Manchester Metropolitan University

BSc. (Hons) Computer Science

An Investigation into Value Profiling and its Applications

Author:
Graham Markall

Supervisor:
Dr. Andy Nisbet

Abstract

Value Profiling is the recording of live inputs and outputs of computations during the execution of a program. Value Profiling can be used to determine the invariance of the inputs and outputs of these computations. Computations with high degrees of invariance in their inputs may be the subject of optimisations to exploit this behaviour. Previous research into Value Profiling has shown that there is a significant amount of invariance in the inputs of computations throughout the execution of most programs that cannot be detected at compile-time. Other research efforts have used Value Profile data to guide the design of schemes to increase performance or decrease power consumption. This project seeks to validate the hypothesis that there is a high level of invariance in the inputs of computations throughout the execution of most programs. Value Profiling tools were developed on two different platforms and used to record Value Profile data. The recorded Value Profile data is examined, and it is shown that there are high levels of invariance in the inputs of computations throughout the execution of most programs. Additionally the Value Profile data was used to guide the design of a bus encoding to reduce power consumption by the memory bus, and to develop a cache (termed a *Value Reuse Cache*) which stores and recalls the live inputs and outputs of frequently occurring computations to improve performance. Both of these schemes are promising - the Value Reuse Cache is shown to have an average hit rate of over 60% for all cacheable instructions across all benchmarks, and the bus encoding reduces switching activity by over 40% in certain cases.

Acknowledgements

People who have contributed to this project and I would like to thank are:

- Dr. Andy Nisbet, for his supervision of this project, and providing me with guidance and suggestions throughout.
- The technicians in the Department of Computing and Mathematics, who assisted by setting up a machine with plenty of RAM and extra disk space to use for Value Profiling runs.
- Sarah Burberry, for finding me quiet places to work on this report.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 What is Value Profiling?

Computations in an executing program have a set of live inputs and live outputs. Live inputs are the inputs used by the computation. Live outputs are the variables which the computation can potentially write to and modify.

Value profiling is the recording of live inputs and outputs of computations during the execution of the program. At the end of the execution, all the data recorded constitutes the *value profile* for the execution of the program. The granularity of a computation may vary. At the smallest scale, a single instruction can be regarded as a computation. In this case, the inputs of the computation may be the instruction's operands, the current state of some of the registers/flags, or memory locations. The outputs of the computation may be registers, flags or memory locations.

Larger naturally identifiable granularities include:

- Basic blocks - A basic block is a set of instructions which always runs sequentially from beginning to end (Parsons, 1992). There are no branch instructions in the middle of a basic block, and no branch instruction branches into the middle of a basic block. The live inputs of a basic block may include the current state of the registers/flags, and several memory locations.

- Function/procedure calls - A function or procedure call is a jump that passes control to a subroutine (Parsons, 1992). After the subroutine has finished execution, control is returned to the caller. The live inputs of a function call are usually its parameters. The live outputs of a function call are its returned value, and the results of any side-effects it may have.

- Arbitrary sections of code - These are often referred to as *traces*. A trace is a sequence of several instructions which are executed sequentially. A trace may span several basic blocks. A trace ends either when an unconditional branch is made, after a certain number of conditional branches, or after a certain number of instruction executions (Luk *et al.*, 2005).

1.2 Why use Value Profiling?

Value Profiling can be used to determine the invariance of the inputs and outputs of over the set of all computations[1] throughout the execution of a program. Invariance in the inputs of a computation is termed *Value Reuse*. The computations with the most Value Reuse can be the target of optimisations to increase the overall performance of the program. Additionally, the profile information can be used to guide these optimisations. Computations with high degrees of Value Reuse may be modified to exploit this behaviour.

Examples of the application of Value Profiling and Value Reuse Optimisations include:

- (Feller, 1998) used Value Profiling to determine the invariance of instructions and their operands, and the values stored in memory locations referenced by load instructions. The Value Profile

[1] Recall that a computation could be an individual instruction, a basic block, a function call, or an arbitrary code section.

information was used to guide source-code optimisations of two of the programs profiled, *m88ksim* and *hydro2d*. The modifications led to a speedup of 13% and 12% respectively.

- (Yang & Gupta, 2002) used Value Profile information to show that throughout the execution of a set of 15 benchmarks, up to 48% of memory locations were occupied by 8 benchmark specific distinct values. This information was used to guide the design of an encoding for a low power data bus (Yang *et al.*, 2004).
- (Kumar, 2003) used Value Profiling to determine the invariance of calls to functions in a math library. The Value Profile information was used to guide the implementation of a *Function Evaluation History Table* (FEHT) which stored the results of the most recent calls to the function. The implementation of the FEHT decreased the execution time of the benchmarks tested by up to 6%.
- (Huang & Lilja, 2003) developed a compiler-assisted scheme for reusing the outputs of computations with a granularity between that of instructions and basic blocks, termed *subblocks*. It was shown that a speedup of 36% is possible by reusing the outputs of subblocks chosen by the compiler.

1.3 The Low-Level Virtual Machine Compiler Infrastructure

Low-Level Virtual Machine (LLVM) is a compilation framework created with the goal of allowing transformation and analysis of arbitrary programs at all stages of compilation and execution (Lattner & Adve, 2004a). The source code to all of the elements of LLVM is available.

Modified versions of GCC (*llvm-gcc*) and G++ (*llvm-g++*) which output LLVM *bitcode* (code compiled to LLVM and stored in its native format) have been created. Any C or C++ program to which the full source code is available can be compiled to LLVM bitcode. Generally no modifications need to be made to the source code - however, some Makefiles may require editing.

LLVM uses a common low-level code representation, called an *Intermediate Representation* (IR). Code compiled by llvm-gcc and llvm-g++ is converted to the LLVM IR. Optionally, an optimisation phase can be run on the generated IR. The LLVM compiler (llc) converts the IR to machine code of the required target machine. Several backends are available for the LLVM compiler, including x86, MIPS, PowerPC etc. The same IR is used as the source to generate the machine code for all of the backends.

An interpreter for the LLVM IR bitcode is provided. The organisation of the interpreter source code is quite convenient for instrumentation with additional code to record value profile information. At the IR stage, the target code is in a machine-independent representation. Any profile data gathered using the bitcode interpreter should therefore have relevance for all architectures.

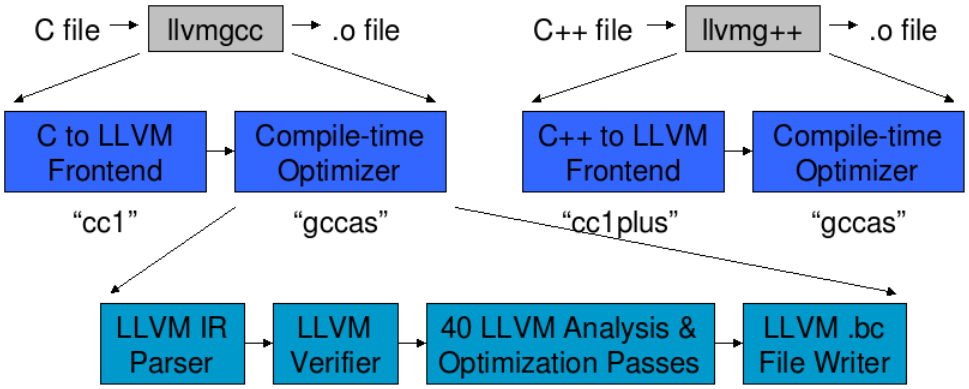

Figure 1.1: The stages of the LLVM compilation process from C and C++.

The above figure, from (Lattner & Adve, 2004b), gives an outline of the stages involved in compiling and optimising a program using LLVM. The C- and C++-to-LLVM frontends output code in LLVM IR. Normally, the LLVM IR will be optimised in the "40 LLVM Analysis & Optimisation Passes" stage, although this is optional. The final output from the "LLVM .bc File Writer" stage can be loaded and executed by the interpreter. The LLVM Compiler, *llc* can be used to compile this .bc file to native code for any supported target, including x86, ARM, etc.

1.4 Pin

Pin is a tool for the instrumentation of native code (Luk *et al.*, 2005). Architectures currently supported by Pin include the Intel Itanium & IA-32, and the ARM family of processors. An API is provided which allows the development of instrumentation tools. This API abstracts away the details of the target architecture, allowing the developer to focus on the development of tools without having to be aware of the intricacies of the underlying system.

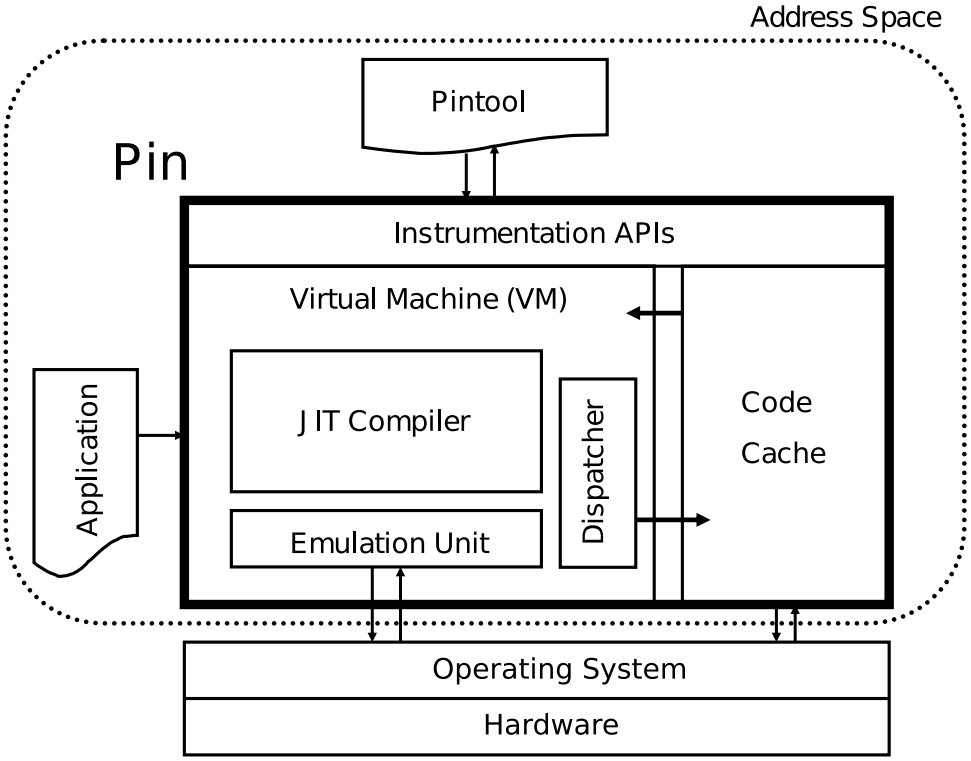

Figure 1.2: The architecture of Pin.

Figure 1.2 from (Luk *et al.*, 2005), shows the architecture of Pin. The programs with which executables are instrumented are called *PinTools*. When Pin is run, the target application and the PinTool are both loaded by Pin. The JIT Compiler recompiles portions of the program to include the instrumentation code and places them in the Code Cache prior to their execution. Additionally, the instrumented portions of code are modified so that when their execution completes, control is returned to the Virtual Machine, which can determine the next portion of code to be executed. If this portion of code has already been recompiled, then it is retrieved from the Code Cache and execution continues. Otherwise, the JIT Compiler is called to recompile and instrument the required portion of code.

In normal operation of an executable (without Pin), the components shown in Figure 1.2 would be reduced to three components: the Application, the Operating System, and the Hardware. The Application would run directly on top of the Operating System layer, which in turn runs on the Hardware layer.

The Pin API is ideal for the development of Value Profiling tools for the x86 architecture. Arbitrary code may be inserted into specific points in the program (e.g. when a memory accesses takes place, or upon the execution of specific instructions) which can record information about the state of the program. Information about the state of the program for Value Profiling would include the values of instruction operands, or the value being transferred across the memory bus.

1.5 MiBench Benchmarks/MiDataSets

The MiBench (Guthaus *et al.*, 2001) set of benchmarks were created with the goal of characterising the workload of embedded processors and microcontrollers. Five categories of benchmarks make up the suite: Industrial control, Network, Security, Consumer Devices, Office Automation and Telecommunications.

Each of the benchmarks are available as C source code. As a result, it is possible to use llvm-gcc to compile each of the benchmarks to the LLVM IR.

Only two input sets are provided with each of the benchmarks, *Small* and *Large*. The Small input set is designed to represent a simple application of the benchmark. The Large input set is designed to represent a more complex, real-world application of the benchmark.

It is expected that performance variations will be exhibited across different input sets for the same benchmark. Therefore, only two input sets are not sufficient to represent a realistic subset of all inputs. MiDatasets (Fursin *et al.*, 2007) provides an additional 18 input sets for each benchmark. Using 20 datasets for each benchmark allows a more representative sample of the performance variations to be generated. Furthermore, the effects of any optimisations can be tested more thoroughly.

1.6 Conclusion to Introduction

This project involves the investigation of Value Profiling on LLVM and the x86 architecture (using Pin). Value Profile data is gathered for the MiBench benchmarks in conjunction with the MiDatasets sets of data. This Value Profile data is analysed and used to guide the design of optimisations which exploit Value Reuse. In order to determine a starting point for investigations into Value Profiling, a literature review was undertaken to assess work already complete in this area, and identify areas requiring further investigation.

The structure of this report is as follows: In Chapter 2 a literature review is presented. Chapter 3 outlines areas of investigation and describes how they will be investigated. Chapter 4 describes the implementation of the Value Profiling tools. In Chapter 5 the results of using these tools are presented and analysed. In Chapter 6 a cache which exploits Value Reuse is presented. In Chapter 7, Value Profiling in conjunction with a cache simulator, and an encoding for a low-power data bus is presented. Chapter 8 gives conclusions to the areas of investigation specified in Chapter 3. Chapters 9 and 10 contain an evaluation of the project and further work to be undertaken.

Chapter 2

Literature Review

2.1 Introduction

The goals of this literature review are to:

- Establish the current state of investigation into Value Profiling.
- Critically assess the work done by others in this area.
- Determine a starting point for investigations into Value Profiling.
- Review the current applications of Value Profiling.

Relevant literature has been gathered and examined. Methods which have been used by others to investigate Value Profiling will be adapted for use with the LLVM infrastructure and Pin in subsequent sections of this report.

2.2 Value Profiling

(Calder *et al.*, 1997)

This motivation for this paper is that variables which exhibit invariant or semi-invariant behaviour at run time cannot be easily identified by a compiler. As the compiler cannot identify these variables, or the values they may frequently store, it is not possible to make optimisations to efficiently exploit this behaviour. A method (*Value Profiling*) is proposed and analysed to determine the variables that exhibit semi-invariant behaviour, in order to enable the compiler to make more effective optimisations. Several potential applications of Value Profiling are suggested, including guiding compiler optimisations and providing hints for value prediction hardware.

A software implementation of a Value Profiler is presented. The Value Profiler does not store all the values encountered during the execution of the program - a limited number is stored in a table, and an algorithm is designed to store only the most frequently occurring values. The algorithm maintains a count of how frequently each instruction has been encountered in the last time period. Periodically the table is sorted in descending order of the most frequently executed instructions, and the bottom half of the table (the least frequently encountered half) is cleared to make room for new entries.

An additional Value Profiling scheme is also presented, called *Convergent Value Profiling*. This scheme is designed to reduce the total time taken to produce an accurate Value Profile for a program. At the beginning of the execution, the Value Profiler is set to profile all instruction executions. Throughout the execution of the program, the Value Profiler checks to see if the Value Profile information for each instruction opcode is converging to a steady state. If it is considered that the Value Profile data for a particular opcode has converged, then Value Profiling for that instruction opcode is turned off. As Value Profiling is turned off for each instruction opcode, the speed of the execution of the program will increase as the Value Profiler consumes less time recording Value Profile data. In order to ensure that the Value Profile data is representative of the entire execution of the program, profiling is periodically turned back

on for each instruction opcode. This allows the profiler to determine if the Value Profile for each opcode has moved from its converged state to an unconverged state.

The conclusion to this paper states that the analysis possible with Value Profiling can be used to determine regions of a program which are optimisable. However, schemes to exploit this potential are not presented in this paper.

2.3 Dynamic Instruction Reuse

(Sodani & Sohi, 1997)

This paper presents a method for reusing the outputs of instructions which have previously been executed using similar inputs. The paper claims to present the very first implementation of a scheme to reuse the results of previous instruction executions. The motivation for developing this method was originally to reduce the branch misprediction penalty in superscalar architectures. However, it is suggested that other scenarios where instructions with the same inputs and outputs are computed also arise frequently.

A microarchitectural mechanism to reuse the result from a previous computation is described, and is termed a *Reuse Buffer*. The Reuse Buffer stores particular attributes of an instruction, and its output. When an instruction is dispatched, the existence of a similar instruction in the Reuse Buffer is checked. If the instruction already exists in the Reuse Buffer, the stored output is retrieved to avoid re-executing the same instruction.

Three different methods of storing the attributes of an instruction in the buffer are discussed. All of these schemes use the program counter as an index into the Reuse Buffer. Each scheme uses a different method to compare the current instruction with instructions in the Reuse Buffer:

Scheme 1 stores the operand values of instructions in the Reuse Buffer. The operand values of the current instruction are compared to the operand values of instructions in the buffer which have a program counter entry similar to the address of the current instruction.

Scheme 2 stores the register names of instructions in the Reuse Buffer. Additionally, this scheme has to keep track of whether results in the reuse buffer are still valid, as data written to registers which an instruction refers to will invalidate the result stored in the reuse buffer. This scheme was devised to make comparison of instructions simple.

Scheme 3 is a more complicated scheme which establishes chains of dependent instructions, and attempts to track the reuse status of these chains of instructions.

Twelve benchmarks were used to test the effects of each implementation of the reuse buffer. Each scheme was tested with a 32-entry, 128-entry and 1024-entry Reuse Buffer. The Reuse Buffer was implemented as a fully associative buffer. The percentage of reused instructions from each group of instructions was recorded. This instruction profile was used to estimate the speed increase compared to a similar architecture which does not implement a Reuse Buffer.

Scheme 1 reused the most instructions. Scheme 2 performed quite poorly. Scheme 3 performed well, but was still not as effective as scheme 1. Scheme 1 therefore provided the greatest speed increase. A summary graph of the proportions of reused executions from each group of instructions is also given. Integer operations were the most reused[2], followed by address calculation instructions, and finally control instructions. A 4-way set associative buffer was also implemented, using scheme 1 for instruction comparison. Almost identical performance can be seen between the fully associative and 4-way set associative Reuse Buffers.

The conclusion to the paper states (as this is the original implementation of a value reuse buffer) that there is further work to be done to refine the implementation of reuse buffers, as these preliminary results appear promising. A highlighted area in which further work is to be done is the implementation of a scheme to decide which instructions should be placed into the reuse buffer, as it is observed that around 80% of entries placed in the Reuse Buffer are evicted without ever being reused.

[2]These two statements (see overleaf) appear to contradict each other. However, it is possible that there are many integer operations present in floating-point benchmarks, which are responsible for a some of the Value Reuse/Value Locality present

2.4 An Analysis of the Potential for Global Level Value Reuse in the SPEC95 and SPEC2000 Benchmarks

(Yi & Lilja, 2001)

The contribution of this technical report is to demonstrate the difference between *Local-level Value Locality*[3] (reoccurrence of the same instruction opcode, operands and program counter) and *Global-level Value Locality* (reoccurrence of instruction opcode and operands, but disregarding the program counter). It is suggested that the exploitation of Global-level Value Locality had previously not been investigated as current Value Reuse techniques use the program counter as an index into the Value Reuse Cache.

A method to test whether there is a difference in the amount of Value Locality at the local and global levels is described. Most results are presented as several tables of numbers - though interpretation of the data is not impossible, it is difficult to verify the statements made regarding the results by inspection. The opcodes of instructions which occur frequently in each benchmark are stated. It is stated that there is no significant difference in the level of Value Locality between integer and floating-point benchmarks[2]. It can be seen that every benchmark frequently executes arithmetic instructions, though no conclusions are drawn regarding the distribution of instruction opcodes.

The authors conclude that the results of the experimentation show that there is greater Value Locality at the global level than there is at the local level. It is not explicitly stated whether further investigation into the exploitation of Global-level Value Locality is worthwhile. It is suggested that a large reduction in execution latency for certain instructions (those requiring many cycles to execute) could be made with the implementation of a Value Reuse scheme. However, because there is greater Value Locality at the global level than at the local level, it can be concluded that further investigation into the exploitation of Global-level Value Locality is warranted.

2.5 Frequent Value Locality and its Applications

(Yang & Gupta, 2002)

It is shown that for 15 SPEC95 benchmarks, a small number of distinct values are stored repeatedly in main memory. Up to 48% of memory locations were occupied by eight benchmark specific distinct values. The contribution of this paper is to demonstrate applications exploiting the Value Locality of the memory. Applications presented are a low power data bus, and a low power Frequent Value Cache. Both of these applications are relevant in the context of embedded architectures.

