two screens. A simple solution to this problem was to change the texture used for the simulation control screen. The revised colour scheme uses metallic textures for the HASE Dinero output screens and plain textures for the input screens, thus clearly distinguishing the different purposes of the two classes of screen. The modified simulation control screen is shown in figure 5.27.

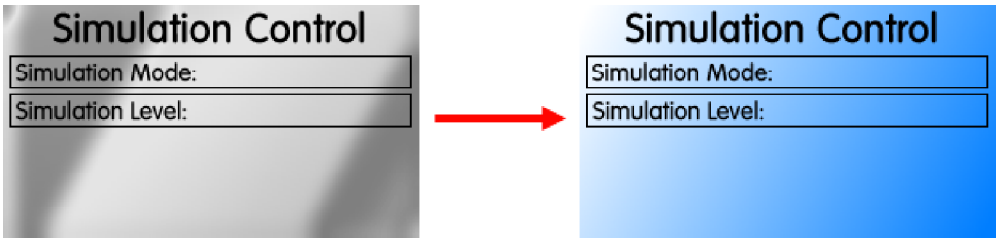


Figure 5.27: The Modified Simulation Control Screen

8. One final usability problem frequently mentioned was that the methodology required to instruct HASE to run a simulation, read the results trace file into memory and then animate the design window is over-complex and not intuitive. Whilst this is clearly a usability issue, as I have restricted the project so that I cannot modify HASE itself, this problem cannot be resolved within the scope of the HASE Dinero project.

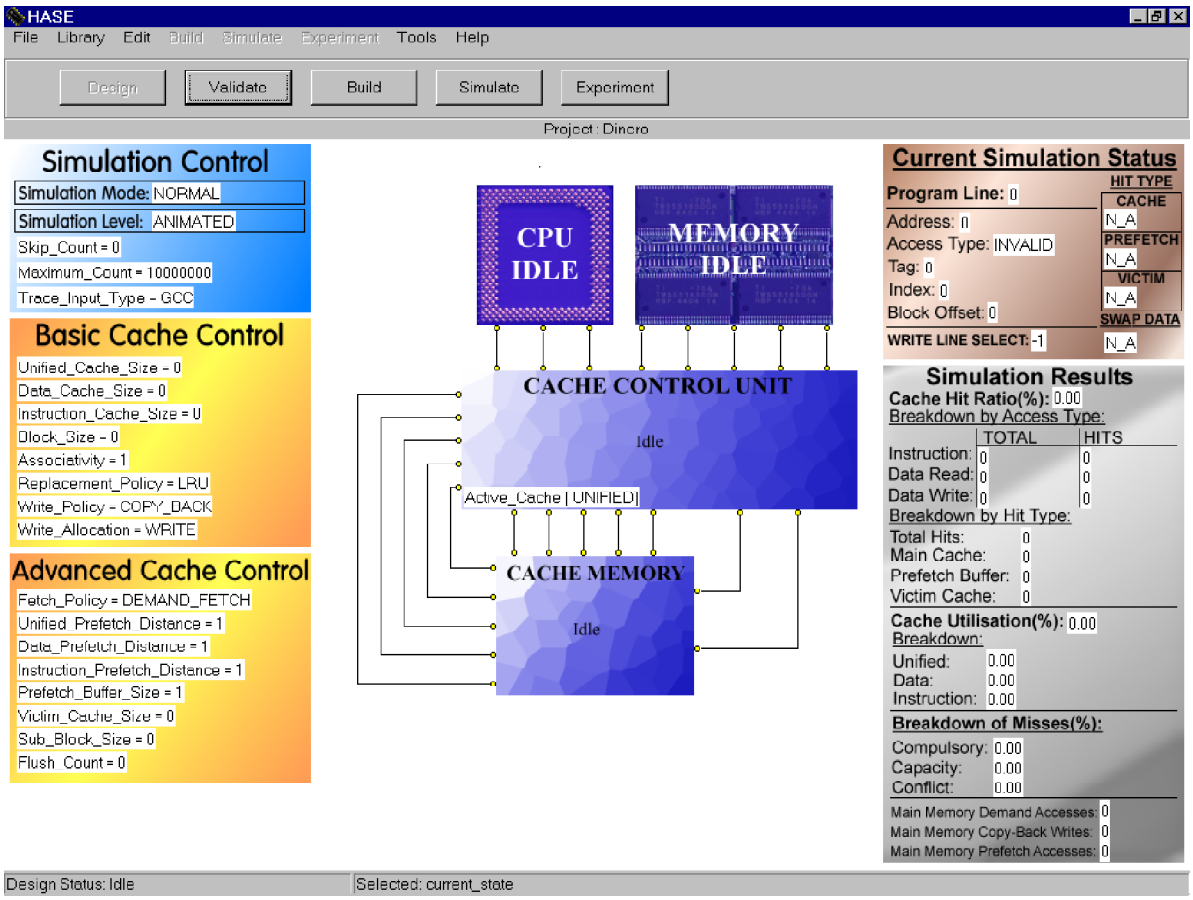


Figure 5.28: The Modified HASE Dinero Initial Window

6. Validation of the HASE Version of Dinero

6.1 Overview

The validation of HASE Dinero was an important aspect of the project due partly to the complexity of the cache simulation model, but mostly due to the high likelihood of there being errors in some parts of the four thousand lines of source code produced. Additionally, as HASE Dinero is intended to provide a teaching environment, an important aspect of the simulator quality was that it should produce correct results.

Unfortunately, software testing is an imprecise process with one particularly relevant quote being "testing can only show the presence of errors never their absence" [Dijkstra, 8]. In addition, it is widely held that only a probabilistic theory can describe the relationship between test measurements and product quality [8]. As a consequence of these facts, two main types of testing technique have been developed, each of which attempts to discover as many faults as possible in the program being tested:

1. Functional (Black-Box) Testing

This type of testing technique derives a test plan from a specification of the system functionality. The test plan ensures that every aspect of the system functionality is tested. To achieve this, all of the possible inputs to the system are divided into discrete sets, with several representative input combinations from each set being chosen such that every possible combination of input types is tested. It is important to note that this technique cannot test every possible input combination, for most programs, as the required test-set would be far too large. This technique is called black-box testing because the test plan does not take into account the source code for the program being tested. Functional testing lies at the heart of every reasonable testing plan [8].