The investigation into Value Locality is introduced in the context of existing research by (Lipasti *et al.*, 1996) and (Gabbay & Mendelson, 1997). Three methods of analysing the Value Locality of the benchmarks are presented:

- Frequent Value Occurrence in Memory: A method for determining the top 8 most frequently occurring values in memory is described. The results show that on average, 48% of all memory locations are occupied by eight benchmark specific distinct values. It is shown that of these values, the most frequently occurring value is usually 0.
- Frequent Value Distribution in Time: A method to measure the occurrence of frequent values in memory at regular intervals throughout the execution of the benchmark is presented. Results for this benchmark are presented as plots showing the occurrence of frequent values against time. The authors conclude that it can be seen that the most frequently occurring values occur throughout the entire execution of each of the benchmarks. This conclusion is easily verified by inspection of the plots.
- Frequent Value Distribution in Memory: A method to determine the uniformity (or otherwise) of the distribution of frequent values in memory is described. Results are presented in the form of plots of the frequency of occurrence against memory address. The authors state that frequent values occur with a high degree of uniformity throughout the memory. Therefore, no matter what part of memory is currently in use, frequent values are likely to be observed. This conclusion may also be verified by inspection of the plots.

[3]Some literature refers to *Value Locality*. This is synonymous with Value Reuse in this report.

As it has been shown that there is a high degree of Value Locality in memory, the authors hypothesise that frequent values will occur at other areas in the memory hierarchy, including the data bus and the processor cache. The authors propose to test the hypothesis with 15 SPEC95 benchmarks.

Three methods were developed to test this hypothesis:

Method 1: The most frequent values in a given program are found once throughout its execution. The program is instrumented to intercept the data values in all load and store operations. When a load or store operation is executed, the value is recorded. If there are multiple occurrences of a value throughout the lifetime of the program, a count of the occurrences is recorded along with the value, rather than storing multiple copies of the same value. At the end of the execution, the list of values is sorted to find the most frequently occurring values.

Results are presented for this method show the percentage of all memory accesses comprised by the top value up to the top 128 values. It can be seen from the graph that in some cases, only one value represents over 70% of all load/store operations. On average, 50% of all accesses are represented by the top 128 values.

Subsequently the benchmarks were re-executed with a different input. The sets of most frequent values found with the first input set were shown to have a large intersection with the most frequent values found using the second input set.

Method 2: As Method 1 is not suitable for implementation in hardware, this method was designed as a hardware scheme to find frequent values. A technique similar to the Value Profiling technique presented in (Calder *et al.*, 1997) is used, with a modification. Instead of the lower half of the frequent value table being cleared periodically, a *swapping* mechanism is implemented to sort the table.

This method is shown to be very effective in finding frequent values. A comparison is made for each of the 15 benchmarks between the percentage of frequent value accesses found by the swapping method, and the percentage found by the original value profiling method. Again, a high percentage of frequent value accesses are found. Graphs are presented which show the percentage of frequent value accesses, against the number of instructions profiled. As more instructions are profiled, the percentage of frequent value accesses increases up to a peak percentage. Most benchmarks display between 30% and 50% of memory accesses involving frequent values. Some benchmarks have a much higher percentage, including *m88ksim* at 92%, *su2cor* at 75% and *fpppp* at 78%.

Method 3: This method finds a constantly changing set of frequent values throughout the execution of a program. A 32 entry table of frequent values was maintained with a *Least Recently Used* (LRU) eviction policy. On average, 32% of memory accesses involved frequent values found using this method. However, up to 68% of all memory accesses for the *compress* benchmark involved frequent values. The previous two methods had not been as successful in finding frequent values for the *compress* benchmark. This leads the authors to conclude that a changing set of values may provide better results for some benchmarks, though a fixed set of frequent values is successful with others.

As it has been shown that a small number of constantly changing values are frequently involved in memory accesses for all benchmarks, two applications exploiting this characteristic are presented.

Frequent Value Cache: A design for a low-power data cache is presented. The cache exploits the Value Locality of memory accesses, by storing the frequent values in an encoded form. The encoded form only requires $log_2 n$ bits to represent a frequent value. The set of frequent values do not change throughout the program execution. Some detail of the design is presented, which is omitted from this review.

A simulation was used to estimate the power reduction of the Frequent Value Cache. For a 64KB cache, the power reduction was found to be 33%.

Frequent Value Encoding: An encoding was designed for a low-power data bus. Similar to the Frequent Value Cache, the Frequent Value Encoding exploits the Value Locality of memory accesses. Unlike the Frequent Value Cache, the encoding scheme maintains a changing set of 32 frequent

values. A description of the function of the encoding is not be presented here. The results presented show that there is an average of 30% reduction in switching activity on the data bus as a result of the Frequent Value Encoding, thus power consumption is reduced as it is proportional to the switching activity.

In conclusion, this paper has presented a study to determine the prevalence of a small number of frequent values in memory. It has been thoroughly demonstrated that a high percentage of memory locations are occupied by a small number of distinct values. From this, the hypothesis that frequent values will be found elsewhere in the memory hierarchy was formed. This was thoroughly shown to be the case. The two applications developed as a result of these findings are effective in reducing the power consumption of a processor cache and data bus.

An encoding for a low power data bus and a low power Frequent Value Cache are both of relevance to embedded architecture, as many embedded devices operate on battery power, such as mobile phones and PDAs. An investigation into the Value Locality of memory locations is warranted by the results found in this paper. If similar results can be produced, it can be shown that typical applications executed on embedded processors (of which the MiBench suite is representative) may be suitable for hardware schemes to reduce power consumption.

2.6 Increasing Instruction-Level Parallelism with Instruction Precomputation

(Yi *et al.*, 2002)

In this paper, a *Precomputation Table* is presented. A Precomputation Table is a small cache on the processor which stores instruction opcodes, operands and the output of the instruction. When a dynamic instruction which has the same opcode and operands as an entry in the Precomputation Table enters the pipeline, the output is retrieved from the Precomputation Table. This output is then stored in the location where the output of the dynamic instruction would have been stored. The instruction is then removed from the pipeline. This process has effectively bypassed execution of the instruction.

15 different SPEC95 and SPEC2000 benchmarks are profiled to determine the top 2048 arithmetic unique computations. Only two different input sets, labelled Input Set A and Input Set B are used for each benchmark. It is shown that between 13.7% and 44.8% of all dynamic instruction executions are due to the top 2048 arithmetic unique computations.

The top arithmetic unique computations from the Input Set A are then used to populate a Precomputation Table in the simulated hardware that the benchmarks are executed on. Each benchmark is re-executed using Input Set A, with varying sizes of Precomputation Table. The execution of each benchmark is profiled to determine which instructions were bypassed, and this information is used to speculate on the percentage speedup as a result of using the Precomputation Table. Table sizes used were 16, 32, 64, 128, 256, 512, 1024 and 2048. The percentage speedup for each benchmark and Precomputation Table size is presented in a graph, which is convenient for interpretation of the results. It can be seen that in general, larger sizes of Precomputation Table increase the percentage speedup. The percentage speedup is between 4.6% and 12.2%, depending on the size of the Precomputation Table, and the benchmark being executed. Each benchmark is then re-executed and profiled with Input Set B, still using the Precomputation Table from Input Set A. The percentage speedup for each of the benchmarks for Input Set B is slightly lower than that of Input Set A for each size of Precomputation Table.

The authors conclude that this shows that the input set does not determine which are the most frequently executed arithmetic unique computations, but rather this this is a characteristic of the benchmark. Whilst the results do not disagree with this conclusion, more input sets could have been tested to verify this conclusion. There is no discussion of how similar the two input sets were. It may be possible to choose two input sets which have greatly differing sets of most frequently occurring unique arithmetic computations.

A comparison is made between the percentage of speedup gained by using a Precomputation Table and the percentage speedup gained by using an implementation of a Value Reuse Cache. Precomputation Tables with 32, 256 and 2048 entries are compared against Value Reuse Caches with 32, 256 and 2048 entries. Details of how the Value Reuse Cache is implemented are not given. It would have been useful if

the eviction policy of the Value Reuse Cache were documented, as this would aid in making comparisons of the results and conclusions in this paper with those in other papers.

The Precomputation Table is loaded with the precomputations from Input Set A. The benchmarks are executed with Input Set B. It is likely that this is done to compare the worst-case speedup of instruction precomputation with value reuse rather than the best case. The results show that there is a greater percentage speedup obtained by using a precomputation table than by using an equivalent size Value Reuse Cache in almost all cases.

The authors conclude that the benefits of Instruction Precomputation exceed the benefits of Value Reuse, and do so with less hardware complexity. This claim cannot be verified without more information on the implementation of the Value Reuse Cache. However, it is expected that a Value Reuse Cache requires more complex hardware. As the value reuse table is dynamic, extra hardware would be required to manage its contents.

2.7 Value Reuse Optimization: Reuse of Evaluated Math Library Function Calls Through Compiler Generated Cache

(Kumar, 2003)

This paper presents a compiler scheme for reusing the results of function calls made to the math library. The compiler instruments a program with code which implements a cache of the results of function calls. The instrumented code performs a lookup in the cache to determine if the result has previously been computed, to avoid redundant calls to the function. This cache is called the *Function Evaluation History Table* (FEHT).

Kumar states that it is important to consider the difference in the execution times of the instrumented and original versions of the program. This is to avoid the case where the instrumented program takes longer to execute than the original one. The method for managing the contents of the FEHT must be carefully determined. Three potentially viable methods are presented.

The first, round robin, is a simple algorithm which replaces entries in the FEHT using a variable to denote the index of the entry to be replaced. Each time a new entry is to be added, the current entry is replaced with the new entry, and the counter which keeps track of the current entry is incremented by one. If the call placed within a loop, the induction variable of the loop is used as the index to the entry to be replaced.

The second scheme is based on profiling the program to determine which inputs to the math library function occur most frequently. The results of these most frequent calls are then hard-coded into the instrumented program. The modified program performs a lookup in the set of precomputed results to determine if it already has the result of a function call. If it does not have a precomputed result, the function has to be called to compute the result.

Kumar's final scheme is based on a combination of the first two - a lookup table is included for precomputed frequent values, and a round-robin buffer is implemented.

It is stated that the scheme is not unilaterally applied to all math library function calls, as this would almost certainly slow down the execution of the program. A heuristic based on the probability of a high frequency of calls to the math library at a particular call site is used to determine which function calls to instrument.

The effect of the compiler scheme was tested using seven benchmarks. The number of call sites instrumented and the FEHT size for each function are listed in a table. It can be seen that only a small number of call sites (between 1 and 12) are instrumented. The FEHT size is also very small, set at 1 entry for all benchmarks except *alvinn*, which has an FEHT size of 10.

The results presented show that execution time is decreased in most cases by a small percentage - the results are presented in a very small graph so it is difficult to determine exact values. However, it is stated that the maximum decrease in execution time is 6%. Another graph shows that the reduction in the number of calls to the math library is quite dramatic - up to 99% in some cases. However, it seems logical that this does not translate to an increase in speed of 99%, as the program must now spend time searching through and managing the FEHT.

Kumar concludes that the speed increase would be greater if the scheme were implemented in hardware. However, this claim is unverified, and could be the subject of further investigation. It is also concluded that the reasons for the speed increase are:

- A reduction in the number of external library calls.
- Less instructions are executed overall.
- Less external function calls increase the Value Locality of the instruction cache.

All of these conclusions appear to be logical, given the results which have been obtained. Kumar has also suggested that the instrumentation could be applied to user-defined functions, but this is very difficult to analyse by a compiler in languages such as C or C++. This claim is supported with reference to (Huang, 2000).

2.8 Exploiting Frequent Field Values in Java Objects for Reducing Heap Memory Requirements

(Chen *et al.*, 2005)

This paper presents a method to reduce the memory consumption of Java applications by sharing or eliminating field values. The motivation for this work is to enable more complex software to run on embedded platforms, which often have restrictive amounts of memory. The basis of the scheme is similar to that in (Yang & Gupta, 2002), which is that a small number of values are repeatedly found in main memory.

It is stated that compression schemes have previously been used to make more effective use of memory. However, compression schemes are inefficient as memory is partitioned into blocks. Access of a single area of memory requires the whole block to be decompressed. As an alternative, a scheme is proposed which does not compress memory, but instead shares instance variables between objects which have stored similar values in their instance variables.

The distribution of frequent values on the heap is first considered. An instrumented *Java Virtual Machine* (JVM) is used to profile the values stored in instance variables. Eight benchmarks are profiled. The five most frequently occurring values occupy over 80% of all values in all of the benchmarks. Additionally, between 30% and 55% of these values are zero in any given benchmark. A scheme to classify each field according to the potential for sharing its value with other fields is presented.

Level 0 fields do not have a predominant frequent value.

Level 1 fields have a non-zero frequent value.

Level 2 fields frequently store zero values.

A scheme is proposed to optimise the memory usage of Level 2 fields. All Level 2 fields are not stored anywhere if their value is zero. Only when a non-zero value is stored is memory space used in storing the value. A scheme to optimise the memory usage of Level 1 fields is also presented. This involves sharing the fields of multiple Level 1 objects which store the same value. A format is proposed to store the objects with shared fields in memory.

The JVM is modified to implement these two schemes. Each of the benchmarks are profiled again, this time with the JVM running inside a simulator to evaluate its memory usage. The results (in graph form) show that on average, the first scheme reduced the space occupied by objects by 26%, and the second scheme brought a reduction of 38%.

The reduction of heap memory space in total is less than these two percentages as arrays are also stored on the heap. The schemes do not attempt to optimise array memory usage. The average of the reduction in the maximum heap occupancy throughout the execution of the benchmarks is also presented. Scheme 1 gives an average of 7% reduction, whilst Scheme 2 gives a 14% reduction.

Speculation of the performance overhead of the implementation of the schemes is also provided. It is stated that as the execution was performed inside a simulator, it is not possible to obtain completely accurate information on the overhead incurred. It can be seen from the results that the performance overhead of Scheme 1 is generally less than 2%. For a single benchmark, *compress*, the overhead incurred approaches 4%. The average overhead is 1.8%. Scheme 2 has greater overheads. The average overhead is 4.2%, and the maximum overhead, for *mtrt*, is 8.6%.

The authors state that this analysis of the overhead is not likely to accurate for a deeply pipelined superscalar processor. However, as most embedded architectures are not deeply pipelined, the estimate is likely to be accurate in this context.

The conclusion to this paper state that schemes such as the proposed ones may be implemented on embedded architectures to overcome the challenge of limited memory in embedded systems. This is a scheme which exploits Value Locality, as measured in the heap space occupied by objects.

A similar method of reducing the memory space in benchmarks executing inside the LLVM interpreter may be possible to implement. However, it may be very difficult to implement as successfully as for a JVM, as LLVM is not object oriented, so memory usage cannot be directly attributed to particular instances of objects. However, as LLVM supports composite data types (such as structs), memory usage could be attributed to particular instances of these composite data types. This would have the additional challenge of ensuring memory space occupied by structs which were sharing values did not have their memory corrupted by another memory access, as there is nothing to prevent access outside the bounds of a memory location or structure inside LLVM (for example, the `GETELEMENTPTR` instruction is capable of calculating a memory address outside the bounds of an array for a load or store instruction to access).

2.9 Review

There is generally a high degree of invariance in the values stored in memory - A large number of memory locations are occupied by members of a small set of distinct values.

This leads to invariance in other areas of the memory hierarchy, and within instruction executions - For example, the small set of frequent values will directly translate to a small set of values frequently being transferred across the data bus or in cache memory. Further to this, if the majority of data in memory is made up of these frequent values, the instructions which operate on the data will frequently perform the same operations.

The papers and articles reviewed generally have some or all the following goals:

- To determine or estimate the level of invariance in the areas considered (variables, memory locations, instruction executions, function calls, etc.). (Calder *et al.*, 1997), (Yi & Lilja, 2001), (Yang & Gupta, 2002), (Kumar, 2003), (Chen *et al.*, 2005)
- To pinpoint specific areas where there is a high level of invariance (identifying the locations which exhibit invariant behaviour, and perhaps the set of values taken by these locations) (Yang & Gupta, 2002), (Yi *et al.*, 2002), (Kumar, 2003), (Chen *et al.*, 2005)
- To develop applications guided by Value Profile data, or applications which exploit Value Locality. (Sodani & Sohi, 1997), (Yang & Gupta, 2002), (Yi *et al.*, 2002), (Kumar, 2003), (Chen *et al.*, 2005)

As the previous papers managed to meet all the previous three goals successfully, it is expected that these goals can also be met for applications executing on the LLVM and the x86 architectures. Specifically:

- The MiBench benchmarks are expected to exhibit a significant level of Value Reuse. Some benchmarks will exhibit more Value Reuse than others. Some datasets will create conditions more favourable to Value Reuse than others.
- The areas (memory locations/variables/instruction call sites) which exhibit Value Reuse will identifiable.
- Schemes to exploit Value Reuse in the MiBench Benchmarks on LLVM/the x86 architecture can be successfully implemented. Potential applications include the design of a protocol to reduce power consumption of the data bus, and a cache to bypass the execution of instructions which have previously been encountered.

The following section will outline areas which may be investigated to seek the fulfilment of these goals.

Chapter 3

Areas of investigation & Hypotheses

3.1 Summary

There are some "natural levels" at which the compiler/machine operates which could be the subject of value profiling. These are:

Instruction. The smallest unit of computation at which the processor operates. Instructions usually have a small number of inputs and outputs. These inputs and outputs may be registers, memory locations, *immediate values* (values which are statically specified) etc. Some instructions perform special functions, and do not have inputs or outputs, but instead have some side-effect on the state of the processor. Value Profiling of instructions is accomplished by recording the type of instruction, and the values of its inputs at the time of execution. The origin of its inputs is not usually recorded as this does not affect the output of the instruction. The outputs are typically not recorded, as they are only a function of the inputs. The address at which the instruction is stored in memory may also be recorded.

Basic Block. A basic block is a set of instructions which execute sequentially without a branch instruction. Execution always begins at the first instruction in the basic block. Branch instructions in other basic blocks will never jump into the middle of a basic block. A basic block typically consists of less than 10 instructions (Huang & Lilja, 2003). Basic blocks have a potentially unlimited number of inputs and outputs. In practice, because of their small size, they typically have less than 10 inputs and outputs. Value Profiling of basic blocks requires the locations of the inputs and outputs to be determined. The inputs of each basic block are recorded as the basic block executes. As with instructions, the outputs are not typically recorded as they can be determined from the inputs. A unique identifier for the basic block may need to be determined, in order to differentiate between basic blocks. This can be done by recording the address of the first instruction in the basic block.

Trace. A trace is a set of instructions which execute sequentially. Branches are permitted in traces. Value Profiling of traces could be performed using a similar method to the one for profiling basic blocks. However, Value Profiling of traces is more complicated than Value Profiling of basic blocks, because branches are permitted - two traces which begin at the same instruction may not follow the same path of execution, unlike a basic block which always executes the same sequence of instructions.

Function Call. A function is made up of several basic blocks. A function typically computes a result based on some inputs, or performs an operation which has side-effects. In the former case, the result is returned to the callee on the stack. In the latter case, the function will write its outputs to an area other than the stack. Inputs to the function may come from the callee, which passes the inputs on the stack, or from elsewhere. Functions are normally defined in a high-level language(e.g. C, C++, Java etc.). Value Profiling of function calls is performed by recording the inputs to the function and the name of the function. Again the outputs need not be recorded as they are a function of the inputs. The origin of inputs which are not passed on the stack must be determined in order to record all the inputs for Value Profiling. Value Profiling can be simplified by restricting

profiling to only those functions which take all their inputs from the stack. Examples of functions which take inputs from the stack only include those of the math library, as shown in (Kumar, 2003). Additionally, the call site may be of interest, and can be recorded.

Memory access. Values which are loaded from/stored to memory are transferred across the *data bus*. The data bus is of a finite width (typically a number of bits which is a power of 2, e.g. 32, 64, 128 etc.). This limitation in size requires multiple values to be transferred sequentially across the bus. Value Profiling of memory accesses is accomplished by recording the individual values which are transferred across the data bus.

For the purpose of this project, we seek to limit the scope to Value Profiling of instructions and memory accesses.

3.2 Analysing the Usage and Effects of the GETELEMENTPTR Instruction

The LLVM IR provides a special instruction, the `GETELEMENTPTR` (referred to as `GEP` from now on) instruction, which is used to compute the address of a memory location which is accessed through a struct or array. As the scope of this project includes Value Profiling of both instruction executions and memory locations, the effect of this instruction is considered in detail as it will have an effect on both of these areas.

3.2.1 Allocation of Storage Space

It is down to the compiler to decide when a store instruction is issued:

```
int a;
a=1;
```

Compiles to:

```
%a = alloca i32, align 4
store i32 1, i32* %a
```

The compiler will allocate space in memory for a variable and then access that space using a pointer with the same name as the identifier of the variable in the source code.