2. Structural (Clear-Box) Testing

This type of testing technique requires the test designer to have access to the source code for the program being tested (hence clear-box testing). The test plan produced ensures that either every possible line in the program is tested or every function in the program is tested, depending on the testing granularity required, during the course of the test executions. This type of testing plan would be difficult and tedious to design by hand, and so complete structural coverage is usually ensured by producing a program that automates the structural testing procedure and ensures that complete coverage is applied.

The method used by both of the above techniques to actually test a program is to execute the tested program with certain input data combinations, and then to analyse the output data to check that it conforms to expectations. Neither of the above techniques can ensure complete program reliability because it is effectively intractable to check every possible combination of inputs for most programs. It is claimed that an effective test plan should incorporate both functional and structural testing techniques [8].

6.2 The Validation Test-Space Problem

HASE Dinero incorporates two separate modes of simulation (Fast and Animated), each of which has nineteen input parameters, of which only four are constrained to a limited range. In addition, the Fast mode simulator has twenty one output parameters and the Animated mode has thirty two output parameters, as well as the actual animation of the display. As a consequence, it should be clear that to test the entire input / output range of HASE Dinero was effectively an intractable problem within the time constraints of the project.

As a complete test of HASE Dinero was infeasible, I devised a test-plan using the functional testing technique which would ensure that coverage of all of the essential features of HASE Dinero was included, as well as providing limited coverage for the additional features of the simulators. Whilst I would ideally have liked to use both the functional-testing and the structural-testing techniques, the time constraint of the project did not allow for an automated test program to be produced to apply structural testing to the source code. The test plan used is described in the next section.

6.3 HASE Dinero Correctness Testing

Firstly, I decided that only the Fast mode of HASE Dinero should be tested against the original Dinero implementation, as both of the compared simulators could then be used to process one million lines of source code and thus any discrepancies between the simulation results would be clearly highlighted. After the Fast mode of HASE Dinero had been validated, the Animated simulation mode was validated against the Fast simulation mode, with both simulation modes being constrained to five hundred input trace file references (which is the maximum reference count for Animated mode).

I decided that only the cache hit ratio would be used to compare the results of the various simulators as the additional simulation results (for example: cache utilisation and the hit breakdown) are generated as additional bookwork in the course of calculating the hit ratio. Whilst this simplification of the output testing was not ideal, when the fact that ongoing testing took place during the course of HASE Dinero development is taken into account, the total coverage of the project testing procedure can be seen to include the additional parameters of the results screen, to a limited degree.

The verification of the Animated simulation mode against the Fast simulation mode was much simpler than the verification of the Fast simulation mode as a simple visual inspection could be used to determine whether the content of the HASE results window changed as the results of the Fast simulation execution were loaded, relative to the results of the Animated simulation initially displayed.

6.3.1 Fast Mode HASE Dinero Validation

The test plan I used to compare the results of Fast mode HASE Dinero with the original Dinero divided the simulation input parameters into two discrete sets. Parameters in the *first* input set were used in all possible combinations as input test-sets, thus providing complete coverage of the first input set. All possible combinations of the first input set were tested using both the GCC and spice input trace files, with one million trace file references being processed. Two different input

trace files were used to ensure that the same simulation results were not produced from one trace file by both simulators due to chance alone. Cache parameters were placed in the first input set on the basis of the parameter being a fundamental cache configuration feature, and also due to variance of the parameter causing completely different sections of simulation source code to be used. Note that all of the tests executed for the first test-set used a fixed cache size / block size.

The First Input Parameter Set:

1. Cache Associativity: Direct-Mapped, Eight-Way Set-Associative, or Fully-Associative.

2. Cache Replacement Policy: LRU, FIFO (Random not tested – results are expected to vary!)

3. Cache Write Policy / Cache Allocation Policy: Copy-Back & Write, Write-Though & Write, Write-Through & No-Write.

A total of thirty six tests were executed during the enactment of the test plan for the first test-set.

In addition to cache parameter combinations tested in the first test-set, non-essential cache configuration features were selected to be tested and placed in the *second* test-set. Parameters were placed in the second test-set on the basis of the parameter not being an essential cache configuration parameter and yet being available to be simulated in both HASE Dinero and the original Dinero. Each parameter in the second test-set was tested in isolation three times, using a different cache configuration randomly chosen from first test-set at each iteration.

The Second Input Parameter Set:

1. Separate Data Cache and Instruction Cache.

2. Block Size changed to 16 bytes from 32 bytes.

3. The Always-Prefetch Fetch Policy.

4. The Miss-Prefetch Fetch Policy.

5. The Forward-Prefetch Fetch Policy.

6. The Sub-Block-Prefetch Fetch Policy.

7. Sub-Block Placement.

There were two HASE Dinero input parameters omitted from the testing plan described above: *flush count* and *victim cache*. The flush count was omitted as its operation is so simple that a test is not required. The victim cache parameter cannot be simulated in the original Dinero and so could not be validated.

It would be pointless to show the results produced during execution of the above two test plans, as the only meaningful aspect of the results obtained is whether the hit ratio produced was the same for HASE Dinero as the original Dinero. During the course of Fast mode validation, three errors in the project source code were discovered and resolved. The corrective action taken was sufficiently simple that I did not feel that other simulation results would be affected (which would have required a repeat execution of the test plan).

During the course of testing Fast mode HASE Dinero against the original Dinero, one discrepancy was discovered that has not been resolved. This discrepancy is described below. All tests apart from the one described below produced identical results from both simulators.

I found that the results produced by HASE Dinero and the original Dinero varied very slightly when any of the prefetching policies were in use. Over a total of twelve tests, I found that the variation in the hit ratios encountered had an average absolute discrepancy of 0.48%, and a maximum absolute discrepancy of 0.96%. As the discrepancies were produced as a consequence of processing one million trace file references, clearly the actual difference between the implementation of the two simulators is negligible. As a consequence, and taking into account the very low variation of the hit ratio, I did not investigate the issue further. This decision also took into account that the average hit ratio for the twelve tested cache architectures was over 80%, of which approximately 50% of the hits came from the prefetch buffer, thus if there had been a significant discrepancy between the prefetch policy implementations in the two simulators it would have been more clearly highlighted.

6.3.2 Animated Mode HASE Dinero Validation

The validation technique used for the Animated mode of HASE Dinero has already been outlined in this chapter. To clarify the technique, a description is provided below.

Exactly the same tests as for the validation of the Fast mode of the simulator were used where possible (the tests requiring sub-block placement were omitted as sub-blocks have not been implemented in Animated mode). Each test was first executed using the Animated mode simulator with five hundred trace file references being processed. Secondly the HASE Dinero results screen was updated to show the final results for the Animated mode simulation. A Fast mode simulation was then executed and the ensuing results trace file used to update the HASE Dinero results screen. When the results screen was updated using the Fast mode results trace file, it was possible to perform a visual inspection to see if any of the results changed from those produced using Animated mode; any changes were highly visible.

As a consequence of the Animated mode validation, three minor errors were found in the HASE Dinero source code, all of which were corrected. After the corrections had been applied, the results produced for the Animated mode simulation were exactly the same as for the Fast mode simulation.

6.4 Simulation Speed Evaluation

The final testing requirement of the HASE Dinero project was to compare the simulation speed of the original Dinero with the Fast mode of HASE Dinero.

During the course of testing, I discovered that the major determining factor of the time required to run any given simulation is the level of cache associativity, not the cache size as might be expected. Results supporting this conclusion are shown in figure 6.2.

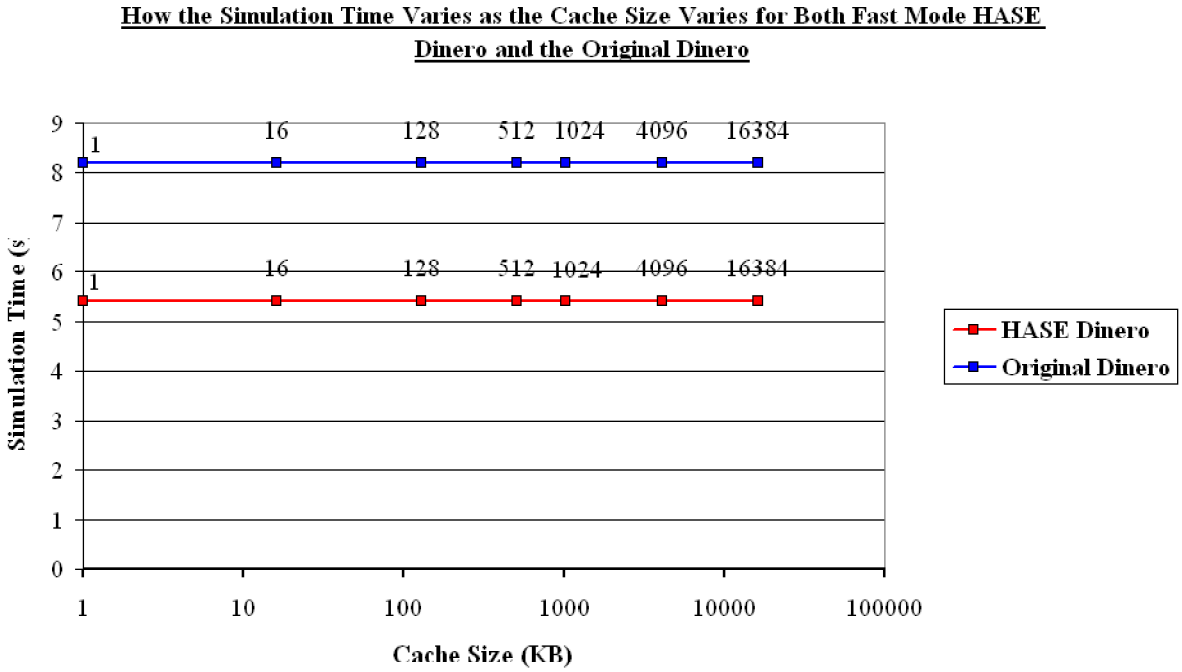


Figure 6.2: A Comparison of Simulation Execution-Times Between the Fast Mode of HASE Dinero and the Original Dinero as the Cache Size Varies

Figure 6.2 appears to indicate that the Fast mode of HASE Dinero offers better performance than the original Dinero, however this deduction is not correct as the original Dinero was executed on a multi-user UNIX Sun server, whereas HASE Dinero was executed on a single-user Windows NT PC, with a 500 megahertz processor and 128 megabytes of main memory, which is likely to offer better performance than the Sun server. In order to minimise the effect of the multi-tasking environments that the two simulators were executed in, I performed each test three times, with the averages being shown in figure 6.2.

Figure 6.3 shows that as the associativity of the cache increases, so does the time required to run the cache simulation. During the course of testing, it became apparent that the original Dinero offers far superior simulation performance for caches with a high degree of associativity. The Fast mode of HASE Dinero required 2654 seconds to perform a simulation of a fully-associative cache with 16384 cache lines, whereas the original Dinero required just 17 seconds for the same cache configuration. When this fact is combined with knowledge that HASE Dinero was executed on a faster platform, the large performance gap between the two simulators should be clear. As a consequence of this performance issue with the Fast mode of HASE Dinero, I performed a series of tests to determine how the simulation run-time scales with the associativity-level of the simulated cache, the results of which are shown in figure 6.3.