3.2.2 Introducing the GETELEMENTPTR Instruction

The `GEP` instruction is required for computation of the address when using an index into an array, or accessing a variable within a struct. If we access a memory location as if it were an array, like so:

```
int *a;
a[0]=1;
```

Then the compiler is forced to use a `GEP` operation to compute the address of the zeroth element in the array pointed to by a.

```
%a = alloca i32*, align 4
%tmp = load i32** %a
%tmp1 = getelementptr i32* %tmp, i32 0
store i32 1, i32* %tmp1
```

Variable names are unique (due to the LLVM IR using SSA form[1]) in each assignment. This `GEP` instruction adds on the correct number of bytes to the address stored in $\%tmp$. As we are accessing the first element in the array, 0 is added on. However, if the third element were being accessed:

```
a[2]=1;
```

[1] A program is in Static Single Assignment (SSA) form if there is a single assignment statement for each variable in the program (Johnson, 2004).

The GEP instruction generated would be:

```
%tmp1 = getelementptr i32* %tmp, i32 2
```

It should be noted that this would add on eight bytes, as each `i32` will occupy four bytes. The GEP instruction calculates the offset correctly as the type of the variable in the array is specified as well as the quantity.

The other use of GEP is to compute the address of a member of a struct. A minimal example of C code to illustrate this:

```
struct lev1 {
        int a;
        int b;
};

void test() {
        struct lev1 d;
        d.a = 2;
        d.b = 5;
}
```

The relevant portions of the LLVM IR code output from the compiler are:

```
%struct.lev1 = type { i32, i32 }

%d = alloca %struct.lev1, align 8
%tmp = getelementptr %struct.lev1* %d, i32 0, i32 0
store i32 2, i32* %tmp
%tmp1 = getelementptr %struct.lev1* %d, i32 0, i32 1
store i32 5, i32* %tmp1
```

The *lev*1 struct has to be defined in the code so that the GEP instruction is familiar with its form and uses this information in the computation of addresses. This example is similar to that of the array shown earlier, as the GEP instruction only computes the address of $d.a$ by adding `i32 0` to the base address of d, and by adding `i32 1` to the base address to compute $d.b$.

3.2.3 Compilation of the GETELEMENTPTR Instruction

To examine what a GEP instruction becomes when compiled natively, the following C extract will be used:

```
b = 3;                                                  // Line 1
a[2] = 5;                                               // Line 2
a[b] = 6;                                               // Line 3
```

When compiled to LLVM IR this becomes:

```
store i32 3, i32* %b                                    // Line 1
%tmp2 = load i32** %a                                   // Line 2
%tmp3 = getelementptr i32* %tmp2, i32 2                 //      |
store i32 5, i32* %tmp3                                 //      v
%tmp4 = load i32** %a                                   // Line 3
%tmp5 = load i32* %b                                    //      |
%tmp6 = getelementptr i32* %tmp4, i32 %tmp5             //      |
store i32 6, i32* %tmp6                                 //      v
```

When compiled to x86 assembly this becomes (in AT&T syntax):
(Noting that a = `8(%esp)`, b = `4(%esp)` to make to code easier to understand)

```
movl    $3, 4(%esp)                                  // Line 1
movl    8(%esp), %eax                                // Line 2
movl    $5, 8(%eax)                                  //      v
movl    4(%esp), %eax                                // Line 3
movl    8(%esp), %ecx                                //      |
movl    $6, (%ecx,%eax,4)                            //      v
```

Compiling the LLVM IR code to ARM assembly instead results:
(Noting that a = `[sp, #+4]` and b = `[sp]` to make the code easier to understand)

```
mov r3, #3                                           // Line 1
str r3, [sp]                                         //      v
mov r3, #5                                           // Line 2
ldr r2, [sp, #+4]                                    //      |
str r3, [r2, #+8]                                    //      v
mov r3, #6                                           // Line 3
ldr r2, [sp]                                         //      |
ldr r1, [sp, #+4]                                    //      |
str r3, [r1, +r2, lsl #2]                            //      v
```

- Line 1 does not use the GEP instruction, it only stores 3 into the location b. This is done in one instruction on the x86, on line 1 of the assembly output. As the ARM architecture only allows instructions (excluding the `ldr` and `str` instructions) to operate on registers, this requires two steps to store the value 3 into memory.

- Line 2 uses the GEP instruction to compute the address of the element at index 2 of the array a. The index given to the GEP instruction is `i32 2`, which is equal to the base address plus 8 bytes. This arithmetic is accomplished on the third line of the x86 assembly code. EAX has already been loaded with the base address of the array a. 8 is added to this address as the destination to be used by the `movl` instruction. The ARM code accomplishes the operation in a similar manner, adding 8 to $r2$, which has been loaded with the base address of the array a to compute the destination.

- Line 3 is different because before optimisations the compiler cannot see that b is a constant and instead must perform arithmetic using the value of b as if it were variant. The LLVM IR code uses the GEP instruction, but with an index that is a variable argument (`i32 %tmp5`) which contains the value that is in b. The x86 assembly code works in a similar manner, loading EAX with the value stored in b, and ECX with the base address of the array a. To compute the destination address, the contents of EAX are multiplied by 4 (for an `i32`) and added to the address stored in ECX. The ARM computes the address of the destination again in the same manner, loading $r2$ with the contents of b and $r1$ with the address of a. Subsequently it multiplies the contents of $r2$ by 4 (for the `i32`) by left-shifting two bits. The ARM uses the left shift as its organisation permits the operand passing through a barrel shifter before being passed to the *Arithmetic Logic Unit* (ALU) (Knaggs & Welsh, 2004).

- The compiled code works in almost exactly the same manner for both architectures, the only slight differences being due to the architecture of the different targets. The code was also compiled for the Thumb instruction set for ARM, but is not listed here as it is functionally equivalent to the ARM code, only being slightly more conservative with register use.

To examine what GEP operations for a struct become, the code below assumes that d is a `struct lev1` as defined above:

```
d.a = 2;                                             // Line 1
d.b = 5;                                             // Line 2
```

Which becomes in LLVM IR:

```
%tmp = getelementptr %struct.lev1* %d, i32 0, i32 0     // Line 1
store i32 2, i32* %tmp                                  //      v
%tmp1 = getelementptr %struct.lev1* %d, i32 0, i32 1    // Line 2
store i32 5, i32* %tmp1                                 //      v
```

And in x86 assembly:
(Noting that `(%esp)` is currently the base address of the struct d)

```
movl    $2, (%esp)                                      // Line 1
movl    $5, 4(%esp)                                     // Line 2
```

And in ARM assembly:
(Noting that `[sp]` is currently the base address of the struct d)

```
mov r3, #2                                              // Line 1
str r3, [sp]                                            //      v
mov r3, #5                                              // Line 2
str r3, [sp, #+4]                                       //      v
```

This example is simpler, as the struct is defined at the beginning of the code, the GEP instruction knows the index to use to find each element of the struct. The only arithmetic necessary is adding the correct offset to the base address of the struct. It can be seen in the assembly language code that there is no addition to the base address of d to calculate the address of element a, and to calculate the address of element b, 4 is added onto the base address in line 2. The ARM assembly works similarly, but requires two instructions per store again due to its architecture.

3.2.4 Use of the GEP Instruction in the MiBench Benchmarks

As the GEP instruction has a potentially unlimited number of operands, an LLVM pass was produced to record the number of indices of each GEP instruction in an LLVM bitcode file. This pass was run on all the compiled bitcode files of the MiBench benchmarks.

Across all the benchmarks, the maximum number of indices that any GEP instruction has is 8, which is in *consumer-lame*. However, the majority of GEP instructions have only one or two indices. In most of the benchmarks there is only a small number of GEP instructions with more than two indices.

3.2.5 Observations

The GEP instruction is used to compute the address of the required element of a pointer or struct. The instruction only makes computations, it does not dereference pointers (Spencer, 2008). The GEP instruction, when compiled, becomes arithmetic operations with pointers only. *The GEP operation provides a reason for a strong correlation between invariance in memory accesses, and invariance in computations.* This is significant because:

- If a memory location is repeatedly accessed, the computation of that memory location is likely to be performed repeatedly.
- If the same memory location is repeatedly accessed, subsequent computations using the contents of this memory location are likely to be repeated throughout the execution of the program.
- If the same location is repeatedly accessed due to repeated results computed by the GEP instruction, the values travelling across the address and memory buses are also likely to have a high degree of invariance.

However, this does not mean that the GEP instruction is the only reason for this correlation: (Yang & Gupta, 2002) showed that only a very small number of values (up to 8) occupy up to 48% of all memory locations. Therefore it is also possible that the GEP instruction need not compute the same locations repeatedly for a program to exhibit a high degree of invariance in computations and the values transferred across the memory bus.

3.3 Hypotheses

Hypothesis 1. Value Reuse is prevalent in instruction executions and memory accesses throughout the execution of most programs.

Hypothesis 2. There is a greater level of Value Reuse at the global level[4] than there is at the local level[4] in instruction executions.

Hypothesis 3. There is a greater level of Value Reuse at the global level[5] than there is at the local level[5] in memory accesses.

Hypothesis 4. There is a correlation between Value Reuse in instruction executions, and Value Reuse in memory accesses. This hypothesis has been formed based on the conclusions of investigating the `GEP` instruction.

Hypothesis 5. It is possible to exploit Value Reuse in Instruction Executions to improve performance in terms of decreasing execution time, either through increasing Instruction-Level Parallelism or decreasing the number of cycles required to execute a single instruction.

Hypothesis 6. It is possible to exploit value reuse in memory accesses to decrease power consumption.

Hypothesis 7. As the LLVM IR is architecture independent, Value Profile data collected by executing a particular program using the LLVM interpreter is representative of its execution on all architectures.

3.4 Methods of investigation

Each of the hypotheses will be considered in the contexts of the LLVM infrastructure, and the x86 architecture. The MiBench benchmarks will be used, with input datasets from the MiDatasets suite.

Hypothesis 1 will be investigated by performing the following:

- Performing Value Profiling of Instruction Executions.
- Performing Value Profiling of Memory Accesses.
- Examination will be made of the Value Profile data for both Instruction Executions and Memory Accesses. The examinations will be conducted with the goal of determining if repeated use is made of the same values in either of these areas.
- If it is found that there is a significant level of Value Reuse in both Instruction Executions and Memory Access, then it will be considered that this hypothesis is supported by the evidence.

Hypothesis 2 will be investigated by performing the following:

- Performing Local-level Value Profiling of Instruction Executions.
- Performing Global-level Value Profiling of Instruction Executions.
- Developing and testing a scheme to exploit Global-level Value Reuse.
- Modifying and testing the scheme to only exploit Local-level Value Reuse.
- If it is found that there is a significantly greater level of Value Reuse in Instruction Executions at the global level than at the local level then it will be considered that the hypothesis is supported by the evidence.
- Additional support will be added to the hypothesis if the Global-level Value Reuse Exploitation Scheme is more successful at exploiting Value Reuse than the Local-level Value Reuse Exploitation Scheme.

[4]Global-level Value Reuse in Instruction Executions does not consider the Program Counter. Local-level Value Reuse does consider the Program Counter (i.e. two executions of the same instruction opcode and operands with differering Program Counter values are considered different operations).

[5]Global-level Value Reuse in Memory Accesses does not consider the location in memory of the value being accessed. Local-level does consider the location of the value (i.e. transferring the same value from two different locations in memory are considered two different operations).

Hypothesis 3 will be investigated by performing the following:

- Performing Local-level Value Profiling of Memory Accesses.
- Performing Global-level Value Profiling of Memory Accesses.
- If it is found that there is a significantly greater level of Value Reuse in Memory Accesses at the global level than at the local level then it will be considered that the hypothesis is supported by the evidence.

Hypothesis 4 will be investigated by performing the following:

- The Value Profile data gathered whilst investigating Hypothesis 1 will be analysed to determine if there is a correlation between Value Reuse in Instruction Executions and Value Reuse in Memory Accesses.
- If it is found that benchmarks which exhibit high levels of Value Reuse in Instruction Executions also exhibit high levels of Value Reuse in Memory Accesses, then it will be considered that the evidence supports the hypothesis.

Hypothesis 5 will be investigated by performing the following:

- Examining the test results of the schemes to exploit Instruction Level Value Reuse developed to investigate Hypothesis 2.
- If there is a significant benefit of either or both of these schemes, then it will be considered that the evidence supports the hypothesis.

Hypothesis 6 will be investigated by performing the following:

- Investigating potential methods of encoding bus traffic to reduce power.
- An encoding scheme will be proposed.
- Testing the encoding scheme against the benchmarks will be outside the scope of this project. This could be performed as further work.
- As a result, no evidence to support or disprove Hypothesis 6 will be gathered.

Hypothesis 7 will be investigated by performing the following:

- A comparison will be made between Value Profile data recorded using implementations of Value Profiling on the LLVM Infrastructure and the x86 architecture.
- An attempt may be made to determine a tranformation which allows the prediction of Value Reuse on the x86 architecture from the Value Profile Data recorded using the LLVM Infrastructure
- If such a transformation can be found, and tested appropriately, then it will be considered that Value Profile Data recorded using LLVM is representative of Value Profile Data which may be recorded on other architectures, and consequently the hypothesis is supported.
- Further work to support this hypothesis could be done by recording Value Profile Data on other architectures (ARM, MIPS etc.) and attempting to determine if a similar transformation could be made from the LLVM Value Profile Data to Value Profile/Value Reuse Information for those other architectures.

In order to investigate these hypotheses, implementations of Value Profiling were developed for LLVM and the x86 architectures. These implementations are discussed in the following chapter.

Chapter 4

Implementation

4.1 Development methodology - The Waterfall Lifecycle

The Waterfall Lifecycle is a sequence of tasks which are specified in the development of a system. The five stages of the lifecycle (from (Whiteley, 2004)) are:

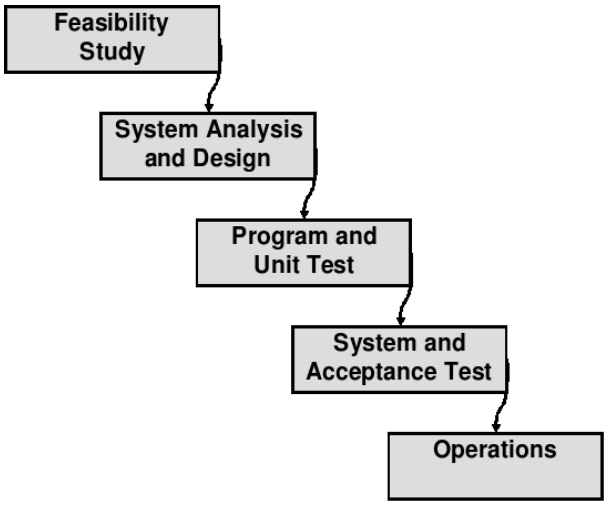

Figure 4.1: The Waterfall Lifecycle.

The waterfall lifecycle is described in the context of developing an *Information System* (IS) for an organisation. As the goal of this project is not to develop an IS for an organisation, but instead to modify existing tools to implement Value Profiling, certain tasks in the lifecycle will not be performed as they are unnecessary. Each stage of the lifecycle is as follows:

Feasibility Study. The following aspects of the feasibility study will be considered in the development of tools within the scope of this project:

- Consideration of the technical feasibility of the system, in order to determine whether the proposed system can and will work.
- Production of an outline of the system to be developed. This will include specifying what the system will not do.
- A technical summary of the required software/hardware.

Some tasks traditionally involved in the feasibility study will not be considered, as they are not required within the scope of this project. These are:

- A financial justification of the system.

- Consideration of the ethical acceptability of the system.
- Production of a costing of the system.

A plan of how the system may be implemented is sometimes be produced at this stage. However, as this implementation will not be a large system, any plans of how to implement the system will all be placed in the System Analysis & Design stage.

System Analysis & Design. Tasks to be performed within this stage include:

- An analysis of the requirements.
- A logical design of the system meeting the requirements.
- A technical design based on the logical design.

Methods of analysing requirements which are normally used when developing an IS include interviews, observation (of staff) and questionnaires. These are inappropriate for this project and will not be used. Examination of the current system will be performed, as the Value Profiling tools will be developed on top of existing tools.

This stage will have the following outputs:

- Diagrams summarising the design of the system.
- Specifications of any hardware/software required.
- UML diagrams specifying any new code to developed, and an overview of how this code will be integrated into existing tools.

Program & Unit Test. This stage consists of the implementation in a programming language of the technical specification produced in the previous stage. As the system under development is relatively small compared to an IS which would normally be developed using the Waterfall Lifecycle, testing will be performed in the following stage.

System & Acceptance Test. The following tasks are part of this stage:

- Development of test cases.
- Determination of expected results of each test.
- Execution of the test cases using the developed system.
- A comparison between the expected result and the actual result. When the expected result differs from the actual result, this is indicative of a bug. In this case, action must be taken to resolve the error. This could include correcting the code of the system, correcting the test case, or re-analysing the test case to ensure that the expected result is correct.

The acceptance test is a test of the system by the users. This will not be considered in this project, as the system is not developed for a set of users, but instead to be used as a tool to gather Value Profile data.

Operations. This stage is the use of the system. Most parts of this stage are unnecessary in this system, including:

- Training of maintenance and support staff - there are no maintenance/support staff.
- Training of users - the system is not developed for a set of users, so there are no users to train.
- Loading the data on to the new system. As the system developed is not an information system, but is a tool to gather Value Profile data, there is no pre-existing data to load onto the system.

This stage within this project will consist of the execution of benchmarks using the developed tools, and making a permanent record of the gathered Value Profile data.

4.2 Feasibility Study

4.2.1 Technical Feasibility

The system for Value Profiling will be developed by extending existing tools. For this to be possible, the following requirements must be met:

1. Source code for the existing tools or a suitable API must be provided.
2. Access to a compiler which can compile this code must be available.
3. The existing systems must be written in a manner which can support their extension for Value Profiling - i.e. the code must be straightforward enough to understand, and there must be logical points where Value Profiling code can be inserted.

Requirement 1 is met by both LLVM and Pin.

- LLVM meets this requirement as it is open-source, and full C++ source code is provided for the LLVM Compiler Infrastructure.
- Pin meets this requirement. Although source to Pin is not available, it provides an API for the dynamic instrumentation of executing binaries. This is sufficient to insert code which records Value Profile data at any point necessary in the execution of the program.

Requirement 2 is met by both LLVM and Pin. A C++ compiler, *g++*, is available. This is sufficient to compile LLVM. Tools implemented using the Pin API are written in C++, which can be compiled using g++.

Requirement 3 is met by both LLVM and Pin.

- The LLVM Interpreter, which executes code in the form of LLVM IR, has a function to perform the execution of each type of instruction. Value Profiling of Instruction Executions can easily be added to the functions which execute those instructions which are to be profiled. Two functions exist which perform memory operations. These two functions are a logical point to insert code which performs Value Profiling of memory access.
- The Pin API provides facilities to examine the state of the registers every time a particular type of instruction is executed. This allows a tool to be written which records the operands of profiled instructions. Additionally, the API allows every memory access to be examined. This will allow recording of the values which are transferred across the data bus.

Therefore, all of these requirements are all met by both LLVM and Pin.

4.2.2 An outline of the systems

The systems developed will allow the following operations to be performed:

- Execution of a program compiled to LLVM IR/an x86 binary.
- Throughout the execution of the program, instructions and their operands, and potentially the program counter will be recorded. A count of the number of similar sets of inputs will be kept, rather than storing duplicate information.
- When execution of the program terminates, the data collected will be output as a CSV file which contains the following information about each unique set: Number of occurrences, instruction opcode, operands, and the program counter, if it had been recorded.
- Alternatively, instead of recording attributes of executing instructions, the value transferred across the memory bus may be recorded. The memory location involved in the memory operation, and which direction the memory access was in may be recorded.

- In the case of storing values transferred across the memory bus, at the end of the execution a CSV file will still be output. The information regarding each unique set will in this case be: Number of occurrences, whether the operation was a load or store, the value transferred across the bus, and the memory location.

The system to be developed will only be for gathering the Value Profile data. It will not perform any processing on this data. Instead, the full data is output so that it may potentially be transformed and analysed in multiple ways.

4.2.3 Required Hardware/Software

The following hardware and software will be required:

- A standard PC. A large amount of memory will be required (4GB) as the execution of a program frequently involves the execution of hundreds of millions of instructions. As information regarding a significant proportion of these instructions will be recorded, a large amount of memory will be used. As the LLVM Interpreter will be likely to execute programs very slowly compared to the execution of machine code, a fast processor is also desirable. The Pin manual (Luk *et al.*, 2008) states that the processor must support the SSE2 instruction set in order for Pin to function. A Pentium 4 or above meets this requirement.
- A large amount of disk space (100GB) will be required to store the value profile data which is output for each dataset for each benchmark.
- The PC should be running the Linux operating system. Pin requires that the system comes with a runtime which allows an LD_ASSUME_KERNEL value of 2.4.1 (Luk *et al.*, 2008).
- An up-to-date GNU C++ compiler must be installed on the machine. The LLVM Getting Started Guide (Criswell *et al.*, 2008) states that it is demanding of the compiler, so the compiler must be recent and not have certain bugs which are listed in the guide.

4.3 System Analysis & Design

4.3.1 Requirements Analysis for Value Profiling of Computations at the Instruction Level

(Yi *et al.*, 2002) and (Yi & Lilja, 2001) have previously investigated Value Profiling at this level. Parts of the *Simplescalar Tool Set* (Burger & Austin, 1997) were instrumented to record Value Profile data. The Simplescalar Tool Set, the LLVM infrastructure, and the x86 architecture all use different instruction sets. Therefore the LLVM and x86 instruction sets must be examined to determine which instructions should be profiled to investigate this area, as it is not possible to use the instructions documented in the previous work.