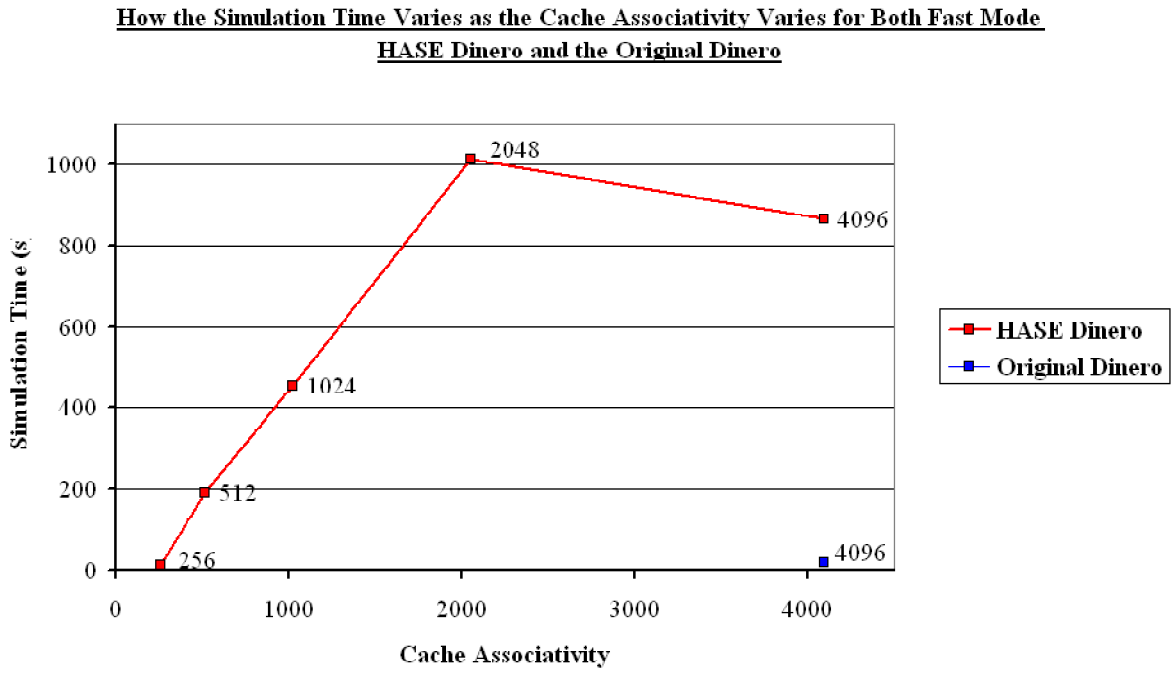


Figure 6.3: A Comparison of Simulation Execution-Times Between the Fast Mode of HASE Dinero and the Original Dinero as the Cache Associativity Varies

Figure 6.3 clearly indicates that the simulation execution time linearly scales with the associativity-level simulated, up to 864 seconds for HASE Dinero, whilst the execution time for the original Dinero does not exceed twenty seconds. Note that the execution time for the original Dinero is only shown for a 4096-way associative cache as this was the longest time required by the original Dinero for any tested cache configuration. An interesting point to note from figure 6.3 is that the HASE Dinero execution time is less for a 4096-way associative cache than it is for a 2048-way associative cache. This performance increase is gained because, for the particular cache configuration used to produce the results in figure 6.3, 4096-way associativity represents a fully-associative cache. As was described in **section 4.2.4.3**, the Fast mode of HASE Dinero uses an optimised lookup mechanism when a fully-associative cache is simulated. The benefit of this mechanism is clearly demonstrated in figure 6.3, where a 4096-way associative cache requires less time to simulate than a 2048-way associative cache.

As was described in **section 4.2.4.3**, I decided to not incorporate the fast cache lookup algorithm into set-associative caches. This decision was made partly because all real, hardware-based, caches of a significant size (one kilobyte and above) use low associativity levels due to the significant cache latency problem caused by implementing high-associativity cache-hit / cache replacement schemes in hardware. For this reason, I do not believe that the relatively slow performance of the Fast Mode of HASE Dinero will cause a problem to users of HASE Dinero, especially when it is remembered that HASE Dinero is primarily intended as a teaching tool, not as a cache evaluation tool for a cache design professional. Additionally, the results shown in figure 6.3 demonstrate that the Fast mode of HASE Dinero has a low execution time requirement whilst the associativity level is 256-way or less. I do not anticipate that any users of the HASE Dinero simulator will require a cache with a higher associativity than 256-way to be simulated.

7. Conclusion

The HASE Dinero project had two main goals: firstly to provide a cache architecture teaching environment based on the HASE simulation environment, and secondly to provide a mechanism for cache design students to perform realistic cache simulations. The project has successfully attained both of these goals.

The Animated mode of HASE Dinero, together with the in-built tutorial mode and HTML user-guide, was found to be a useful teaching aid during the HCI evaluation stage of the project. The Fast mode of HASE Dinero has been shown to be comparable with the original Dinero, in terms of simulator efficiency, if the associativity level of the simulated cache is at most 256-way. It is not foreseen that associativity levels higher that 256-way will be required to be simulated because most hardware cache implementations have relatively low associativity.

In conjunction with the successful attainment of the above two project goals, the validation stage of the project demonstrated that both simulator modes produce results which are identical to the original Dinero, with the exception of one slight discrepancy. An additional factor of the project implementation that can be taken to indicate a high degree of accuracy is that both versions of the HASE Dinero simulator (Fast and Animated), each of which uses completely separate source code, produced identical results for all of the tests performed.

Throughout the course of the project, a number of areas for potential future work with the project have become apparent:

1. The most significant omission from the original Dinero cache simulator is a realistic timing delay model. The Dinero cache simulator only calculates cache hit ratios, with no indication of the actual time required to service all of the CPU memory address references that have been processed. Smith [17] uses a proportional memory hierarchy delay model where one time unit is required to service a cache hit, and ten time units are required to service a demand request from main memory. The incorporation of a realistic timing delay model into HASE Dinero would allow four of the cache optimisation techniques described in **section 4.4.1** to be simulated, as well as providing a useful additional metric of cache performance.

2. HASE Dinero could be incorporated into an existing HASE CPU simulation, such as the DLX simulator [2, 10]. This would allow two important additions to be made to the HASE Dinero simulator: firstly, HASE Dinero does not currently have any ‘real’ data handling capability, with byte sequence numbers being used because the actual data stored in the cache has no meaning for a Dinero simulation. However, if HASE Dinero were to be used as the memory hierarchy for a CPU then it would be essential to allow real data to be stored in both the cache and main memory. Secondly, a current area of much research into improving cache performance is prefetching [7]. If HASE Dinero was added to a CPU simulator then the CPU instruction set could be extended to include special prefetch commands that would dynamically modify the prefetch stride, during CPU operation, so that the cache hit ratio was optimised.

3. I found that during the course of HASE Dinero development I spent approximately one third of my time discovering how the HASE simulation environment operates. This understanding was essential in order to produce a working simulator in HASE; as HASE has evolved over a period of eight years, with no up-to-date manual being currently available, it should be clear that the methodology required to gain an understanding of the advanced simulation features of HASE is a time-consuming process. In addition to the above issue with HASE project development time, two important usability problems with the HASE interface were identified in chapter 5. A solution to these two problems would enhance the usefulness of HASE as a learning environment. One final aspect of HASE that I feel could be altered into a more useful form is the Experiment mode. NT HASE allows multiple Dinero cache simulations to be performed in one large simulation batch, by running the simulations separately in turn and separately storing the results trace files. Unfortunately, there is no method available to read the results of a batch simulation back into HASE en masse. In addition, there is no mechanism available in the Experiment mode to instruct HASE to only perform simulations with parameters that are a power of two, with only linear increments of each simulated parameter being allowed. As a consequence, many of the results trace files produced by a batch HASE Dinero simulation do not contain useful data as the simulated parameters were not a power of two and thus invalid. In addition, it would be useful if HASE were to be updated to include a graphing package that could be used to display the results of batch simulations in both a tabular and a graphical format.