4.3.2 Choosing instructions of the LLVM IR to profile

The LLVM Assembly Language Reference Manual (Lattner & Adve, 2008) groups instructions into several categories:

- Terminator Instructions
- Binary Operations
- Bitwise Binary Operations
- Vector Operations
- Memory Access/Addressing Operations
- Conversion Operations

- Other Operations

To determine which of these instructions should be profiled, information on the distribution of different types of dynamic instructions presented in (Guthaus *et al.*, 2001) will be used.

Terminator Instructions are those instructions which end a Basic Block. These include instructions such as branch instructions and return instructions. This type of instruction only accounts for between 2.5% and 20% of dynamic instructions across all the benchmarks in the MiBench suite. Compared to the other types of instruction, this represents only a small fraction of all instructions, so these will not be profiled.

Binary Operations include: `ADD`, `SUB`, `MUL`, `UDIV`, `SDIV`, `FDIV`, `UREM`, `SREM` and `FREM`. (Guthaus *et al.*, 2001) states that up to 82% of all dynamic executions are due to integer or floating-point operations. The majority of benchmarks have between 50% and 60% of all dynamic executions from integer operations only. Binary operations encompass most of the integer operations and all of the floating-point operations in the LLVM instruction set. These instructions should all be profiled. It appears that most of the MiBench benchmarks do not execute any floating point operations. However, the benchmarks which do execute floating point operations have a significant fraction of dynamic instructions from floating point operations, so both the integer and floating-point versions of these instructions should be profiled.

Bitwise Binary Operations include: `SHL`, `LSHR`, `ASHR`, `AND`, `OR`, `XOR`. All of these operations are integer operations and consequently they should all be profiled, as Binary Operations.

Vector Operations should not be profiled. Few, if any embedded architectures support vector operations. The MiBench benchmarks are a set of benchmarks for embedded applications (Guthaus *et al.*, 2001). Therefore, it is unlikely that they will contains any vector operations.

Memory Access/Addressing. One of these instructions is of interest: the `GEP` instruction, which was discussed in Section 3.2. (Sodani & Sohi, 1997) found that a high level of reuse is present in address calculation instructions. Value Profile data including the `GEP` instruction may be used to determine if a similar amount of reuse of address calculation exists in the MiBench benchmarks.

Conversion Operations are those which convert from one type to another (casting). These will not be profiled.

Other Operations describes those operations which did not fit any other category. `ICMP` and `FCMP` are the two compare operations, for integer and floating-point operations respectively. (Guthaus *et al.*, 2001) states that conditional branches make up around 10% of all dynamic instruction executions. As there is only a small number of conditional branches, it is likely that there is only a small number of comparison operations which precede these instructions. Therefore, they will not be profiled as it is unlikely that they will make up a significant fraction of dynamic executions. The `PHI` instruction implements the ϕ node in SSA form. The ϕ node behaves like a conditional assignment operator, based on which path the executing program previously took (Johnson, 2004). `PHI` instructions will not be profiled. The `SELECT` instruction implements a simple binary predicate operator. The output of the `SELECT` instruction will be one of the two values of its scalar inputs, depending on whether its boolean input is true or false. This instruction will not be profiled, as examining the disassemblies of each benchmark shows that no select instructions are present in most of the executables. The `CALL` and `VA_ARG` instructions relate to function calls, which are not being profiled.

Table 4.1 shows all instructions which will be profiled on LLVM. Table 4.2 shows all instructions which will not be profiled on LLVM.

4.3.3 Choosing x86 Instructions to Profile

The x86 instruction set is far more complex than the LLVM IR. Hundreds of different instructions are usable on a modern Pentium 4 processor. A consideration at this stage of the design is that the integer operations are executed in a different way to floating-point operations. This is due to the legacy

ADD	SUB	MUL	FDIV	SDIV
UDIV	FREM	SREM	UREM	AND
OR	XOR	SHL	ASHR	LSHR
GETELEMENTPTR				

Table 4.1: Instructions which will be profiled on LLVM for Instruction-level Value Profiling.

Terminator	Vector	Memory Access	Conversion	Other
RET	EXTRACTELEMENT	MALLOC	TRUNC	ICMP
BR	INSERTELEMENT	FREE	ZEXT	FCMP
SWITCH	SHUFFLEVECTOR	ALLOCA	SEXT	PHI
INVOKE		LOAD	FPTRUNC	SELECT
UNWIND		STORE	FPEXT	CALL
UNREACHABLE			FPTOUI	VA_ARG
			FPTOSI	GETRESULT
			UITOFP	
			SITOFP	
			PTRTOINT	
			INTTOPTR	
			BITCAST	

Table 4.2: Instructions which will not be profiled on LLVM for Instruction-level Value Profiling

design of the Pentium 4 processor, which is based on the Intel 80386 original design. In the 80386, the integer operations were performed on the main processor, and an optional floating-point arithmetic processor could be installed, the 80387. Because of this asymmetric design, floating point instructions would execute differently to integer operations. To maintain compatibility with the 80386, the Pentium 4 works in the same way, even though the floating-point processor is on the same die as the main processor. To keep the design of the Pin-based Value Profiling implementation straightforward, only profiling of integer operations will be implemented initially. If it is found that frequently-executed integer operations only represent a small fraction of all instruction executions, then profiling of floating-point operations will be subsequently implemented.

Because of the large number of instructions, not all will be listed here. For a full list of instructions on the x86 architecture, see (Intel, 2007). Only instructions to be profiled are listed here. Instructions to be profiled are:

Unary Arithmetic Operations. These include the `INC` and `DEC` instructions. These increment or decrement a single operand by 1. As these are likely to be frequently used by loop counters, they will be profiled.

Binary Arithmetic Operations. These include `ADC`, `ADD`, `DIV`, `MUL`, `IMUL` and `IDIV`. These instructions perform similar functions to the *Binary Operations* in the LLVM IR as described in the previous section and consequently will be profiled.

Bitwise Binary Operations. These include `RCL`, `RCR`, `ROL`, `ROR`, `SAL`, `SAR`, `SHL`, `SHR`, `AND`, `OR`, and `XOR`. These instruction are similar to be *Bitwise Binary Operations* described in the section above and will be profiled.

Table 4.3 shows all instructions which will be profiled on the x86 architecture. Instructions which are not profiled are not shown as there are too many to list.

ADC	ADD	AND	DEC	DIV	IDIV	IMUL
INC	MUL	NOT	OR	RCL	RCR	ROL
ROR	SAL	SAR	SHL	SHR	SUB	XOR

Table 4.3: Instructions which will be profiled on the x86 architecture.

4.3.4 Additional attributes to record - LLVM & x86 implementations

Recording the instruction opcodes alone will be of little benefit. Additional information regarding each dynamic instruction execution must be stored.

All papers describing Instruction-level Value Profiling stated that the the operands of each dynamic instruction were stored along with the opcode. The operands are relevant as they are characteristics of a dynamic instruction which determine its output. Without comparing the operands of two instructions of the same opcode, it is not possible to know whether they will both produce the same output. Therefore, the proposed implementations will also record the operands of the instruction. In order to test Hypothesis 2, an option to record the Program Counter at the execution of every instruction must be provided.

All of the attributes stored regarding a single instruction execution will be referred to as its *input set*. The input set of a dynamic instruction will be used as the basis for comparison with another dynamic instruction. If all of the attributes of one input set do not match all of the attributes of the second input set, the input sets are not considered equal. There are some instructions which have commutativity over their operands. Where an instruction is commutative over its operands, the implementation must recognise that the order of operands is unimportant for comparison. The full input set of a dynamic instruction will comprise:

{Program Counter, Opcode, Operand 1, Operand 2}

As we are merely seeking to determine the potential for Value Reuse in Instruction Executions, the output of the instruction is not stored as it is simply a function of the opcode and its operands. Throughout the execution of a program being profiled the frequency of each unique input set will be recorded. At the end of the execution, all of the input sets and their frequency of occurrence will be output to a specified CSV file on disk.

4.3.5 Requirements Analysis for Value Profiling of Memory Accesses

(Yang & Gupta, 2002) previously investigated Value Profiling at this level. The following information will be required to be stored about each memory access:

- Value transferred.
- Type of the Value being transferred.
- The address at which the transferred value resides.
- The direction of the transfer (whether it is a load or a store).
- The frequency of the transfer throughout the execution of the program.

The direction of the transfer is recorded as it may be interesting to see what the ratio of loads to stores is. Additionally, the location of the transferred value will also be recorded so that it is possible to determine the distribution of a particular value in memory if necessary. This is necessary for Local-level Value Profiling of Memory Accesses. The type of the value being transferred across the bus may also be of interest - it is only possible to record this on LLVM, as the interpreter stores type information at runtime. At runtime on the x86 architecture, the type information has been discarded, so it is only possible to consider the actual value transferred, not what it represents.

4.3.6 A Design of Classes to Record Value Profile Data

Normally in this stage a logical design would first be produced, and then a technical design which is independent of the logical design would be produced. However, in this project the logical and technical designs are closely tied due to the implementation being based on existing tools. Several classes have been designed which meet the requirements to store Value Profile data of Instruction Executions and Memory Accesses within the LLVM & Pin infrastructures. Figure 4.2 shows a UML diagram of the classes which store Value Profile data of all Instruction Executions on the LLVM Interpreter, apart from the GEP instruction.

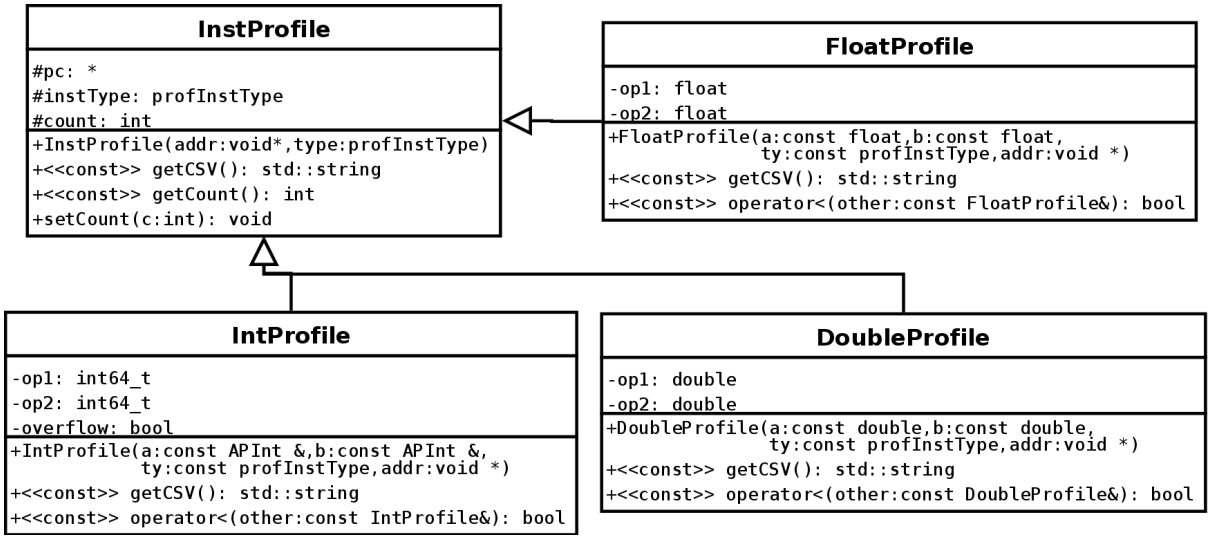

Figure 4.2: InstProfile class and subclasses to profile all Binary Operations using LLVM.

The InstProfile class stores attributes which are common to instructions which have integer, float and double operands. This includes the address of the instruction (in Local-level Value Profiling), the type of instruction, and a count of the number of executions of instructions matching this profile. The `getCSV()` function returns a string representation of the InstProfile object. The count member variable may need to be read/changed by other parts of the code, so a setter and getter are provided.

The IntProfile, FloatProfile and DoubleProfile subclasses each store the operands of the instruction using the correct representation - e.g. an IntProfile object stores the operands as the `int64_t` type. The `getCSV()` function is overridden by each of these classes, and it adds additional information regarding the operands to the value returned from the `getCSV()` function of the superclass. The `operator<()` function is provided in each class as the STL `set` requires this function for sorting elements in a `set` (Hewlett-Packard, 1994).

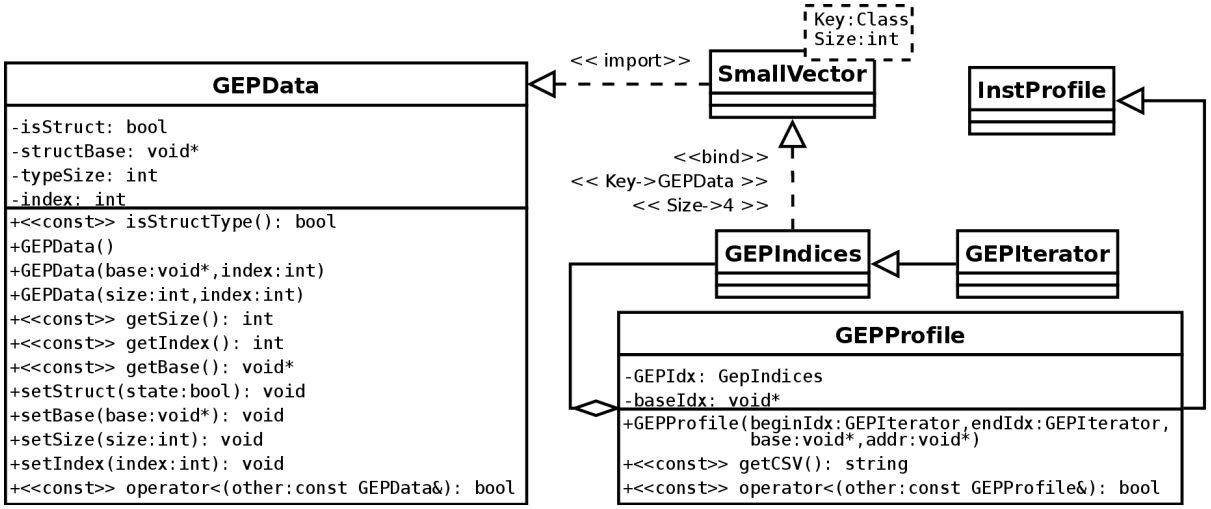

Figure 4.3: Classes involved in profiling the GEP instruction using LLVM.

Figure 4.3 shows the classes involved in storing the Value Profile data of GEP instructions. Detail of the InstProfile object is not shown on this class diagram as it is already shown in Figure 4.2. The GEPData class stores the profile of a particular pair of arguments to the GEP instruction. As the number of arguments to the GEP instructions are potentially unlimited (though seldom greater than 8), a type is defined (GEPIndices) which is a SmallVector of GEPData objects. The SmallVector class is provided

within the LLVM infrastructure (Lattner *et al.*, 2008), and is used similarly to an STL Vector (Hewlett-Packard, 1994). The SmallVector was chosen in this case because it is more efficient for small collections of objects. The GEPIterator class is provided to allow sequential iteration over all of the elements of a GEPIndices. The GEPProfile object is a subclass of InstProfile, which also stores a GEPIndices object, to record all of the arguments to the GEP instruction. The `operator<()` method is again provided to allow the object to be used as the Key in an STL `set`.

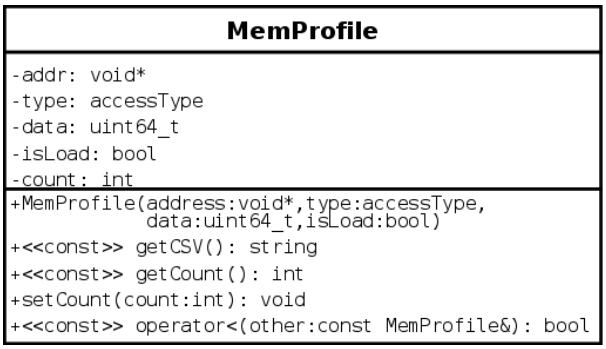

Figure 4.4: The MemProfile class for Value Profiling Memory Accesses using LLVM.

Figure 4.4 is a UML diagram of the MemProfile object which stores the Value Profile of a Memory Access. This does not inherit from the InstProfile object, as its characteristics are slightly different. Attributes which this object store include the address which the value is loaded from/stored to in Local-level Value Profiling, the type of value being transferred (int, double etc.), whether the operation is a load or store, and a count of memory accesses matching this profile. The functions provided are similar to those of the same names already discussed for the other objects.

Figure 4.5 is a UML diagram of the class which stores all the profile data during the execution of the program. An STL `set` is used to store all the objects of each type of Profile. The ProfData class stores all of these `set`s. The `insert()` method is provided so that new objects may be inserted into any of the sets storing Value Profile data. The `setFile()` procedure is provided so that a pointer to an open `ofstream` can be passed to the class. The `outputProfile()` procedure writes the profile to disk by calling the `getCSV()` function of every object in each set, and writing the returned values through the `ofstream` object.

Adapting the LLVM Profiling classes for use with Pin

The classes to store Value Profile data of Instruction Executions can be considerably simplified for use with Pin. As only integer operations are to be profiled, there is no need for the FloatProfile and DoubleProfile classes. Also on the x86 architecture, there is no GEP instruction, so the GEPProfile and GEPIndices classes are also not required. This only leaves a single subclass of InstProfile, the IntProfile class. This has been merged with the InstProfile class to simplify the design. The ProfileData class is also simplified, as it no longer required methods for the insertion of instructions which do not have integer operands, and does not need multiple sets for all the classes which were removed. The Memory Access Value Profiling code has been separated from the instruction execution profiling code, as the Pin API is designed to allow the creation of multiple distinct tools to perform separate tasks, whereas the LLVM interpreter is a single program, to which all the instrumentation code must be added. Figure 4.6 shows the modified classes which store Value Profile data of Instruction Executions.

The MemProfile class is slightly simplified for use with Pin. As the type of the value cannot be determined, the `type` member variable has been removed. A separate (to the Instruction Execution Value Profiling code) ProfileData class is implemented to store MemProfile objects in a `set`. Figure 4.7 shows the modified classes to store Value Profile data of Memory Accesses.

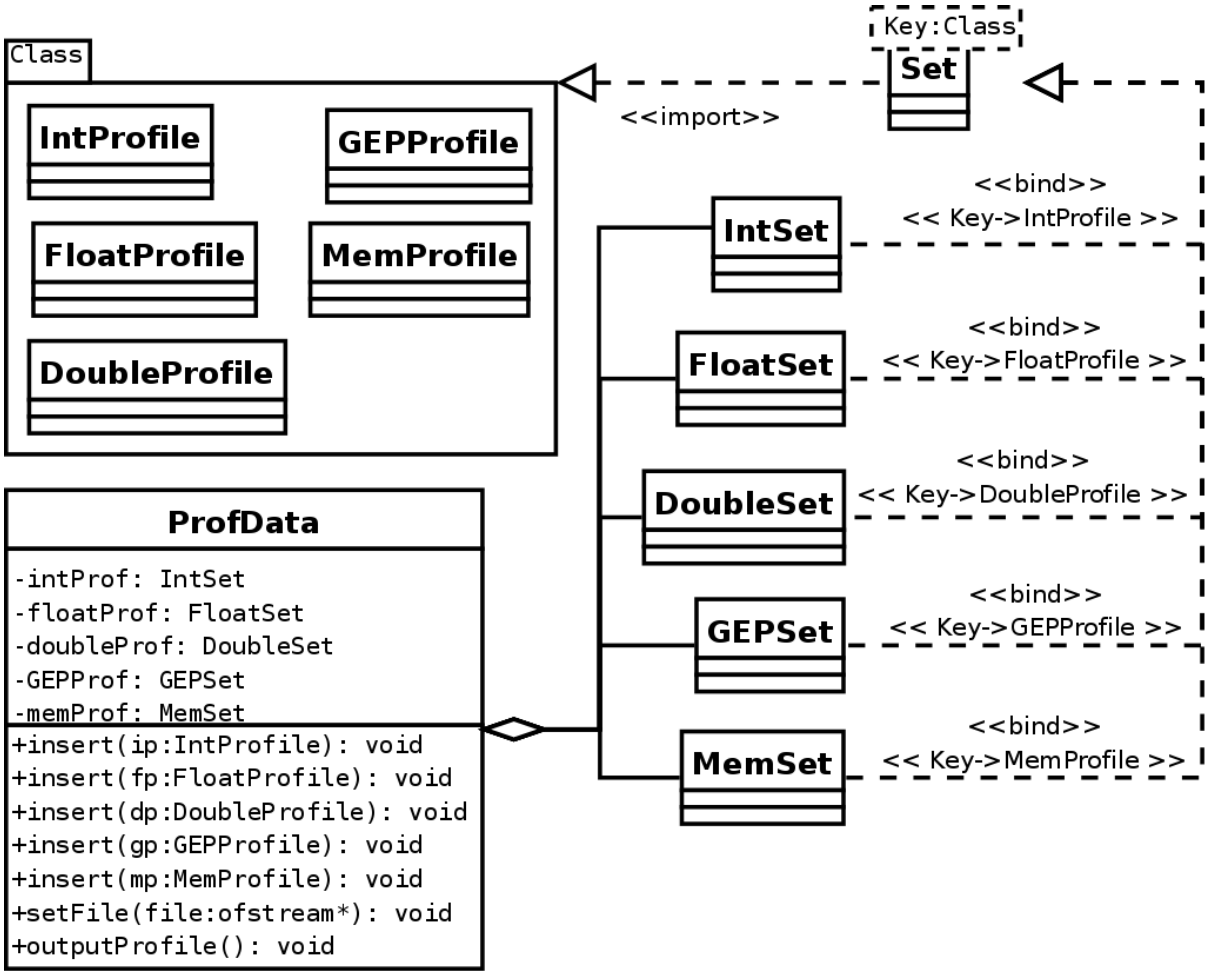

Figure 4.5: The ProfData class and STL sets of profiling classes used with LLVM.