4. I feel that there is one aspect of the Animated simulation mode that could be improved so that HASE Dinero would be easier to understand. The cache control unit could be changed into a *compentity*, with the lower-level entities of the cache control unit providing an animated demonstration of how each CPU address is split into tag, index and block-offset components, and how these separate components are used in determining whether a cache hit occurs. The steepest part of the cache architecture learning curve is undoubtedly that of mastering how the split of the CPU address varies with different cache configurations; the addition of this extra feature to HASE Dinero would address the issue.

Appendix A: The HTML Tutorial Guide

Tutorials Section

Back to Contents

How to Use the HASE Dinero Tutorials

This section assumes that you currently have a copy of NT HASE with the Dinero project loaded and built. If this is not the case, please follow the guidelines in the Guide to Use section of this website.

Each tutorial is designed to be used in conjunction with reading the descriptive text on this page. Furthermore, the tutorials are designed to be run in sequential order as each builds on concepts previously introduced.

There are 14 tutorials available in HASE Dinero, each of which can be run by using the following set of instructions:

1. Before running a tutorial, ensure that the cache is shown at its lowest level of hierarchy. This can be done by first clicking on the Design button at the top of the HASE screen, then right-clicking on the Level 1 Cache entity. Choose Expand from the ensuing pop-up menu.
2. Put HASE into simulation mode. This can be done by clicking on the Simulate button at the top of the screen.
3. Right-click on the Simulation Control window, in the main screen, and select Simulation Parameters from the pop-up menu. Next choose the desired tutorial from the drop-down list of the Simulation Mode parameter, then click on Ok.
4. Go to the Simulate menu at the top of the screen and click on the Run Simulation command.
5. Once the simulation is completed, go back to the Simulate menu and choose the Animate command. This will load the Animation window.
6. Click on the Change button from the Animation window and then double-click on the tracefile.trace file in the results sub-directory.

You are now ready to view the tutorial. The recommended way of doing this is to click on the yellow button of the Animation window showing a '1' over the top left-hand corner of a page. This instructs HASE to playback one simulation time unit. Note that the tutorial can be paused in the middle of animation by clicking on the pause button. Alternatively, clicking on the play button will cause the animation to run until simulation completion.

The main benefit of using HASE Dinero in tutorial mode is the descriptive text, shown at the bottom of the HASE screen, which indicates what is currently happening in the simulation.

All of the tutorials require one or more of the following screens to be open:

Unified Cache Contents
Data Cache Contents
Instruction Cache Contents
Prefetch Buffer
Victim Cache

To open one of the cache contents windows, right click on the Cache Memory entity (in the main window) and select View->*Desired* Cache Contents from the ensuing pop-up menu. Note that the resulting window can be resized and moved to suit.

To open the prefetch buffer or victim cache windows, right click on the Cache Control Unit (in the main window) and select View->Prefetch Buffer (or Victim Cache) from the ensuing pop-up menu. Note that the resulting window can be resized and moved to suit.

For an explanation of the contents of the Cache Contents windows, click here.

For an explanation of the contents of the Prefetch Buffer or Victim Cache windows, click here.

The HASE Dinero Tutorials

It is essential that you gain an understanding of how CPU addresses are used to determine where data can be stored in a cache BEFORE running any of the HASE Dinero tutorials. For an explanation of this process, click here.

The following tutorials are available:

Tutorial 1: A Direct-Mapped Cache
Tutorial 2: A Set-Associative Cache
Tutorial 3: A Fully-Associative Cache
Tutorial 4: The Benefit of Increased Block Size
Details on the Principle of Locality can be found here.
Tutorial 5: The Operation of a Split Instruction / Data Cache
Tutorial 6: A Demonstration of LRU Replacement
Tutorial 7: A Demonstration of FIFO Replacement
Tutorial 8: A Demonstration of Random Replacement
Tutorial 9: The Copy-Back Write Policy
Tutorial 10: The Write-Through, Write Policy
Tutorial 11: The Write-Through, No-Write Policy
Tutorial 12: The Always-Prefetch Fetch Policy
Tutorial 13: The Miss-Prefetch Policy
Tutorial 14: The Benefit of a Victim Cache

Appendix B: Parameter Validity Tests

This appendix lists the parameter validity checks performed by HASE Dinero before a simulation is permitted to execute.

As of May 2000, the following constants are defined for use by the validity checks:

```
MAX_CACHE_SIZE = 16 (MB)
MAX_ANIMATED_CACHE_SIZE = 4 (KB)
MAX_ANIMATED_TRACE_LINES = 500
MAX_FAST_TRACE_LINES = 10000000
DEFAULT_ANIMATED_TRACE_LINES = 100
MAX_BUFFER_SIZE = 100 (Prefetch Buffer and Victim Cache)
```