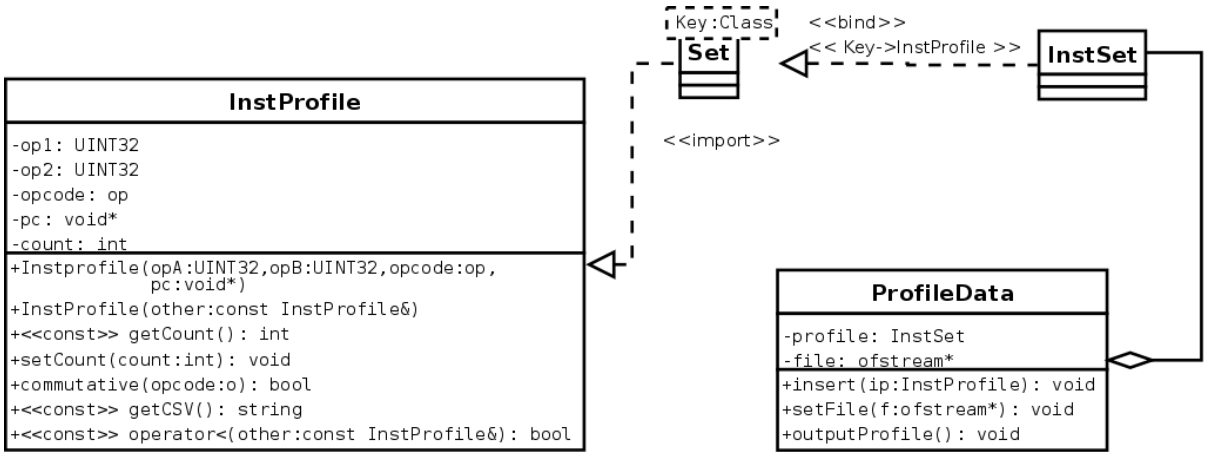

Figure 4.6: Classes involved in Value Profiling Instruction Executions on the x86 architecture using Pin.

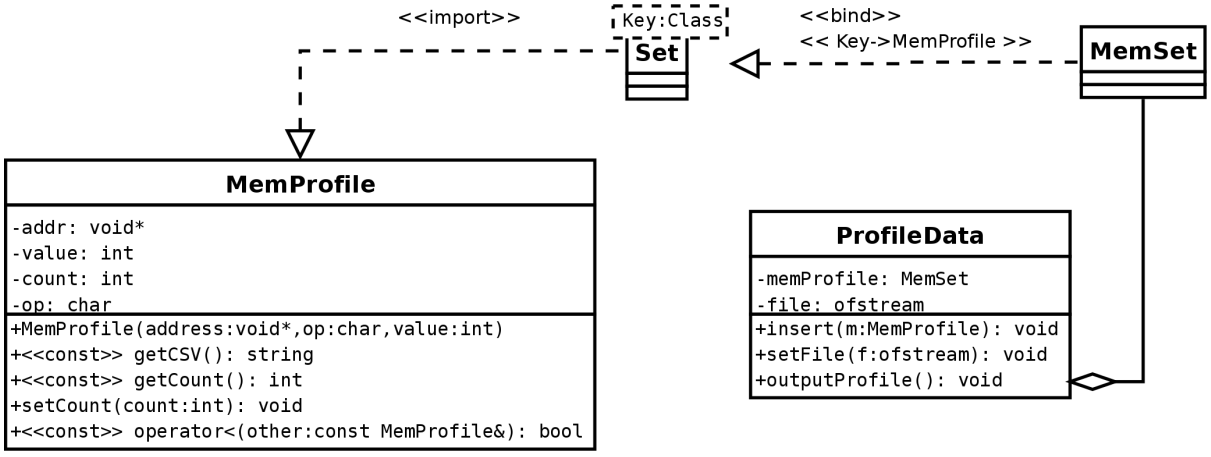

Figure 4.7: Classes involved in Value Profiling Instruction Executions on the x86 architecture using Pin.

Opcode	Function	Opcode	Function
ADD	executeAddInst()	FDIV	executeFDivInst()
SUB	executeSubInst()	SDIV	visitBinaryOperator()
MUL	executeMulInst()	UDIV	visitBinaryOperator()
FREM	executeFRemInst()	AND	visitBinaryOperator()
SREM	visitBinaryOperator()	OR	visitBinaryOperator()
UREM	visitBinaryOperator()	XOR	visitBinaryOperator()
SHL	visitShl()	LSHR	visitLShr()
ASHR	visitAShr()	GEP	executeGEPOperation()

Table 4.4: Functions which perform the execution of opcode in the LLVM Interpreter.

4.3.7 Inserting Instrumentation Code on LLVM

Value Profiling of Instruction Executions

The source file containing the code which interprets LLVM executables is `Execution.cpp` which is stored relative to the root of the LLVM source tree in the `lib/ExecutionEngine/Interpreter` folder. Functions which perform the execution of each instruction opcode on the LLVM IR are provided in this file. Each of these functions is passed the values of the operands of the current instruction. These functions provide a natural location into which Value Profiling instrumentation can be inserted. Table 4.4 shows the functions which execute each profiled opcode.

An example of a function which performs an instruction execution (in this case `visitLShr()`) before the insertion of Value Profiling instrumentation:

```
void Interpreter::visitShl(BinaryOperator &I) {
  ExecutionContext &SF = ECStack.back();
  GenericValue Src1 = getOperandValue(I.getOperand(0), SF);
  GenericValue Src2 = getOperandValue(I.getOperand(1), SF);
  GenericValue Dest;
  Dest.IntVal = Src1.IntVal.shl(Src2.IntVal.getZExtValue());
  SetValue(&I, Dest, SF);
}
```

The same function after the insertion of Value Profiling code:

```
void Interpreter::visitShl(BinaryOperator &I) {
  ExecutionContext &SF = ECStack.back();
  GenericValue Src1 = getOperandValue(I.getOperand(0), SF);
  GenericValue Src2 = getOperandValue(I.getOperand(1), SF);
```

```
  GenericValue Dest;
  if(InstructionProfiling)
    ProfData::insert(IntProfile(Src1.IntVal, Src2.IntVal, SHL, (void*)&I));
  Dest.IntVal = Src1.IntVal.shl(Src2.IntVal.getZExtValue());
  SetValue(&I, Dest, SF);
}
```

`InstructionProfiling` is a boolean value which controls whether Value Profiling of Instruction Executions is performed. This value is set by using a command-line switch. If Value Profiling of Instruction Executions is turned on, then the `insert()` method of the ProfData class is called, which records a new IntProfile object storing the profile of the current execution. The first two arguments to the IntProfile constructor are the values of the operands. The third argument, `SHL`, is to record the instruction opcode. The fourth argument is the reference to the instruction object being executed, cast to a void pointer. This is to record the location of the instruction, which is of interest in Local-level Value Profiling of Instruction executions.

All other functions are instrumented in a similar way. The third argument is changed in each function to reflect the opcode of the instruction being profiled. The type of the operands does not need to be stated explicitly, as this is determined by whether an IntProfile, FloatProfile or DoubleProfile object is constructed and passed as an argument of the `insert()` method.

Value Profiling of Memory Accesses

Code to perform Value Profiling of Memory Accesses is designed in a similar way. There are only two instructions which access memory in LLVM, which are the `LOAD` and `STORE` instructions. The functions which carry out these operations are `visitLoadInst()` and `visitStoreInst()`. An example of one of these instructions before instrumentation:

```
void Interpreter::visitLoadInst(LoadInst &I) {
  ExecutionContext &SF = ECStack.back();
  GenericValue SRC = getOperandValue(I.getPointerOperand(), SF);
  GenericValue *Ptr = (GenericValue*)GVTOP(SRC);
  GenericValue Result;
  LoadValueFromMemory(Result, Ptr, I.getType());
  SetValue(&I, Result, SF);
}
```

The same function after instrumentation:

```
void Interpreter::visitLoadInst(LoadInst &I) {
  ExecutionContext &SF = ECStack.back();
  GenericValue SRC = getOperandValue(I.getPointerOperand(), SF);
  GenericValue *Ptr = (GenericValue*)GVTOP(SRC);
  GenericValue Result;
  LoadValueFromMemory(Result, Ptr, I.getType());
  if(MemoryProfiling) {
    switch((int)I.getType()->getTypeID()) {
      case 7:  // Integer type
        if(Result.IntVal.getBitWidth()<=64)
          ProfData::insert(MemProfile(Ptr, INT, Result.IntVal.getZExtValue(), true));
        break;
      case 1:  // Float type
        ProfData::insert(MemProfile(Ptr, FLOAT, (uint64_t)Result.FloatVal, true));
       break;
      case 2:  // Double type
        ProfData::insert(MemProfile(Ptr, DOUBLE, (uint64_t)Result.DoubleVal, true));
        break;
      case 12: // Pointer type
```

```
        ProfData::insert(MemProfile(Ptr, PTR, (uint64_t)Result.PointerVal, true));
        break;
    }
  }
  ++NumLoads;
  SetValue(&I, Result, SF);
}
```

`MemoryProfiling` is a boolean variable set by command-line switch to control whether Value Profiling of Memory Accesses is performed. If this variable is true, then the code determines the type of value being transferred using the `switch` statement. The `insert()` method of the ProfData class is called to insert a new MemProfile object which records the profile of this memory access. The first argument to the MemProfile constructor is a pointer to the location involved in the memory access. The second argument is to record the type of the value transferred. The third argument is the actual value transferred, and the fourth argument is `true` as this operation is a load. Note that the Value Profiling code is added after the call to `LoadValueFromMemory()`. This is important because the correct value would not be recorded if the instrumentation was inserted before the value had actually been loaded from memory.

The `visitStoreInst()` function is instrumented in a similar way. A difference is that the fourth argument to the MemProfile constructor is always `false`, to record that the operation was a store and not a load. Profiling code is inserted before the value is stored to memory, rather than afterwards.

4.3.8 Inserting Instrumentation Code on the x86 Architecture

Value Profiling of Memory Accesses

The standard distribution of Pin comes with the source code to many example tools. One of these is *pinatrace*, a tool which records memory accesses, and outputs details of each memory access to standard output. A drawback of this tool is that a very large amount of data is quickly produced. Using this tool to instrument `/bin/ls` produces an output of several megabytes of text. Larger programs will easily output gigabytes of unsorted data. A solution to this problem is to use the classes already described to store Value Profile data of Memory Accesses until the end of the execution.

Code which outputs each memory access to screen must be replaced with code which records the profile in a new MemProfile object, and inserts this new object into the set of Memprofile objects. The function concerned is the `RecordMem()` function in `pinatrace.cpp`:

```
static VOID RecordMem(VOID * ip, CHAR r, VOID * addr, INT32 size, BOOL isPrefetch)
{
    TraceFile << ip << ": " << r << " " << setw(2+2*sizeof(ADDRINT)) << addr << " "
              << dec << setw(2) << size << " "
              << hex << setw(2+2*sizeof(ADDRINT));
    if (!isPrefetch)
        EmitMem(addr, size);
    TraceFile << endl;
}
```

The `RecordMem` function is called every time an instruction which performs a memory access is executed. Pin passes the current Program Counter, whether the operation is a read or write, the address of the value to be read/written, and the size (number of bytes to be transferred) of the transfer. The `EmitMem()` function outputs the values involved in the memory access to standard output. It does this by reading from the memory locations starting at the value specified by the `addr` variable, continuing until the number of bytes specified by the `size` variable have been read. The `RecordMem` function is modified to:

```
static VOID RecordMem(VOID * ip, CHAR r, VOID * addr, INT32 size, BOOL isPrefetch)
{
    unsigned int *address = static_cast<unsigned int*>(addr);
```

```
    if(size==0) {
      zeroSize++;
      return;
    }

    if(size>4)
      ++moreThanFour;

    // Section A
    for(int i=0; i<size/4; ++i) {
      prof.insert(MemProfile(KnobLocal?&address[i]:(void*)(0),r,address[i]));
      ++regularBounds;
    }

    // Section B
    if(size%4!=0) {
      void *lastAddr = address+(size%4);
      unsigned int value=0;
      ++irregularBounds;
      switch(size%4) {
        case 1:
  value = static_cast<UINT32>(*static_cast<UINT8*>(lastAddr));
  break;
case 2:
  value = *static_cast<UINT16*>(lastAddr);
  break;
case 3:
  value = (*static_cast<UINT32*>(lastAddr)) >> 8;
  break;
      }
      prof.insert(MemProfile(KnobLocal?lastAddr:(void*)(0),r,value));
    }
}
```

The first `if` statement is to check that an access of size zero is not taking place. If the size of the read is zero, there is no work for this function to do. The second `if` statement checks to see if more than 4 bytes (32 bits) are being transferred. This is because it is assumed that the data bus is 32 bits wide, and that transferring more than 32 bits in a single memory access requires the value to be broken up into multiple 32 bit segments for transfer across the bus. A loop (in Section A of the above code) records each 32 bit chunk of the memory value transferred in a new MemProfile object. Section B records any remaining bytes if there are less than four left. In order to record the remaining bytes correctly, the address of the last 1, 2 or 3 bytes to be read is cast to the correct size so that the correct number of bytes are read from memory. When there are 1 or 2 bytes remaining, this is straightforward: 8 or 16 bytes are read from the memory. However, it is not possible to explicitly read three bytes from memory. In this case, four bytes are read from memory, starting at the address of the first byte. This value is then shifted 8 bits right, to discard the (garbage) 8 bits which were read after the three bytes we are interested in.

The first argument passed to the MemProfile constructor is the address of the value transferred. `KnobLocal` is a boolean variable controlled by command-line switch to determine whether Local-level Value Profiling is performed. If this is set to true, then the address of the transferred value is passed to the MemProfile constructor, otherwise 0 is passed. The second argument is a `char`, which is set to `'R'` for a read and `'W'` for a write. The third argument is the value transferred.

Additionally the `RecordMem()` function records statistics on the types of transfers: The number of zero-size transfers, the number of transfers of more than four bytes, the number of times exactly four bytes are transferred across the bus, and the number of times that 1, 2 or 3 bytes are transferred across the bus. These statistics show that an overwhelming majority of the time, exactly 4 bytes are transferred

across the bus. Transfers which are of less than 4 bytes are insignificant when considering the Value Profile of the whole program.

Value Profiling of Instruction Executions

There is not a suitable tool included with the standard Pin distribution which could easily be modified to record the opcodes and operands of instructions. Instead it was decided to that a tool would need to be written from scratch.

When the JIT Compiler in Pin is running, each time a new instruction is encountered, Pin calls back to the PinTool with details of the instruction to add instrumentation. An example of code which instruments the `DEC`, `INC` and `NOT` instructions follows:

```
if(opcode==DEC || opcode==INC || opcode==NOT) {
  if(REG_valid(reg1) && REG_is_gr32)
    INS_InsertCall(
      ins, IPOINT_BEFORE, (AFUNPTR)CacheReg,
      IARG_INST_PTR,
      IARG_UINT32, (UINT32)opcode,
      IARG_REG_CONST_REFERENCE, reg1,
      IARG_END);
  if(INS_OperandIsMemory(ins,0))
    INS_InsertCall(
      ins, IPOINT_BEFORE, (AFUNPTR)CacheMem,
      IARG_INST_PTR,
      IARG_UINT32, (UINT32)opcode,
      IARG_MEMORYREAD_EA,
      IARG_MEMORYREAD_SIZE,
      IARG_END);
}
```

The instrumentation code first tests to see if the opcode is one which is to be instrumented. Subsequently, the type of the operands is determined. The operand for these three instructions may be either a register or memory location. Depending on what the operand is, the `INS_InsertCall()` method instructs Pin to instrument the instruction with either the `CacheReg()` function or the `CacheMem()` function. After the instrumentation of all instructions is complete, the program is executed by Pin. Execution of these instrumented instructions will cause control to be passed back to either the `CacheReg()` or `CacheMem()` function. The `CacheReg()` function follows:

```
static void CacheReg(void *ip, op o, PIN_REGISTER *reg1) {
  ValueProfile.insert(new InstProfile(reg1->dword[0],0,o,ip));
}
```

Pin passes the Program Counter, opcode, and a pointer to an object storing the contents of the register to the `CacheReg()` function. This function calls the `insert()` method of `ValueProfile`, which is an instance of the ProfileData class. The first argument passed to the InstProfile constructor is the value of the first operand. The second argument is the value of the second operand. In the case of the instructions in this example, there is no second operand so this value is set to 0. The third argument of the constructor is the opcode, which has been passed to the `CacheReg()` function through the variable `o`. The fourth argument is the value of the Program Counter, which is necessary when Local-level Value Profile data is being recorded.

Functions to instrument other instructions work in a similar fashion. However, these other instructions may have two or more operands, so extra code is required to determine the type and value of the second operand. These functions are not presented in this report as the method by which the second operand type and value is determined is similar to that already seen.

4.4 Program & Unit Test

This stage consisted of implementing the technical design in a programming language. The testing of the developed code is described in the next section as the Waterfall Lifecycle has been modified for the development of this system.

At this stage a revision was made to the design, after it was found that the code ran very slowly when performing Value Profiling of Instruction Executions. This was suspected to have been because the `set` which stored each InstProfile object had to be searched every time a single instruction was executed. As the `set` grows very large (occupying over 1GB of RAM in many cases), the working set of memory will also grow large, and the cache performance of the processor is likely to be very poor as values will frequently be evicted from the cache.

In order to counter this, a buffer was implemented which stores InstProfile objects before they are inserted into the set. After a certain number of instruction profiles have been recorded (specifiable on the command line, defaulting to 100000), the InstProfile objects are all processed at once. The operation of the buffer is as follows:

- At the beginning of the execution, an array of InstProfile objects is initialised. The size of this array is equal to the number of InstProfile objects which will be stored in the buffer.
- Each time a profiled instruction is executed, the new InstProfile object which is generated is stored in the next free location in the array. The position of the next free location in the array is maintained by storing a counter which begins at 0 and is incremented each time a new InstProfile object is stored in the array.
- Once the array is full, every InstProfile object in the array is inserted into a temporary `set` of InstProfile objects. Each time an InstProfile object is inserted into the `set` which is considered similar to an already existing InstProfile object in the `set`, the count member variable of the existing InstProfile is incremented, instead of storing duplicate InstProfile objects in the `set`.
- Once all the InstProfile objects have been inserted into the temporary `set`, the objects in the array are all deleted and the counter is re-set to zero. Additionally, the items in the temporary `set` are sequentially inserted into the main `set` of InstProfile objects. If an InstProfile which is inserted from the temporary `set` is considered similar to an InstProfile which already exists in the main `set`, then the count of the two similar InstProfile objects is summed and the count member variable of the InstProfile object in the main `set` is updated with this new count.

It was found during testing of the buffer that many similar InstProfile objects are inserted into the buffer. When these similar objects are all inserted into the small temporary `set`, they are converted into a single InstProfile object which is inserted into the main `set`. The operation of the buffer therefore greatly reduces the total number of insertions into the large main `set`. 100000 potential insertions into the main `set` are reduced to several hundred insertions through the use of the buffer.

This reduction in the number of insertions into the main `set` reduces execution time, as the large main `set` takes much longer to search through every time an InstProfile is inserted. Even for a small benchmark, the execution time is reduced. An example of the time taken to execute *automotive-susan-c* on LLVM before the implementation of the buffer:

```
graham@grahamspc:~/project/midatasets/automotive_susan_c/src$ time ./__run 1

real    4m10.281s
user    4m10.152s
sys     0m0.088s
```

After implementation of buffer:

```
graham@grahamspc:~/project/midatasets/automotive_susan_c/src$ time ./__run 1

real    3m56.744s
user    3m55.783s
sys     0m0.384s
```

For larger benchmarks which execute more instructions, the reduction in execution time is expected to be larger as more instructions will be executed throughout the life of the benchmark, leading to a larger main `set` which would require more time to search through at every insert.

4.5 System & Acceptance Test

Test cases have been developed to ensure that the operation of the code which performs Value Profiling of Instruction Executions and Memory Accesses have been developed is correct. A full list of the test cases is given in Appendix B. The test cases have not been developed to test the operation of the LLVM Interpreter or Pin in any way - it is assumed that these function correctly in normal operation. For each test case an expected output from the Value Profiling tools has been developed by considering the execution of the compiled code and manually stepping through the program. Not all test cases were used with all types of Value Profiling as some were inappropriate. For each type of Value Profiling, the expected output is presented, and whether the test passed or failed.

As Local-level Value Profiling of Instruction Executions was not performed and no results were produced (see chapter 5), the results of testing these functions have been omitted.

4.5.1 Global-level Instruction Profiling on LLVM

Test cases 14, 15 and 16 were skipped as they produce a lot of output for this profiling method, and therefore do not constitute minimised test cases. The format of the expected results is similar to the output CSV file from the profiling code. Each line represents a single unique computation. The columns of each line give the frequency, instruction opcode, operand type, operand 1 value and operand 2 value respectively. An exception to this format is when a GEP instruction is recorded, which has a similar format to other instructions, except there are frequently more than two operands. Numeric values are given in decimal format, as the output of these profiling tools gives numeric values in decimal format.

Table 4.5: Testing of Global-level Instruction Profiling on LLVM

Test Case	Expected result	Pass
1	1: ADD INT 1 2	✓
2	2: ADD INT 1 2	✓
3	1: SUB INT 1 2 1: INT SUB 2 1	✓
4	1: ADD INT 0 1 1: ADD INT 1 1 1: ADD INT 1 2 1: ADD INT 1 3 1: ADD INT 1 4 1: ADD INT 1 5 1: ADD INT 1 6 1: ADD INT 1 7 1: ADD INT 1 8 1: ADD INT 1 9	✓
5	1: ADD INT 1 2 1: ADD FLOAT 1 2	✓
6	1: ADD DOUBLE 1 2	✓
7	1: MUL INT 1 2	✓
8	1: MUL DOUBLE 1 2	✓
9	1: SUB DOUBLE 1 2	✓
10	1: FDIV DOUBLE 1 2	✓
11	1: ADD INT 1 1 3: ADD INT 1 2	✓
12	1: GEP (Address A) ARR 4 0 1: GEP (Address A) ARR 4 5	✓

13	2: GEP (Address A) ARR 4 0 2: GEP (Address A) ARR 4 5	✓
14	Skipped	
15	Skipped	
16	Skipped	
17	2: ADD INT 2 3	✓
18	No output	✓
19	1: GEP ARR (Address A) 4 2	✓
20	1: GEP (Address A) ARR 8 0 STR (Address B) 0 1: GEP (Address A) ARR 8 0 STR (Address B) 1	✓
21	1: GEP (Address A) ARR 8 0 STR (Address B) 1 GEP (Address A) ARR 4 1	✓

4.5.2 Global-level Memory Profiling on LLVM

Test cases 14, 15 and 16 were skipped as they produce a lot of output for this profiling method, and therefore do not constitute minimised test cases. The format of the expected results is similar to the output CSV file from the profiling code. Each line represents a single memory access. The columns of each line give the frequency, type of access (load/store), value type, and the value transferred respectively.

Table 4.6: Testing of Global-level Memory Profiling on LLVM

Test Case	Expected result	Pass
1	1: LOAD INT 1 1: LOAD INT 2 1: STORE INT 1 1: STORE INT 2 1: STORE INT 3	✓
2	2: LOAD INT 1 2: LOAD INT 2 1: STORE INT 1 1: STORE INT 2 2: STORE INT 3	✓
3	2: LOAD INT 1 2: LOAD INT 2 2: STORE INT 1 1: STORE INT 2 1: STORE INT 4294967295	✓
4	2: LOAD INT 0 2: LOAD INT 1 2: LOAD INT 2 2: LOAD INT 3 2: LOAD INT 4 2: LOAD INT 5 2: LOAD INT 6 2: LOAD INT 7 2: LOAD INT 8 2: LOAD INT 9 1: LOAD INT 10 1: STORE INT 0 1: STORE INT 1 1: STORE INT 2 1: STORE INT 3 1: STORE INT 4 1: STORE INT 5	✓

	1: STORE INT 6 1: STORE INT 7 1: STORE INT 8 1: STORE INT 9 1: STORE INT 10	
5	1: LOAD INT 1 1: LOAD FLOAT 1 1: LOAD INT 2 1: LOAD FLOAT 2 1: STORE INT 1 1: STORE FLOAT 1 1: STORE INT 2 1: STORE FLOAT 2 1: STORE INT 3 1: STORE FLOAT 3	✓
6	1: LOAD DOUBLE 1 1: LOAD DOUBLE 2 1: STORE DOUBLE 1 1: STORE DOUBLE 2 1: STORE DOUBLE 3	✓
7	1: LOAD INT 1 1: LOAD INT 2 1: STORE INT 1 2: STORE INT 2	✓
8	1: LOAD DOUBLE 1 1: LOAD DOUBLE 2 1: STORE DOUBLE 1 2: STORE DOUBLE 2	✓
9	1: LOAD DOUBLE 1 1: LOAD DOUBLE 2 1: STORE DOUBLE 1 1: STORE DOUBLE 2 1: STORE DOUBLE 18446744073709551615	✓
10	1: LOAD DOUBLE 1 1: LOAD DOUBLE 2 1: STORE DOUBLE 0 1: STORE DOUBLE 1 1: STORE DOUBLE 2	✓
11	4: LOAD INT 1 4: LOAD INT 2 1: LOAD INT 3 2: STORE INT 1 2: STORE INT 2 3: STORE INT 3	✓
12	2: LOAD PTR (Address A) 1: STORE PTR (Address A) 1: STORE INT 1 1: STORE INT 39	✓
13	4: LOAD PTR (Address A) 1: LOAD INT 1 1: LOAD INT 39 1: STORE PTR (Address A) 2: STORE INT 1 2: STORE INT 39	✓
14	Skipped	

15	Skipped	
16	Skipped	
17	2: LOAD INT 2 2: LOAD INT 3 1: STORE INT 2 1: STORE INT 3 2: STORE INT 5	✓
18	1: LOAD PTR (Address A) 1: STORE PTR (Address A) 1: STORE INT 1	✓
19	1: LOAD PTR (Address A) 1: STORE PTR (Address A) 1: STORE INT 1	✓
20	1: STORE INT 2 1: STORE INT 5	✓
21	1: LOAD PTR (Address A) 1: STORE PTR (Address A) 2: STORE INT 5	✓

4.5.3 Local-level Memory Profiling on LLVM

Test cases 14, 15 and 16 were skipped as they produce a lot of output for this profiling method, and therefore do not constitute minimised test cases. The format of the expected results is similar to the output CSV file from the profiling code. Each line represents a single memory access. The columns of each line give the frequency, address of the value, type of access (load/store), value type, and the value transferred respectively.

Table 4.7: Testing of Local-level Memory Profiling on LLVM

Test Case	Expected result	Pass
1	1: (Address X) LOAD INT 1 1: (Address Y) LOAD INT 2 1: (Address X) STORE INT 1 1: (Address Y) STORE INT 2 1: (Address Z) STORE INT 3	✓
2	2: (Address X) LOAD INT 1 2: (Address Y) LOAD INT 2 1: (Address X) STORE INT 1 1: (Address Y) STORE INT 2 2: (Address Z) STORE INT 3	✓
3	2: (Address X) LOAD INT 1 2: (Address Y) LOAD INT 2 1: (Address X) STORE INT 1 1: (Address Z) STORE INT 1 1: (Address Y) STORE INT 2 1: (Address Z) STORE INT 4294967295	✓
4	2: (Address X) LOAD INT 0 2: (Address X) LOAD INT 1 2: (Address X) LOAD INT 2 2: (Address X) LOAD INT 3 2: (Address X) LOAD INT 4 2: (Address X) LOAD INT 5 2: (Address X) LOAD INT 6 2: (Address X) LOAD INT 7 2: (Address X) LOAD INT 8	✓

	2: (Address X) LOAD INT 9 1: (Address X) LOAD INT 10 1: (Address X) STORE INT 0 1: (Address X) STORE INT 1 1: (Address X) STORE INT 2 1: (Address X) STORE INT 3 1: (Address X) STORE INT 4 1: (Address X) STORE INT 5 1: (Address X) STORE INT 6 1: (Address X) STORE INT 7 1: (Address X) STORE INT 8 1: (Address X) STORE INT 9 1: (Address X) STORE INT 10	
5	1: (Address X) LOAD INT 1 1: (Address U) LOAD FLOAT 1 1: (Address Y) LOAD INT 2 1: (Address V) LOAD FLOAT 2 1: (Address X) STORE INT 1 1: (Address U) STORE FLOAT 1 1: (Address Y) STORE INT 2 1: (Address V) STORE FLOAT 2 1: (Address Z) STORE INT 3 1: (Address W) STORE FLOAT 3	✓
6	1: (Address X) LOAD DOUBLE 1 1: (Address Y) LOAD DOUBLE 2 1: (Address X) STORE DOUBLE 1 1: (Address Y) STORE DOUBLE 2 1: (Address Z) STORE DOUBLE 3	✓
7	1: (Address X) LOAD INT 1 1: (Address Y) LOAD INT 2 1: (Address X) STORE INT 1 1: (Address Y) STORE INT 2 1: (Address Z) STORE INT 2	✓
8	1: (Address X) LOAD DOUBLE 1 1: (Address Y) LOAD DOUBLE 2 1: (Address X) STORE DOUBLE 1 1: (Address Y) STORE DOUBLE 2 1: (Address Z) STORE DOUBLE 2	✓
9	1: (Address X) LOAD DOUBLE 1 1: (Address Y) LOAD DOUBLE 2 1: (Address X) STORE DOUBLE 1 1: (Address Y) STORE DOUBLE 2 1: (Address Z) STORE DOUBLE 18446744073709551615	✓
10	1: (Address X) LOAD DOUBLE 1 1: (Address Y) LOAD DOUBLE 2 1: (Address Z) STORE DOUBLE 0 1: (Address X) STORE DOUBLE 1 1: (Address Y) STORE DOUBLE 2	✓
11	2: (Address U) LOAD INT 1 2: (Address V) LOAD INT 1 2: (Address W) LOAD INT 2 2: (Address X) LOAD INT 2 1: (Address Y) LOAD INT 3 1: (Address U) STORE INT 1 1: (Address W) STORE INT 2	✓

	1: (Address Y) STORE INT 3 1: (Address V) STORE INT 1 1: (Address X) STORE INT 2 2: (Address Z) STORE INT 3	
12	2: (Address X) LOAD PTR (Address A) 1: (Address X) STORE PTR (Address A) 1: (Address Y) STORE INT 1 1: (Address Z) STORE INT 39 Where (Address Y)=(Address Z)+0x14	✓
13	4: (Address X) LOAD PTR (Address A) 1: (Address Y) LOAD INT 1 1: (Address Z) LOAD INT 39 1: (Address X) STORE PTR (Address A) 1: (Address Y) STORE INT 1 1: (Address V) STORE INT 1 1: (Address Z) STORE INT 39 1: (Address W) STORE INT 39	✓
14	Skipped	
15	Skipped	
16	Skipped	
17	2: (Address X) LOAD INT 2 2: (Address Y) LOAD INT 3 1: (Address X) STORE INT 2 1: (Address Y) STORE INT 3 1: (Address W) STORE INT 5 1: (Address Z) STORE INT 5	✓
18	1: (Address X) LOAD PTR (Address A) 1: (Address X) STORE PTR (Address A) 1: (Address Y) STORE INT 1	✓
19	1: (Address X) LOAD PTR (Address A) 1: (Address X) STORE PTR (Address A) 1: (Address Y) STORE INT 1	✓
20	1: (Address X) STORE INT 2 1: (Address Y) STORE INT 5	✓
21	1: (Address X) LOAD PTR (Address A) 1: (Address X) STORE PTR (Address A) 2: (Address Y) STORE INT 5	✓

4.5.4 Global-level Instruction Profiling on the x86 Architecture

Test cases 14, 15 and 16 were skipped as they produce a lot of output for this profiling method, and therefore do not constitute minimised test cases. As the stack is set up at the beginning and end of the `main` function, an additional three computations (an `ADD`, `AND` and a `SUB`) are recorded during the execution of every test case. These have been excluded from the expected outputs for brevity. The format of the expected results is similar to the output CSV file from the profiling code. Each line represents a single memory access. The columns of each line give the frequency, instruction opcode, operand 1 value, and operand 2 value respectively. Numeric values are now given in hexadecimal format, as the output of these profiling tools gives numeric values in hexadecimal format.

Table 4.8: Testing of Global-level Instruction Profiling on the x86 Architecture

Test Case	Expected result	Pass
1	1: ADD 1 2	✓
2	2: ADD 1 2	✓

3	1: SUB 1 2 1: SUB 2 1	✓
4	1: INC 0 1: INC 1 1: INC 2 1: INC 3 1: INC 4 1: INC 5 1: INC 6 1: INC 7 1: INC 8 1: INC 9	✓
5	1: ADD 1 2	✓
6	No output.	✓
7	1: IMUL 1 2	✓
8	No output.	✓
9	No output.	✓
10	No output.	✓
11	2: ADD 1 2 1: INC 1 1: INC 2	✓
12	1: ADD (Address A) 20	✓
13	2: ADD (Address A) 20	✓
14	Skipped	
15	Skipped	
16	Skipped	
17	2: ADD 2 3	✓
18	No output.	✓
19	1: ADD (Address A) 8	✓
20	No output.	✓
21	1: ADD (Address A) 4	✓

4.5.5 Global-level Memory Profiling on the x86 Architecture

Test cases 14, 15 and 16 were skipped as they produce a lot of output for this profiling method, and therefore do not constitute minimised test cases. Additionally cases 6, 9 and 10 were also skipped as they produce output which is too complex to be considered a minimised test case. As the stack is set up at the beginning and end of the main function, an additional three reads and three writes are recorded during the execution of every test case, which are omitted here for brevity. The format of the expected results is similar to the output CSV file from the profiling code. Each line represents a single memory access. The columns of each line give the frequency, type of access (read/write), and the value transferred respectively.

Table 4.9: Testing of Global-level Memory Profiling on the x86 Architecture

Test Case	Expected result	Pass
1	1: READ 1 1: READ 2 1: WRITE 1 1: WRITE 2 1: WRITE 3	✓
2	2: READ 1 2: READ 2 1: WRITE 1	✓

	1: WRITE 2 2: WRITE 3	
3	2: READ 1 2: READ 2 2: WRITE 1 1: WRITE 2 1: WRITE FFFFFFFF	✓
4	2: READ 0 2: READ 1 2: READ 2 2: READ 3 2: READ 4 2: READ 5 2: READ 6 2: READ 7 2: READ 8 2: READ 9 1: READ 10 1: WRITE 0 1: WRITE 1 1: WRITE 2 1: WRITE 3 1: WRITE 4 1: WRITE 5 1: WRITE 6 1: WRITE 7 1: WRITE 8 1: WRITE 9 1: WRITE A	✓
5	1: READ 1 1: READ 3F800000 1: READ 2 1: READ 40000000 1: WRITE 1 1: WRITE 3F800000 1: WRITE 2 1: WRITE 40000000 1: WRITE 3 1: WRITE 40400000	✓
6	Skipped	
7	1: READ 1 1: READ 2 1: WRITE 1 2: WRITE 2	✓
8	3: READ 0 1: READ 3FF00000 2: READ 40000000 3: WRITE 0 1: WRITE 3FF00000 2: WRITE 40000000	✓
9	Skipped	
10	Skipped	
11	4: READ 1 4: READ 2 1: READ 3	✓

	2: WRITE 1 2: WRITE 2 3: WRITE 3	
12	2: READ (Address A) 1: WRITE (Address A) 1: WRITE (Address B) 1: WRITE 1 1: WRITE 27 1: WRITE 28	✓
13	4: READ (Address A) 1: READ 1 1: READ 27 1: WRITE (Address A) 1: WRITE (Address B) 2: WRITE 1 2: WRITE 27 1: WRITE 28	✓
14	Skipped	
15	Skipped	
16	Skipped	
17	2: READ 2 2: READ 3 1: WRITE 2 1: WRITE 3 2: WRITE 5	✓
18	1: READ (Address A) 1: WRITE (Address A) 1: WRITE (Address B) 1: WRITE 1 1: WRITE 4	✓
19	1: READ (Address A) 1: WRITE (Address A) 1: WRITE (Address B) 1: WRITE 1 1: WRITE 10	✓
20	1: WRITE 2 1: WRITE 5	✓
21	2: WRITE 5	✓

4.5.6 Local-level Memory Value Profiling on the x86 Architecture

Test cases 14, 15 and 16 were skipped as they produce a lot of output for this profiling method, and therefore do not constitute minimised test cases. Additionally cases 6, 9 and 10 were also skipped as they produce output which is too complex to be considered a minimised test case. As the stack is set up at the beginning and end of the main function, an additional three reads and three writes are recorded during the execution of every test case, which are omitted here for brevity. The format of the expected results is similar to the output CSV file from the profiling code. Each line represents a single memory access. The columns of each line give the frequency, address of value, type of access (read/write), and the value transferred respectively.

Table 4.10: Testing of Local-level Memory Profiling on the x86 Architecture

Test Case	Expected result	Pass
	1: (Address X) READ 1 1: (Address Y) READ 2	
1		✓

	1: (Address X) WRITE 1 1: (Address Y) WRITE 2 1: (Address Z) WRITE 3	
2	2: (Address X) READ 1 2: (Address Y) READ 2 1: (Address X) WRITE 1 1: (Address Y) WRITE 2 2: (Address Z) WRITE 3	✓
3	2: (Address X) READ 1 2: (Address Y) READ 2 1: (Address Z) WRITE 1 1: (Address U) WRITE 1 1: (Address V) WRITE 2 1: (Address V) WRITE FFFFFFFF	✓
4	2: (Address X) READ 0 2: (Address X) READ 1 2: (Address X) READ 2 2: (Address X) READ 3 2: (Address X) READ 4 2: (Address X) READ 5 2: (Address X) READ 6 2: (Address X) READ 7 2: (Address X) READ 8 2: (Address X) READ 9 1: (Address X) READ 10 1: (Address X) WRITE 0 1: (Address X) WRITE 1 1: (Address X) WRITE 2 1: (Address X) WRITE 3 1: (Address X) WRITE 4 1: (Address X) WRITE 5 1: (Address X) WRITE 6 1: (Address X) WRITE 7 1: (Address X) WRITE 8 1: (Address X) WRITE 9 1: (Address X) WRITE A	✓
5	1: (Address X) READ 1 1: (Address U) READ 3F800000 1: (Address Y) READ 2 1: (Address V) READ 40000000 1: (Address X) WRITE 1 1: (Address U) WRITE 3F800000 1: (Address Y) WRITE 2 1: (Address V) WRITE 40000000 1: (Address Z) WRITE 3 1: (Address W) WRITE 40400000	✓
6	Skipped.	
7	1: (Address X) READ 1 1: (Address Y) READ 2 1: (Address X) WRITE 1 1: (Address Y) WRITE 2 1: (Address Z) WRITE 2	✓
	1: (Address R) READ 0 1: (Address S) READ 0 1: (Address T) READ 0	

8		✓

	1: (Address U) READ 3FF00000 1: (Address V) READ 40000000 1: (Address W) READ 40000000 1: (Address S) WRITE 0 1: (Address T) WRITE 0 1: (Address X) WRITE 0 1: (Address U) WRITE 3FF00000 1: (Address V) WRITE 40000000 1: (Address Y) WRITE 40000000 Where (Addr Y)=(Addr X)+4, (Addr V)=(Addr T)+4, (Addr U)=(Addr S)+4, (Addr W)=(Addr R)+4	
9	Skipped.	
10	Skipped.	
11	2: (Address W) READ 1 2: (Address X) READ 1 2: (Address Y) READ 2 2: (Address Z) READ 2 1: (Address U) READ 3 1: (Address W) WRITE 1 1: (Address X) WRITE 1 1: (Address Y) WRITE 2 1: (Address Z) WRITE 2 1: (Address U) WRITE 3 2: (Address V) WRITE 3	✓
12	2: (Address X) READ (Address A) 1: (Address X) WRITE (Address A) 1: (Address Y) WRITE (Address B) 1: (Address Z) WRITE 1 1: (Address V) WRITE 27 1: (Address W) WRITE 28	✓
13	4: (Address X) READ (Address A) 1: (Address Y) READ 1 1: (Address Z) READ 27 1: (Address X) WRITE (Address A) 1: (Address W) WRITE (Address B) 1: (Address Y) WRITE 1 1: (Address U) WRITE 1 1: (Address Z) WRITE 27 1: (Address V) WRITE 27 1: (Address W) WRITE 28	✓
14	Skipped.	
15	Skipped.	
16	Skipped.	
17	2: (Address X) READ 2 2: (Address Y) READ 3 1: (Address X) WRITE 2 1: (Address Y) WRITE 3 2: (Address Z) WRITE 5	✓
18	1: (Address X) READ (Address A) 1: (Address X) WRITE (Address A) 1: (Address Y) WRITE (Address B) 1: (Address Z) WRITE 1 1: (Address Z) WRITE 4	✓
19	1: (Address W) READ (Address A) 1: (Address W) WRITE (Address A)	✓

	1: (Address X) WRITE (Address B) 1: (Address Y) WRITE 1 1: (Address Z) WRITE 10	
20	1: (Address X) WRITE 2 1: (Address Y) WRITE 5	✓
21	2: (Address X) WRITE 5	✓

4.6 Operations

As stated previously, this stage consists of the usage of the developed system. The results gathered using the systems are presented in subsequent chapters of this report.