Tests performed for both Animated mode and Fast mode simulations:

```
//CHECK CACHE SIZES & VALIDITY OF SIZE
if((basic->Unified_Cache_Size == 0) && (basic->Data_Cache_Size == 0) && (basic->Instruction_Cache_Size == 0)){
        if(error_line < 15) strcpy(errors[error_line++], "Invalid_Cache_Sizes-All_Zero");
}
if(basic->Unified_Cache_Size > 0){
        if((basic->Data_Cache_Size != 0) || (basic->Instruction_Cache_Size != 0)){
                if(error_line < 15) strcpy(errors[error_line++], "Invalid_Cache_Sizes-Unified_OR_Data&Ins");
        }
}
else if(basic->Unified_Cache_Size < 0){
        if(error_line < 15) strcpy(errors[error_line++], "Unified_Cache_Size_cannot_be_negative");
}
else if((basic->Data_Cache_Size <= 0) || (basic->Instruction_Cache_Size <= 0)){
        if(error_line < 15) strcpy(errors[error_line++], "Invalid_Cache_Sizes_-_Ins_XOR_Data=1");
}
if(basic->Unified_Cache_Size > (MAX_CACHE_SIZE*1024*1024)){
        if(error_line < 15) strcpy(errors[error_line++], "Unified_Cache_Size_too_big");
}
if(basic->Data_Cache_Size > (MAX_CACHE_SIZE*1024*1024)){
        if(error_line < 15) strcpy(errors[error_line++], "Data_Cache_Size_too_big");
}
if(basic->Instruction_Cache_Size > (MAX_CACHE_SIZE*1024*1024)){
        if(error_line < 15) strcpy(errors[error_line++], "Instruction_Cache_Size_too_big");
}
if(MAX_CACHE_SIZE <= 2048){
        if(!check_power(basic->Unified_Cache_Size)){
                if(error_line < 15) strcpy(errors[error_line++], "UCache_Size_must_be_a_power_of_2");
        }
        if(!check_power(basic->Data_Cache_Size)){
                if(error_line < 15) strcpy(errors[error_line++], "DCache_Size_must_be_a_power_of_2");
        }
        if(!check_power(basic->Instruction_Cache_Size)){
                if(error_line < 15) strcpy(errors[error_line++], "ICache_Size_must_be_a_power_of_2");
        }
}
else{
        if(error_line < 15) strcpy(errors[error_line++], "MAX_CACHE_SIZE(CONTROL.hase)_too_big");
}

//CHECK BLOCK SIZE
if(basic->Block_Size<=0){
        if(error_line < 15) strcpy(errors[error_line++], "Block_Size_must_be_positive");
}
if(!check_power(basic->Block_Size)){
        if(error_line < 15) strcpy(errors[error_line++], "Block_Size_must_be_a_power_of_2");
}
```

```
if(basic->Unified_Cache_Size){
	if(basic->Block_Size > basic->Unified_Cache_Size){
		if(error_line < 15) strcpy(errors[error_line++], "Block_Size_>_cache_size!");
	}
}
else{
	if((basic->Block_Size > basic->Instruction_Cache_Size) || (basic->Block_Size >	basic->Data_Cache_Size)){
		if(error_line < 15) strcpy(errors[error_line++], "Block_Size_>_cache_size!");
	}
}

//CHECK ASSOCIATIVITY
if(basic->Associativity<0){
	if(error_line < 15) strcpy(errors[error_line++], "Associativity_must_be_>=0");
}
if(!check_power(basic->Associativity)){
	if(error_line < 15) strcpy(errors[error_line++], "Associativity_must_be_a_power_of_2");
}

//CHECK WRITE POLICY
if((basic->Write_Policy == COPY_BACK) && (basic->Write_Allocation == NO_WRITE)){
	if(error_line < 15) strcpy(errors[error_line++], "Copy-Back_AND_No-Write_is_invalid");
}

//CHECK PREFETCHING SCHEME
if(advanced->Fetch_Policy != DEMAND_FETCH){
	if((advanced->Fetch_Policy == FORWARD_PREFETCH) || (advanced->Fetch_Policy == SUB_BLOCK_PREFETCH)){
		if(!advanced->Sub_Block_Size){
			if(error_line < 15) strcpy(errors[error_line++], "Fetch_Policy_invalid_without_Sub_Blocks");
		}
	}
	else{
		if(basic->Unified_Cache_Size){
			if(advanced->Unified_Prefetch_Distance < 0){
				if(error_line < 15) strcpy(errors[error_line++], "Unified_Prefetch_Distance_invalid");
			}
			if((advanced->Fetch_Policy == FORWARD_PREFETCH) || (advanced->Fetch_Policy ==
			SUB_BLOCK_PREFETCH)){
				if(advanced->Unified_Prefetch_Distance >= (basic->Block_Size/advanced->Sub_Block_Size)){
					if(error_line < 15) strcpy(errors[error_line++], "Unified_Prefetch_Distance_invalid");
				}
			}
		}
		else{
			if(advanced->Data_Prefetch_Distance < 0){
				if(error_line < 15) strcpy(errors[error_line++], "Data_Prefetch_Distance_invalid");
			}
			else if(advanced->Instruction_Prefetch_Distance < 0){
				if(error_line < 15) strcpy(errors[error_line++], "Instruction_Prefetch_Distance_invalid");
			}
			if(!advanced->Data_Prefetch_Distance && !advanced->Instruction_Prefetch_Distance){
					if(error_line < 15) strcpy(errors[error_line++], "Prefetch_Distances_invalid");
			}
			if((advanced->Fetch_Policy == FORWARD_PREFETCH) || (advanced->Fetch_Policy ==
			SUB_BLOCK_PREFETCH)){
				if(advanced->Data_Prefetch_Distance >= (basic->Block_Size/advanced->Sub_Block_Size)){
					if(error_line < 15) strcpy(errors[error_line++], "Data_Prefetch_Distance_invalid");
				}
				if(advanced->Instruction_Prefetch_Distance >= (basic->Block_Size/advanced->Sub_Block_Size)){
					if(error_line < 15) strcpy(errors[error_line++], "Instruction_Prefetch_Distance_invalid");
				}
			}
		}
	}
}
if(advanced->Prefetch_Buffer_Size < 1){
	if(error_line < 15) strcpy(errors[error_line++], "Prefetch_Buffer_invalid");
}
else if(advanced->Prefetch_Buffer_Size > MAX_BUFFER_SIZE){
	if(error_line < 15) strcpy(errors[error_line++], "Prefetch_Buffer_too large");
}
```

```
//CHECK VICTIM CACHE
if(advanced->Victim_Cache_Size < 0){
	if(error_line < 15) strcpy(errors[error_line++], "Victim_Cache_Size_invalid");
}
if(advanced->Victim_Cache_Size > MAX_BUFFER_SIZE){
	if(error_line < 15) strcpy(errors[error_line++], "Victim_Cache_too_large");
}

//CHECK SUB-BLOCK SIZE
if(advanced->Sub_Block_Size<0){
	if(error_line < 15) strcpy(errors[error_line++], "Sub_Block_Size_must_be_positive");
}
if(advanced->Sub_Block_Size >= basic->Block_Size){
	if(error_line < 15) strcpy(errors[error_line++], "Sub_Block_Size_>=_Block_Size!");
}
if(!check_power(advanced->Sub_Block_Size)){
	if(error_line < 15) strcpy(errors[error_line++], "Sub_Block_Size_must_be_a_power_of_2");
}

//CHECK FLUSH COUNT
if(advanced->Flush_Count < 0){
	if(error_line < 15) strcpy(errors[error_line++], "Flush_Count_must_be>=0");
}

//CHECK CUSTOM TRACE FILE (IF SELECTED)
if(Trace_Input_Type == CUSTOM){
	trace_file = fopen("custom.trace", "r");
	if(!trace_file){
		if(error_line < 15) strcpy(errors[error_line++], "Couldn't_open_custom_trace_file");
	}
	else{
		for(int i=0; i<DEFAULT_ANIMATED_TRACE_LINES; ++i){
			if(!fgets(test, 15, trace_file)){	//read a line from the trace file
				if(!i){	//couldn't read first line of trace file
					if(error_line < 15) strcpy(errors[error_line++], "Error_reading_custom_trace");
					break;
				}
				else{	//trace file smaller than DEFAULT_ANIMATED_TRACE_LINES lines
					Maximum_Count = i;
					break;
				}
			}
			//check the trace data is valid
			if((test[0] < '0') || (test[0] > '4')){
				if(error_line < 15){
					strcpy(errors[error_line], "Invalid_data_in_trace:");
					strcat(errors[error_line++], itoa(i, line_no, 10));
				}
				break;
			}
			for(int j=2; j<11; ++j){
				if((test[j] < '0') || (test[j] > '9')){
					if((test[j] < 'a') || (test[j] > 'f')){
						if((test[j] < 'A') || (test[j] > 'F')){
							if(test[j] != '\n'){
								if(error_line < 15){
									strcpy(errors[error_line], "Invalid_data_in_trace:");
									strcat(errors[error_line++], itoa(i, line_no, 10));
								}
								//break out of outer for loop
								i = DEFAULT_ANIMATED_TRACE_LINES;
								break;
							}
							else break;
						}
					}
				}
			}
		}
	}
	if(trace_file) fclose(trace_file);
}
```

```
//CHECK SIMULATION CONTROL PARAMETERS
if(Maximum_Count > MAX_FAST_TRACE_LINES){
        if(error_line < 15) strcpy(errors[error_line++], "Maximum_Count_too_large");
}
if(Maximum_Count < 1){
        if(error_line < 15) strcpy(errors[error_line++], "Maximum_Count_must_be>0");
}
if(Skip_Count < 0){
        if(error_line < 15) strcpy(errors[error_line++], "Skip_Count_must_be>=0");
}
if(Skip_Count > Maximum_Count){
        if(error_line < 15) strcpy(errors[error_line++], "Skip_Count_must_be<Maximum Count");
}

//CHECK VALIDITY OF SELECTED CACHE COMBINATION (ONLY IF ALL ABOVE CHECKS ARE OK)
if(!error_line){
        if(basic->Unified_Cache_Size){
                if(basic->Associativity > (basic->Unified_Cache_Size / basic->Block_Size)){
                        if(error_line < 15) strcpy(errors[error_line++], "Associativity_>(UCache_Size/Block_Size)");
                }
                if(basic->Associativity == (basic->Unified_Cache_Size / basic->Block_Size)){
                        basic->Associativity = 0;
                }
        }
        else{
                if(basic->Associativity > (basic->Data_Cache_Size / basic->Block_Size)){
                        if(error_line < 15) strcpy(errors[error_line++], "Associativity_>(DCache_Size/Block_Size)");
                }
                if(basic->Associativity > (basic->Instruction_Cache_Size / basic->Block_Size)){
                        if(error_line < 15) strcpy(errors[error_line++], "Associativity_>(ICache_Size/Block_Size)");
                }
                if((basic->Instruction_Cache_Size == basic->Data_Cache_Size) && (basic->Associativity == (basic->Data_Cache_Size
                / basic->Block_Size))){
                        basic->Associativity = 0;
                }
        }
}
```

Tests performed for Animated mode only:

```
if((Maximum_Count - Skip_Count) > MAX_ANIMATED_TRACE_LINES){
        if(!Skip_Count) Maximum_Count = DEFAULT_ANIMATED_TRACE_LINES;
        else if(error_line < 15) strcpy(errors[error_line++], "Animated_Mode_trace_lines_too_high");
}
if(basic->Unified_Cache_Size){
        if(basic->Unified_Cache_Size > (MAX_ANIMATED_CACHE_SIZE * 1024)){
                if(error_line < 15) strcpy(errors[error_line++], "Animated_Mode_Max_Cache_Size_Exceeded");
        }
}
else if((basic->Data_Cache_Size + basic->Unified_Cache_Size) > (MAX_ANIMATED_CACHE_SIZE * 1024)){
        if(error_line < 15) strcpy(errors[error_line++], "Animated_Mode_Max_Cache_Size_Exceeded");
}
if(basic->Block_Size > 16){
        if(error_line < 15) strcpy(errors[error_line++], "Block_Size_must_be_<16_in_Animated_Mode");
}
if(advanced->Sub_Block_Size){
        if(error_line < 15) strcpy(errors[error_line++], "No_Sub-Blocks_in_Animated_Mode");
}
```

Appendix C: The HCI Experiment Sheet

HASE Dinero Experiment Sheet

In this experiment, you will use HASE Dinero to simulate a cache configuration slightly different to the configuration you have just been shown.

The cache configuration that you should simulate is given below:

Ensure that HASE Dinero is set to the Animated simulation level. Modify the cache parameters so that the following configuration is set:

- Trace File Limit: 20
- Trace File Type: SPICE
- Unified Cache Size: 128 bytes
- Block Size: 8 bytes
- Associativity: Fully-Associative
- Replacement Policy: LRU
- Write Policy: COPY-BACK
- Write Allocation Policy: WRITE

You will not be provided with any help in setting the above parameters, if you require assistance, the printed page handed out at the start of the experiment entitled “Basic Cache Control” should provide the assistance you require. If you require further assistance, the HASE Dinero HTML user-guide is available.

Once you have successfully set-up HASE Dinero with the above configuration, run the simulation and animate the resulting trace file. You will be provided with assistance at this stage if you are unsure how to proceed.