4.7 Modification of the LLVM Interpreter to call library functions required by the benchmarks

Comments in the source code state that the LLVM interpreter is intended to be a simple, portable interpreter. The interpreter does not emulate a specific machine, but instead represents the program in memory as sequences of instructions and basic blocks linked together in lists and queues. Execution of the program consists of traversing these structures, and executing functions which simulate the effect of each instruction.

Most of the MiBench benchmarks depends on functions present in the ISO/POSIX C Library. At the time of writing, no implementations of the C Library have been ported to the LLVM IR. To do this would require the source code to *libc* to be modified so that it can successfully be compiled to the LLVM IR. Although the source to libc is freely available, this task would be time-consuming and non-trivial. The LLVM interpreter is only able to execute code in LLVM IR format - it is unable to call arbitrary external library functions which are in the machine's native format. However, it does provide wrapper functions for a small subset of the C Library. The majority of these functions are IO functions, such as `printf()` and `scanf`, and a small number of mathematical functions.

4.7.1 Calling external functions through wrapper functions

Analysis of the code reveals that the interpreter takes the following steps when a call is made to an external library:

- In `callFunction()`, if the name of a function is determined to be that of an external function, a call is made to `callExternalFunction()`. (in `lib/ExecutionEngine/Interpreter/Execution.cpp`:1315)
- `callExternalFunction()` first checks to see if a wrapper function for the external library function exists, by calling `lookupFunction()` (in `lib/ExecutionEngine/Interpreter/ExternalFunctions.cpp`:100)
- `lookupFunction()` searches a map containing the names of wrapper functions. Wrapper function names are in the format `lle_X_<name>()`, where name is the name of the external function. As an example, the wrapper for `printf()` is called `lle_X_printf()`. (`ExternalFunctions.cpp`:73-90)
- When control is returned to `callExternalFunction()`, a check is made to determine whether a wrapper function was found. Unfortunately, as of LLVM 2.1 (The current revision), this check is broken (due to a change in the behaviour of `lookupFunction()`), and execution continues regardless, leading to a crash. (`ExternalFunctions.cpp`:101-107)
- `callExternalFunction()` calls the wrapper function, whose return value is that of the library function. (`ExternalFunctions.cpp`:110-111)
- `callExternalFunction()` passes control back to `callFunction()`, returning the returned value from the library function. (`ExternalFunctions`.cpp:112)

4.7.2 Anatomy of a wrapper function

As additional wrapper functions are to be implemented, it is important to consider the purpose and mechanism of a wrapper function. The wrapper function for `pow()` (from the math library) will be used as an example: (from `ExternalFunctions.cpp`:195-201)

```
01  // double pow(double, double)
02  GenericValue lle_X_pow(FunctionType *FT,
                           const vector<GenericValue> &Args) {
03    assert(Args.size() == 2);
04    GenericValue GV;
05    GV.DoubleVal = pow(Args[0].DoubleVal, Args[1].DoubleVal);
06    return GV;
07  }
```

- Line 1: The prototype of the "wrapped" C function.
- Line 2: The function declaration. The wrapper function name conforms to the format stated above. callExternalFunction always passes a `FunctionType*`, which is rarely used. Also, a vector of the arguments is passed. The GenericValue class is used to store the arguments, as a GenericValue can store an arbitrary precision integer, a pointer, a float or a double. This avoids the need for different wrapper functions to take different arguments. A wrapper function cannot infer from a GenericValue what the type actually passed to it is - it is responsible for interpreting the value of the GenericValue as the correct type before passing it to the native function.
- Line 3: Checks that the correct number of arguments have been passed. In this case two are required, as per the prototype of `pow()`. An incorrect number of arguments implies that there is an error in the bitcode being executed, and causes termination of the interpreter.
- Line 4: Declares a new GenericValue to store the result of the native library function.
- Line 5: Calls the native library function. The arguments are passed as the correct type. The return value is stored into the variable `GV`. The type of the return value must also be specified correctly. The correct type of an argument or return value is always the same as the type used in the prototype.
- Line 6: Returns the result of the native library function. Although it is clear from this example that the GenericValue stores a value of type double, the bitcode must know the correct type of the returned value in order to store the result correctly.

Almost all wrapper functions follow the same guidelines:

1. The arguments passed to the function are converted to the correct type.
2. The native function is called with these arguments.
3. The return value of the native function is recorded as the correct type.
4. This return value is passed back to the caller.

There are some exceptions to these guidelines, but describing the purpose and implementation of every wrapper function is outside the scope of the project.

4.7.3 Choosing the correct functions to implement

To determine the correct functions to implement, each of the benchmarks must be examined to see what library functions it uses. This can be done by first disassembling the object file, and then searching through the disassembly for any `declare` statements. For example, to examine the $automotive-susan-c$ benchmark, the following commands would be used:

```
llvm-dis automotive-susan-c.llvm.bc
grep declare automotive-susan-c.llvm.ll
```

Which gives the output:

```
declare void @exit(i32)
declare i32 @_IO_getc(%struct.FILE*)
declare i8* @fgets(i8*, i32, %struct.FILE*)
declare i32 @fprintf(%struct.FILE*, i8*, ...)
declare %struct.FILE* @fopen(i8*, i8*)
declare i32 @fgetc(%struct.FILE*)
declare i32 @fread(i8*, i32, i32, %struct.FILE*)
declare i32 @fclose(%struct.FILE*)
declare i32 @fwrite(i8*, i32, i32, i8*)
declare double @exp(double)
declare void @llvm.memset.i32(i8*, i8, i32, i32)
declare void @llvm.memcpy.i32(i8*, i8*, i32, i32)
declare i32 @printf(i8*, ...)
declare double @llvm.sqrt.f64(double)
declare double @atof(i8*)
declare i32 @atoi(i8*)
declare i32 @putchar(i32)
```

Each of these functions is used at least once by the benchmark. Some of these are already implemented in `ExternalFunctions.cpp`. However, others must be added. The prototypes of all functions which had to have wrapper functions added in order to allow execution of 17 benchmarks is:

```
double cos(double x);
double sin(double x);
double acos(double x);
double fabs(double x);
int strncmp(const char *S1, const char *S2);
void bcopy(const void *S1, const void *S2, size_t n);
uint32_t htonl(uint32_t hostlong);
int fscanf(FILE *stream, const char *format, ...);
int memcmp(const void *S1, const void *S2, size_t n);
int fseek(FILE *stream, long int offset, int whence);
long int ftell(FILE *stream);
int toupper(int c);
int tolower(int c);
size_t read(int fildes, void *buf, size_t nbyte)
size_t write(int fildes, void *buf, size_t nbyte)
const unsigned short * * __ctype_b_loc  (void);
char *getenv(const char *name);
double atof(const char *str);
long int __strtol_internal(const char *str, char **endptr, int base, int group);
```

The implementation of these functions should allow the following benchmarks to execute correctly:

```
automotive-basicmath
automotive-bitcount
automotive-susan-c
automotive-susan-e
automotive-susan-s
consumer-jpeg-c
consumer-jpeg-d
network-dijkstra
office-stringsearch
security-blowfish-d
security-blowfish-e
```

```
security-rijndael-d
security-rijndael-e
security-sha
telecomm-adpcm-c
telecomm-adpcm-d
telecomm-CRC32
```

It was originally intended that all of these benchmarks will be profiled in the experiments, apart from *automotive-basicmath*, as there are no additional datasets from MiDatasets for this benchmark. These benchmarks were chosen so that there is at least one benchmark from each of the five groups of benchmarks in MiBench, and because these benchmarks used the fewest number of external functions. Other benchmarks could be made to work with the addition of many other external functions, but due to time constraints this has not been carried out.

However, it was subsequently found that the benchmarks, *automotive-bitcount*, *automotive-susan-s*, *security-blowfish-d* and *-e*, and *security-rijndael-e* all fail when executed in the LLVM interpreter for other reasons. It is possible that there could be other parts of the interpreter which lack the required functionality to properly execute these benchmarks. Further investigation into these reasons was decided to be outside of the scope of the project, due to limited time.

4.8 Post-Processing of Instruction Level Value Profile Data

4.8.1 Sorting and Mathematical Operations

The output Value Profile Data from the Value Profiling tools will be large, unsorted lists of all the Value Profile Data gathered throughout the execution of the benchmark. Several (unsorted) example fields from an Instruction-level Value Profile:

```
5       ADC     0       0
2       ADC     4294967293      4294967295
291949  ADD     0       0
277     ADD     0       1
751     ADD     0       2
```

The columns of this data represent: `Frequency`, `Instruction Opcode`, `Operand 1`, and `Operand 2`. This data must be sorted in descending order of frequency, in order for any meaningful conclusions to be drawn from the data. Once this is done, the first line of the sorted file will be the most frequent computation, the next line will be the second most frequent computation, etc. Sorting this data would yield:

```
291949  ADD     0       0
751     ADD     0       2
277     ADD     0       1
5       ADC     0       0
2       ADC     4294967293      4294967295
```

The total number of instructions executed is also output from the Value Profiling tools in a separate file. As it is known what the total number of instruction executions is, the frequency of each computation can be calculated as a percentage of all instruction executions. In the Value Profile data above, supposing that 1000000 instructions were executed in total, the most frequent computation (ADD 0 0) would represent 29.19% of all instruction executions. This would be calculated as follows:

$$\begin{aligned} Percentage &= \frac{freq_1}{total} \times 100 \\ &= \frac{291949}{1000000} \times 100 \\ &= 29.19\% \end{aligned}$$

Where $freq_1$ is the frequency of the most frequent computation, and *total* is the total number of instructions executed.

It will be useful to consider what percentage of all instruction executions are made up by a particular set of computations - for example, the two most frequent computations, or the four most frequent computations. In the example Value Profile data, the percentage of all instruction executions represented by the two most frequent computations would be calculated as follows:

$$\begin{aligned} Percentage &= \frac{freq_1 + freq_2}{total} \times 100 \\ &= \frac{291949 + 751}{1000000} \times 100 \\ &= 29.27\% \end{aligned}$$

Where $freq_1$ is the frequency of the most frequent computation, $freq_2$ is the frequency of the second most frequent computation, and *total* is the total number of instructions executed. A general formula to determine the percentage of all instructions represented by the top N most frequent computations is:

$$Percentage = \sum_{i=1}^{N} \frac{freq_i}{total} \times 100 \tag{4.1}$$

Where $freq_i$ is the frequency of the i^{th} most frequent computation, and again *total* is the total number of instruction executions. This method will also be applied to Memory Access level Value Profile Data. A small example of Memory Access Value Profile Data:

```
450345  Store   INT     0       0
65      Store   PTR     0       0
1100262 Load    INT     0       0
2616    Load    PTR     0       0
38571   Store   INT     0       1
```

Sorting this data would yield:

```
1100262 Load    INT     0       0
450345  Store   INT     0       0
38571   Store   INT     0       1
2616    Load    PTR     0       0
65      Store   PTR     0       0
```

As this data is now sorted, the most frequent value can easily be determined, as can the second most frequent value, and the third most frequent value, and so on. Equation 4.1 can be used to calculate the percentage of all memory accesses that the top N most frequent values represent. For example, to calculate the percentage of all memory accesses represented by the top 4 most frequent values (Assuming that there were 3500000 memory accesses in total:

$$\begin{aligned} Percentage &= \sum_{i=1}^{N} \frac{freq_i}{total} \times 100 \\ &= \frac{freq_1 + freq_2 + freq_3 + freq_4}{3500000} \times 100 \\ &= \frac{1100262 + 450345 + 38571 + 2616}{3500000} \times 100 \\ &= 45.48\% \end{aligned}$$

4.8.2 Presenting the Output

As most works considered in the literature review used graphs to present summaries of Value Profile Data, summaries of Value Profile Data will also be presented as graphs in this report. One paper (Yi & Lilja, 2001) did not use graphs but instead presented summaries as tables of numeric values. This made results difficult to interpret so the same format will not be used in this report.

For each benchmark executed, a histogram will be produced which shows the percentage of all instruction executions represented by the most frequent computations for each dataset. The percentage of all instructions represented by the top 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 and 2048 most frequent computations will be calculated and for each dataset and represented as a histogram on a graph. Each dataset will be represented using a single column, and an average across all datasets produced.

The same method will be used to present the Value Profile data of Memory Accesses: the percentage of all memory accesses represented by the top 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 and 2048 most frequent values will be calculated for each dataset of each benchmark and presented on a single graph for each benchmark. Again an average will be taken across all datasets for each benchmark.

4.9 Conclusion to Implementation

After the implementation and testing of the Value Profiling tools, Value Profile data for the 11 specified benchmarks was gathered and processed. The results produced are presented and analysed in the following chapter.

Chapter 5

Results & Analysis

This chapter presents summaries of the amount of Value Reuse measured in the following areas:

- Global-level Instruction Value Profiling
- Global-level Memory Value Profiling
- Local-level Memory Value Profiling

It was not possible to gather Local-level Instruction Value Profile data as far too much storage space was required. The output data from a single benchmark exceeded all of the available disk space. Because of this, it has so far not been possible to support or disprove hypothesis 2. However, a method is be presented in Chapter 6 concerning this hypothesis.

It is important to note when considering Instruction-level Value Profiling that the results show the percentage of all executions accounted for by each set of instructions, *including those which were not profiled.* Summaries are subsequently presented which show only the percentage of all profiled instructions accounted for by the sets of frequent instructions.

It is also important to note that generally the datasets are not sorted into any particular order (of size, number of colours in image, mean amplitude of sound sample etc.) for any of the benchmarks. The exception is *network-dijkstra*, which exhibits a trend in its datasets which will be discussed later.

5.1 LLVM Value Profile Data

5.1.1 Global-level Instruction Value Profiling

Automotive-susan-c. It can be observed from the results that:

- The most frequently executed instruction/operands set (marked Top-1 on the graph) accounts for up to 9.07% of all instruction executions, and 4.99% on average. (See CD for top 32 values output for each benchmark). With the exception of sets 13 and 16, this instruction is always a GEP instruction, which computes an offset of 0 from the base address. In other words, the instruction does nothing - its output is exactly the same as one of its inputs, the base address input. Therefore, the time spent performing this computation is wasted. On average, 4.99% of executions could be skipped, if it were recognised that this computation is redundant.
- The majority of the top 32 most frequently executed instructions are GEP instructions. These all compute small offsets, usually in the range [-12,12]. It is thought that these operations can be directly attributed to a particular portion of automotive-susan-c, the *brightness lookup table*. This is accessed using a pointer which points to the middle of the table. This pointer is offset by the number of levels of greyscale difference (presumably between one pixel and the next). As the number of levels of greyscale difference between one pixel and the next is usually either zero or very small, this is an explanation for these frequently occurring computations. The lookup table is initialised in a function `setup_brightness_lut()` at line 468 of the source.

- As a consequence of the previous point, the same addresses are frequently computed to read from. Therefore, it is expected that there will be a correlation between the amount of value reuse in instruction executions and memory accesses for this benchmark.
- Across all datasets, around 50-60% of all instruction executions were of instruction opcodes which are profiled. This is an indication that the correct choices have been made as to which instructions to profile. The LLVM IR has 49 instruction opcodes, of which only 16 have been profiled. This shows that the correct choice has been made as profiling extra instructions would only be likely to provide a small amount of extra coverage of all dynamic executions.
- It can be seen that the set of the top 64 most frequently executed instructions account for 20% of all instructions executions. **This supports Hypothesis 1**, that value reuse is prevalent throughout the execution of most programs.
- Across all the datasets of this benchmark, there are many computations which frequently occur in the top 32 most frequently executed instruction/operand sets. This suggests that the frequently occurring instruction/operand sets are to some extent independent of the input set. This adds weight to the hypothesis put forward in (Yi *et al.*, 2002).
- There is some variation in the amount of Value Reuse across benchmarks. For example, set 19 has a low level of Value Reuse, whereas set 11 has a much higher incidence of Value Reuse.

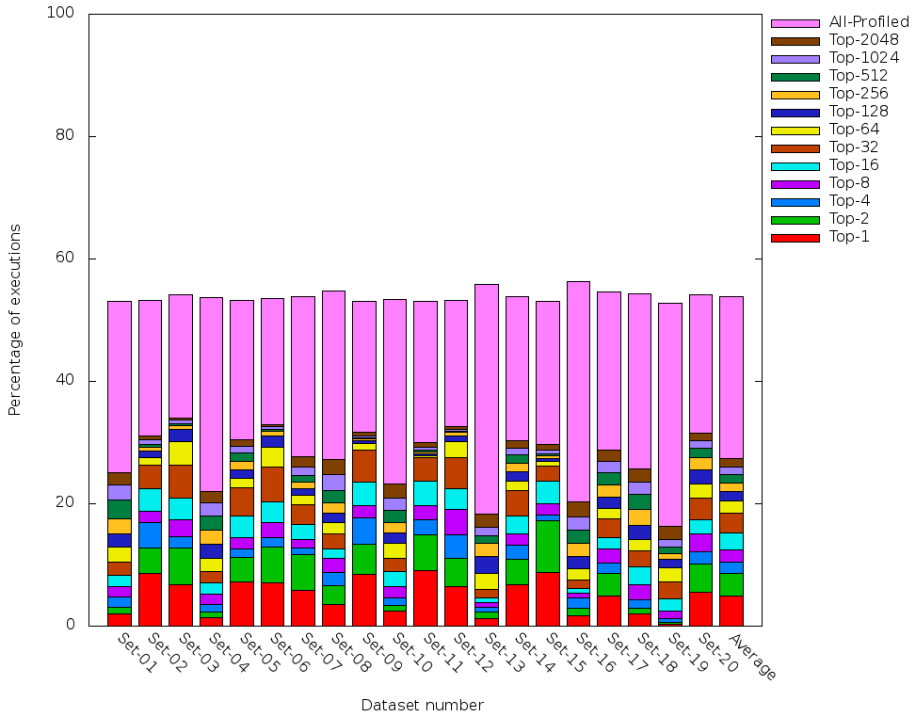

Figure 5.1: Automotive-susan-c. Percentage of all profiled instructions accounted for by the top N frequently occurring instructions at global level on LLVM.

Automotive-susan-e. It can be observed from the results that:

- The amount of Value Reuse (which can be inferred from the height of the bars of the graph) for a particular dataset for this benchmark is similar (though slightly lower) to the amount of value reuse for the same dataset input to the *automotive-susan-c* benchmark. It is quite likely that the two benchmarks execute portions of the same code, as the two benchmarks are both run from a single source file. The `-c` version of the benchmark implements a corner detection algorithm, whilst the `-e` version of the benchmark implements an edge detection algorithm.
- The top most frequently executed instruction/operands set accounts for up to 8.08%, and on average accounts for 3.85% of all instruction executions. The dataset for which the top most

frequently executed instruction/operands set is greatest is set 11, which was also the set with the greatest value reuse for the `-c` version of the benchmark.

- The top 64 most frequently executed instruction/operand sets only account for over 17% of all instruction executions on average. The 512 top most frequently executed instructions exceed 20% of all instructions. This must mean that 448 instruction/operand sets only account for around 3% of all instruction executions, whereas the top 64 instruction/operand sets account for 17% of all instruction executions. This shows that a significant fraction of all instructions executions are again due to a small number of unique computations. **Again this is in support of Hypothesis 1**.
- Again, between 50 and 60% of all instruction executions were profiled. This is further suggestion that the correct choice of instructions to profile has been made.
- As with *automotive-susan-c*, the most frequently occurring instruction/operand set is usually a redundant GEP operation. The benefit of eliminating this operation would be slightly less than with *automotive-susan-c*, as on average only 3.85% of computations would be eliminated.
- Again as with *automotive-susan-c*, many of the top instruction/operand sets are GEP operations computing offsets in the brightness lookup table.
- The same instruction/operand sets frequently occur in the top most frequently executed sets across all datasets for this benchmark.
- Again there is some variance in the amount of Value Reuse across datasets.