Note that you will be asked to describe the current simulation activity as the simulation is animated. **You should single-step through the simulation so that there is sufficient time to describe what is happening on-screen.**

Appendix D: The HCI Questionnaire

HASE Dinero Post-Experiment Questionnaire

Please indicate your agreement or disagreement with the following statements (1 indicates complete disagreement and 5 complete agreement):

1. Animation of the display window during simulation playback helps me to understand what is happening.
   Disagree 1 2 3 4 5 Agree

2. I found the Status screen useful for understanding the current cache activity.
   Disagree 1 2 3 4 5 Agree

3. It is easy to understand the results displayed in the Results screen.
   Disagree 1 2 3 4 5 Agree

4. The colour schemes of the Status, Results, and Cache Configuration screens helps to distinguish the different screen functions.
   Disagree 1 2 3 4 5 Agree

5. The brightly coloured state changes of the Cache Control Unit help me to understand the current simulation activity.
   Disagree 1 2 3 4 5 Agree

6. The HASE Dinero screen does not provide too much information; it is easy to understand the information displayed.
   Disagree 1 2 3 4 5 Agree

7. The colour scheme chosen for HASE Dinero helps make the simulation screen easier to understand.
   Disagree 1 2 3 4 5 Agree

8. I think that HASE Dinero provides a useful tool to assist in understanding how computer caches operate.
   Disagree 1 2 3 4 5 Agree

If you have any comments regarding any specific problems with HASE Dinero or aspects of the interface that you think could have been implemented in a better way, please describe in the space provided below:

The HCI Questionnaire Results

The table below shows the distribution of questionnaire answers for the five test subjects who participated in the HASE Dinero HCI evaluation.

Rating Chosen:	**1**	**2**	**3**	**4**	**5**
Question					
1:	0	0	2	3	0
2:	0	0	0	5	0
3:	0	0	0	0	5
4:	0	1	2	2	0
5:	0	0	2	3	0
6:	0	0	3	2	0
7:	0	1	2	2	0
8:	0	0	1	4	0

Bibliography

[1] AGARWAL, A., HOROWITZ, M. and HENNESSY, J. [1989]. "An Analytical Cache Model," ACM Transactions on Computer Systems, Volume 7.2.

[2] COE, P.S. [1994]. "An On-Line Teaching System for Computer Architecture," University of Edinburgh, 4th Year Project Report.

[3] COE, P.S., HOWELL, F.W., IBBETT, R.N. and WILLIAMS, L.M. [1998]. "A Hierarchical Computer Architecture Design and Simulation Environment," ACM Transactions on Modelling and Computer Simulation, Volume 8.4.

[4] COE, P.S., HOWELL, F.W., WILLIAMS, L.M. and IBBETT, R.N. [1996]. "Evolution of the HASE System," University of Edinburgh, Department of Informatics, School of Computer Science.

[5] COE, P.S., WILLIAMS, L.M. and IBBETT, R.N. [1997]. "An Integrated Learning Support Environment for Computer Architecture," 3rd Annual Workshop on Computer Architecture Education at HPCA-3, Texas.

[6] DIX, A., FINLAY, J., ABOWD, G. and BEALE, R. [1998]. Human-Computer Interaction. Prentice Hall Europe.

[7] GONZÁLEZ, J. and GONZÁLEZ, A. [1997] "Speculative Execution Via Address Prediction and Data Prefetching," International Conference on Supercomputing.

[8] HAMLET, D. [1994]. "Software Quality, Software Process and Software Testing," Portland State University, Department of Computer Science, Centre for Software Quality Research.

[9] HANDY, J. [1993]. The Cache Memory Book, Academic Press, Boston.

[10] HENNESSY, J.L. and PATTERSON, D.A. [1996]. Computer Architecture: A Quantitative Approach, Morgan Kaufmann Publishers, Second Edition.

[11] HOWELL, F.W. and IBBETT, R.N. [1996]. Modelling and Simulation of Advanced Computer Systems: Techniques, Tools and Tutorials, Gordon and Breach.

[12] IBBETT, R.N., HEYWOOD, P.E. and HOWELL, F.W. [1996]. "HASE: A Flexible Toolset for Computer Architects," The Computer Journal, Volume 38.10.

[13] MOUDGILL, M. [1998]. "Techniques For Fast Simulation of Associative Cache Directories," Computer Architecture News, Volume 26.2.

[14] ROSENBLUM, M., BUGNION, E., DEVINE, S. and HERROD, S.A. [1997]. "Using the SimOS Machine Simulator to Study Complex Computer Systems," ACM Transactions on Modelling and Computer Simulation, Volume 7.1.

[15] SCHIRRMEISTER, F. and KROLIKOSKI, S. [1999]. "The System-Level Hardware / Software Co-Design Challenge," Cadence Design Systems Inc.

[16] SIEGEL, H.J., ABRAHAM, S. et. al. [1992]. "Report of the Purdue Workshop on Grand Challenges in Computer Architecture for the Support of High Performance Computing," Journal of Parallel and Distributed Computing, 16.

[17] SMITH, A.J. [1982]. "Cache Memories," ACM Computing Surveys, Volume 14.3.

[18] SO, K. and RECHTSCHAFFEN, R. [1986]. "Cache Operations by MRU Change," Research Report RC 11613 REV, IBM Thomas J. Watson Research Centre, Yorktown Heights, NY.

[19] UHLIG, R.A. and MUDGE, T.N. [1997]. "Trace-Driven Memory Simulation: A Survey," ACM Computing Surveys, Volume 29.2.

[20] UHLIG, R.A., NAGLE, D., MUDGE, T.N. and SECHREST, S. [1994]. "Trap-Driven Simulation with Tapeworm II," Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA.

Contents

Acknowledgements

I am most grateful to Professor Roland Ibbett for providing invaluable advice and encouragement throughout the course of the project.

I would also like to thank Doctor Paul Coe for sharing his prodigious technical knowledge on the operation of HASE.

Finally, I would like to thank all those who found time to participate in the evaluation of the user interface.

University of Edinburgh
Division of Informatics

An Animated Teaching and Evaluation Tool for Computer Caches

4th Year Project Report
Computer Science

Mark Seymour

May 31, 2000

Abstract: The primary project goal was to develop a computer cache teaching tool in the Hierarchical computer Architecture design and Simulation Environment (HASE). The cache teaching tool would demonstrate the operation of a wide range of alternative cache architectures, based on an existing cache simulator called Dinero. HASE provides an ideal platform from which to develop a cache teaching tool, primarily due to the ability of HASE to provide a visualisation of cache activity via an animation of the simulation design window.

The secondary project goal was to enhance the cache teaching tool by implementing a fast memory-efficient simulator, which would allow students to realistically evaluate the performance of arbitrary cache configurations.

Both of the project goals were successfully achieved, with the main tasks undertaken being: the implementation of a cache simulator in HASE, the design of a teaching environment for the cache simulator, and the validation of the cache simulator using standard software testing techniques.