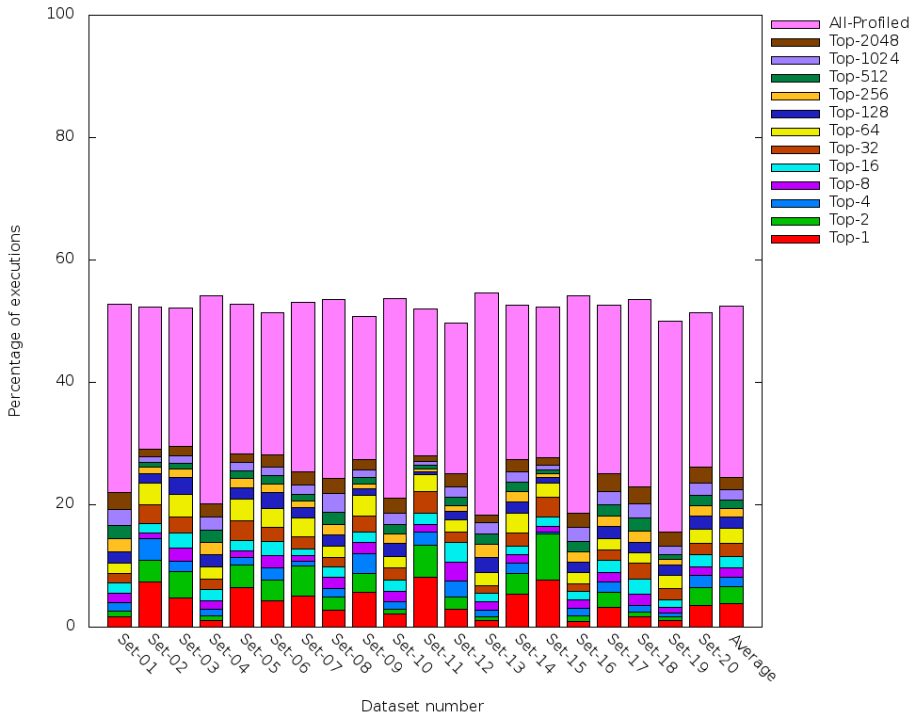

Figure 5.2: Automotive-susan-e. Percentage of all profiled instructions accounted for by the top N frequently occurring instructions at global level on LLVM.

Consumer-jpeg-c. It can be observed from the results that:

- A smaller fraction of all instruction executions were of instruction opcodes which were profiled. In this case, approximately 40% of all instruction executions were profiled.
- The amount of Value Reuse is less variant across data sets.

- There is not a great amount of value reuse of instructions in this benchmark. The top 64 most frequently occurring instruction/operand sets account for less than 10% of all instruction executions. This is much lower than the previous two benchmarks. However, the top 512 most frequently occurring instruction/operand sets account for almost 20% of all instruction executions, which is close to the amount for *automotive-susan-e*. This shows that across the top 512 most frequently occurring instruction/operand sets, the distribution of the number of occurrences is much more even than for Automotive-susan-e.
- **This information is still in support of Hypothesis 1** - 512 unique instruction/operand sets is still a very small amount of sets in the space of all possible instruction/operands sets. This small amount of unique computations accounts for almost 20% of all instruction executions, which leads to the conclusion that value reuse is prevalent in this benchmark.
- The two most frequently occurring instruction/operand sets for most datasets are redundant computations. One of these is subtracting 0 from 0, and the other is adding 0 to 0. As both of these instructions do not change the input operands in any way, they could be eliminated to reduce the number of executions, if the processor were able to recognise that they were redundant.
- Again there is a common set of instruction/operands sets which occurs in the top 32 most frequently occurring instruction/operands sets across all datasets.
- Of the instructions that were profiled, the difference between the 2048 most frequently occurring instruction/operands sets and all other sets that were profiled is less than that for the *automotive-susan* benchmarks.

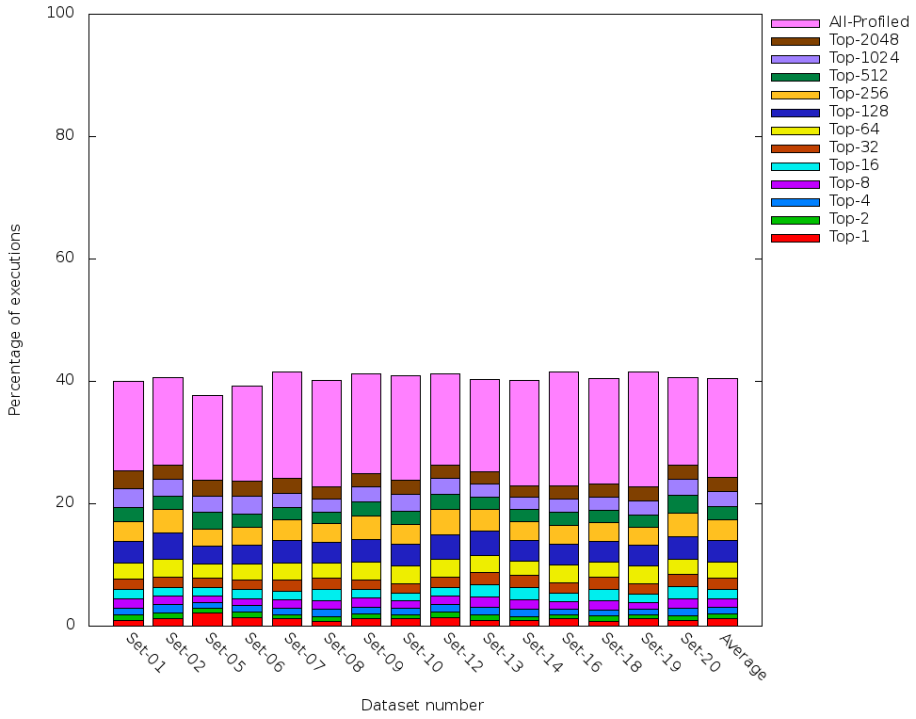

Figure 5.3: Consumer-jpeg-c. Percentage of all profiled instructions accounted for by the top N frequently occurring instructions at global level on LLVM.

Consumer-jpeg-d. It can be observed from the results that:

- There is a great deal of variation in the amount of Value Reuse across datasets. This is very different to the amount of variation in the amount of Value Reuse across datasets for *consumer-jpeg-c*.

- As there is a large difference between the results of these two benchmarks, it is likely that the algorithm for decompressing a JPEG image is very different to the algorithm for compressing. This is in contrast to the (small) differences between *automotive-susan-c* and *automotive-susan-e*.
- With this benchmark, 50-60% of all instruction executions have been profiled, supporting the idea that the correct instructions to profile have been chosen.
- On average, the top 64 most frequently occurring instruction/operand sets represent less than 10% of all instruction executions. Even the top 1024 most frequently occurring instruction/operands sets do not quite cover 20% of all instruction executions. The amount of Value Reuse of instruction executions is therefore considered lower for this benchmark than for other benchmarks seen so far.
- **This provides some support for Hypothesis 1**. There is a degree of Value Reuse, but no small set of individual values is responsible for a large fraction of executions.
- As with other benchmarks, the two most frequently occurring instruction/operands sets are redundant. In this case one set consists of adding 0 and 0, and the other of ORing 0 and 0. However, both of these sets generally only represent a very small fraction of instructions (The heights of the Top-1 and Top-2 bars are very small) so eliminating these executions would not provide a great increase in the speed of execution.
- There are certain instruction/operands sets which occur in the top 32 most frequently occurring instruction/operands sets across all datasets. Some of these are GEP operations, which repeatedly compute the same address in memory to access. This increases the likelihood that the prediction that there is a correlation between the amount of value reuse in instruction executions and the amount of value reuse in memory access is true.
- Additionally this supports (Yi *et al.*, 2002) hypothesis that the frequently executed instruction/operands sets are partially independent of the input set (i.e. that they are a characteristic of the benchmark.

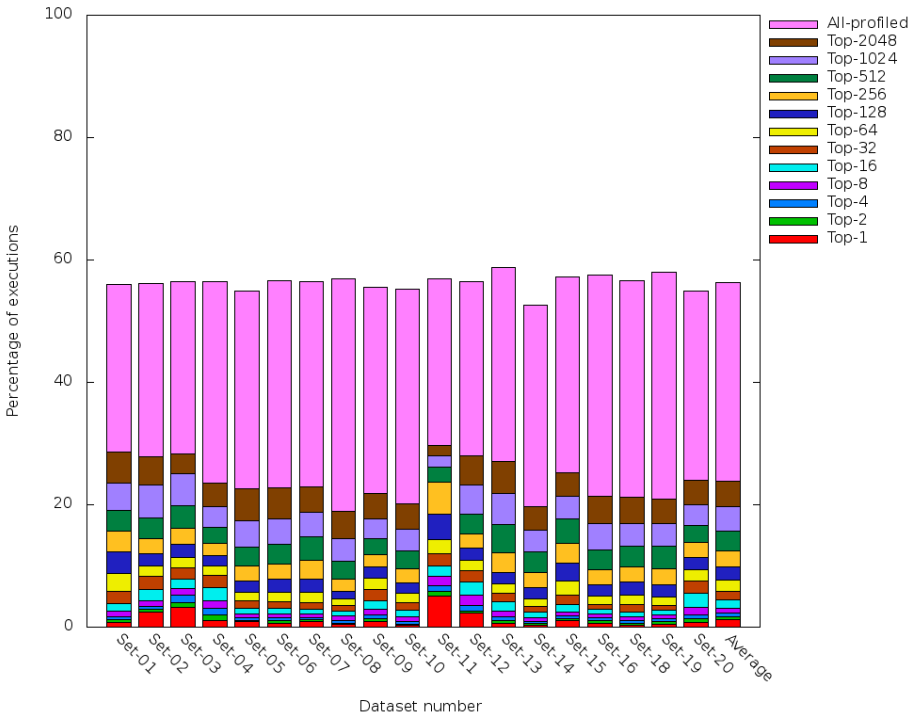

Figure 5.4: Consumer-jpeg-d. Percentage of all profiled instructions accounted for by the top N frequently occurring instructions at global level on LLVM.

Network-dijkstra. It can be observed from the results that:

- There is a trend in the amount of Value Reuse across the datasets of this benchmark. The datasets 1 to 19 are in ascending order of size. As the size of the dataset increases, the amount of value reuse decreases.
- It is thought that the amount of Value Reuse is high when the size of the network is small as there are only a small number of distinct values which represent path weights. As larger networks are input, computing the total weight of the path between two nodes will less frequently involve the same values as there will be more distinct values of path weights.
- It may be possible to use a regression test to determine the amount of Value Reuse for a given dataset before its path weights are computed. This may have an application if there were some cost associated with using a Value Reuse optimisation. The test could be performed before the Dijkstra algorithm were executed to determine if the cost of using the Value Reuse optimisation would be greater than any benefit derived from it.
- On average, the top 2048 most frequently occurring instruction/operands sets account for just under 20% of all instruction executions. **This does lend some support to Hypothesis 1.**
- Across all datasets, there are some instruction/operands sets which frequently occur in the sets of top 32 instruction/operands sets. These are mainly additions of small integers. Again this lends support to (Yi *et al.*, 2002) hypothesis that the frequently occurring instruction/operands sets are independent of the input set.
- The instruction/operands sets which are seen in the top 32 are generally not redundant computations for this benchmark, across all datasets.
- Only a small fraction of all instruction executions were profiled in this case. Just under 30% of all instruction executions were profiled.

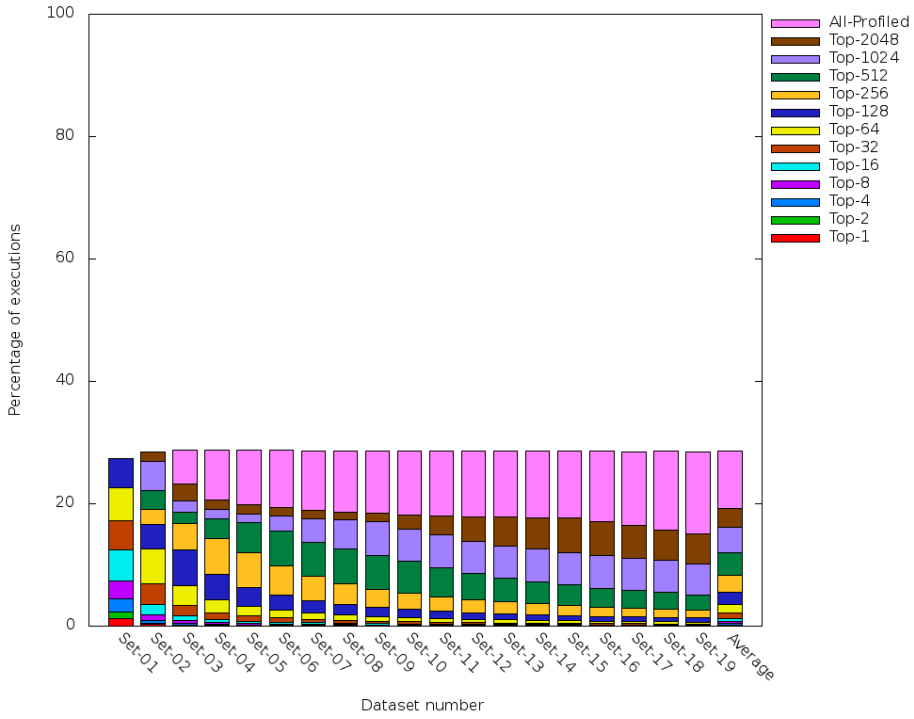

Figure 5.5: Network-dijkstra. Percentage of all profiled instructions accounted for by the top N frequently occurring instructions at global level on LLVM.

Office-stringsearch. It can be observed from the results that:

- There is hardly any variation in the amount of Value Reuse across benchmarks.

- A smaller fraction (just over 30%) of all instruction executions were of profiled instructions.
- Almost all of these profiled instructions consist of 512 instruction/operands sets. This conclusion has been reached as the height of the "Top-512" bar is almost as tall as the "All-Profiled" bar. Therefore, only a small number of profiled instruction executions were not due to the sets in the top 512 most frequently executed instruction/operands sets.
- Across all benchmarks, most of the top 32 most frequently executed instruction/operands sets were GEP instructions. These GEP instructions all compute addresses to access the contents of arrays. Other sets which appear in the top 32 most frequently executed sets are add instructions of small integers. Many of these add instructions appear with the same operands across multiple datasets.
- Because the top 32 most frequently occurring instruction/operands sets are very similar across all datasets, it is suggested that the Value Profile for this benchmark is independent of the input set. Again this supports (Yi *et al.*, 2002) hypothesis.
- **These results support Hypothesis 1**. There is a prevalence of Value Reuse in Instruction executions in this benchmark, for all tested input sets.
- This benchmark only has a low total fraction of all instruction executions made up from profiled instructions as it is likely that there is a large number of comparison operations in this benchmark, as its function is to search for strings in other strings of text. Comparison operations were not profiled.
- The top most frequently executed instruction/operands set is always a redundant GEP operation. Eliminating this operation would be of little benefit in terms of execution time, as executions of this instruction only account for 0.93% of all instruction executions.

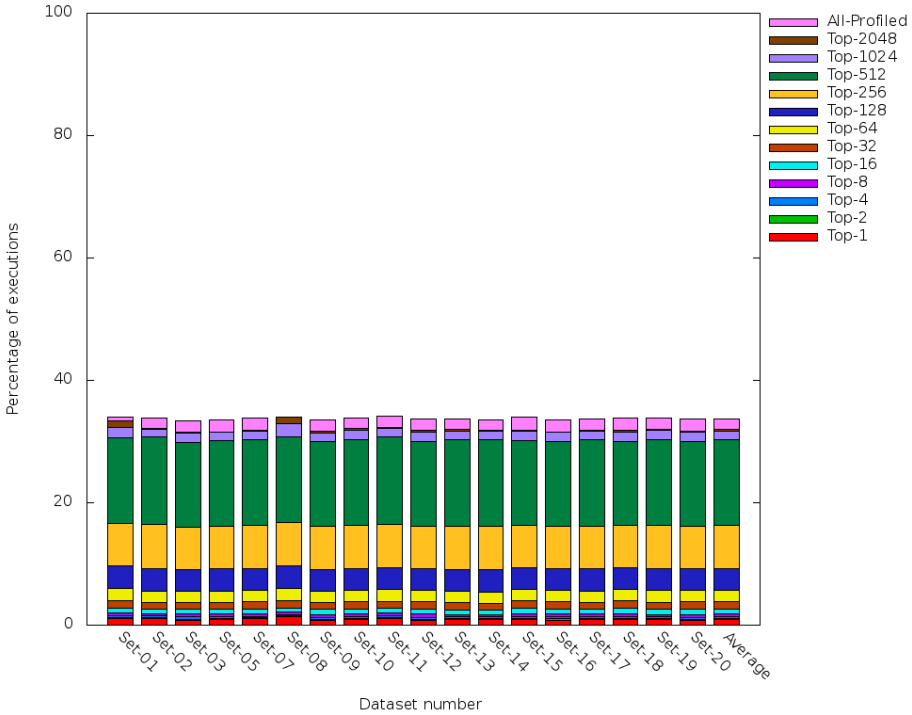

Figure 5.6: Office-stringsearch. Percentage of all profiled instructions accounted for by the top N frequently occurring instructions at global level on LLVM.

Security-rijndael-d. It can be observed from the results that:

- A large fraction of all instruction executions were of profiled instructions. Just under 70% of all instruction executions were profiled.

- Within these profiled instructions, there is a small amount of Value Reuse. Even the top 512 most frequently occurring instruction/operands sets only account for around 15% of all instruction executions.
- The top 32 most frequently occurring instruction/operands sets only account for 2.46% of all instruction executions.
- **This data does provide support for Hypothesis 1**, as there is some value reuse across all datasets.
- The top 32 most frequently occurring instruction/operands sets across all datasets mostly contain the same sets. This again supports (Yi *et al.*, 2002) hypothesis that the most frequent operations are independent of the input set. However, these sets represent an insignificant fraction of all instruction executions.
- Examination of the profile output shows that most dynamic instruction executions are of instruction/operands sets which are only ever executed a single time. These are all XOR operations of two large integers. It is thought that this is a characteristic of this particular benchmark.
- As this benchmark implements an encryption algorithm, its output is likely to have a high degree of entropy. Because of this, the values which it will be working with internally will also have a high degree of entropy, and therefore there are generally few operations which are ever repeatedly executed.
- There is very little variation in the amount of Value Reuse across all datasets for this benchmark.
- The most frequently executed operation is again a redundant GEP operation. Eliminating executions of this operation for the purpose of decreasing execution time would be futile, as this operation only represents 0.15% of all instruction executions.

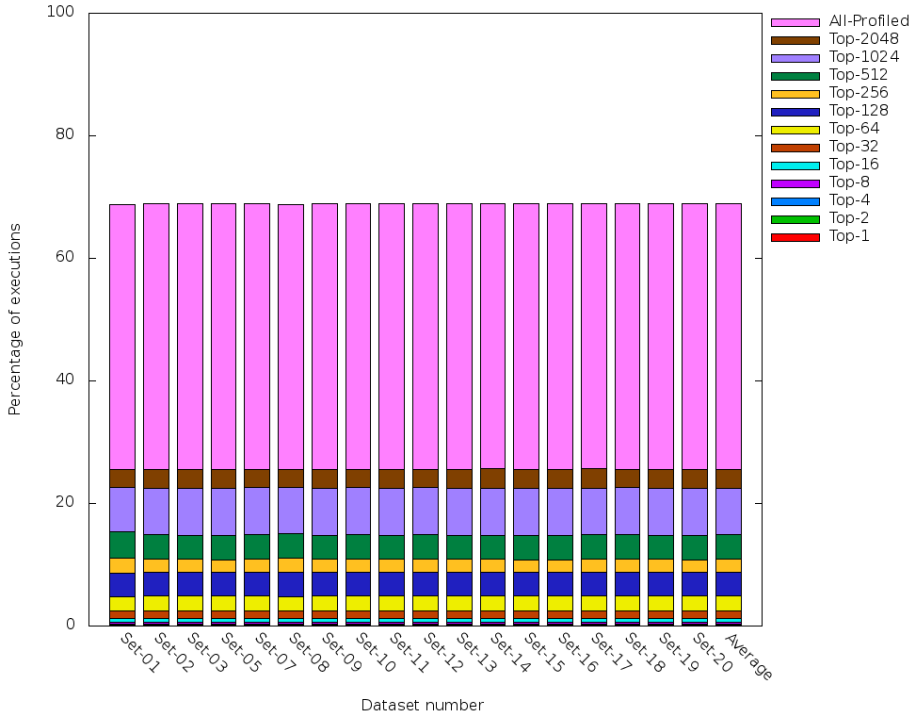

Figure 5.7: Security-rijndael-d. Percentage of all profiled instructions accounted for by the top N frequently occurring instructions at global level on LLVM.

Security-sha. It can be observed from the results that:

- As with *security-rijndael-d*, a large fraction of all instruction executions were of profiled instructions. Within these profiled instructions, there is a similar amount of Value Reuse.
- Again, the top 512 most frequently occurring instruction/operands sets represent just over 15% of all instruction executions on average.
- **These results lend some support to Hypothesis 1**, as there is some Value Reuse present.
- Unusually with this benchmark, no GEP operations occur in the top 32 most frequently executed instruction/operands sets. The top 32 instruction/operands sets consist solely of the addition of small integers. Again many of these are common across all the datasets.
- Occasionally redundant operations are present in these top 32 instruction/operands sets, such as the addition of 0. As with certain other benchmarks there would be little benefit in eliminating these operations as they constitute such a small fraction of all instruction executions.
- The remaining instruction executions are almost all XOR operations of two large numbers, which are only executed a single time throughout the whole execution of the benchmark.
- This benchmark is also an encryption algorithm, like *security-rijndael-d*. This suggests that there may be some common elements in the operation of the two schemes.
- It is possible that programs performing the same function using different algorithms may generally have similar Value Profiles. However, a conclusion cannot be drawn on the basis of these two benchmarks, but further investigation would be necessary.

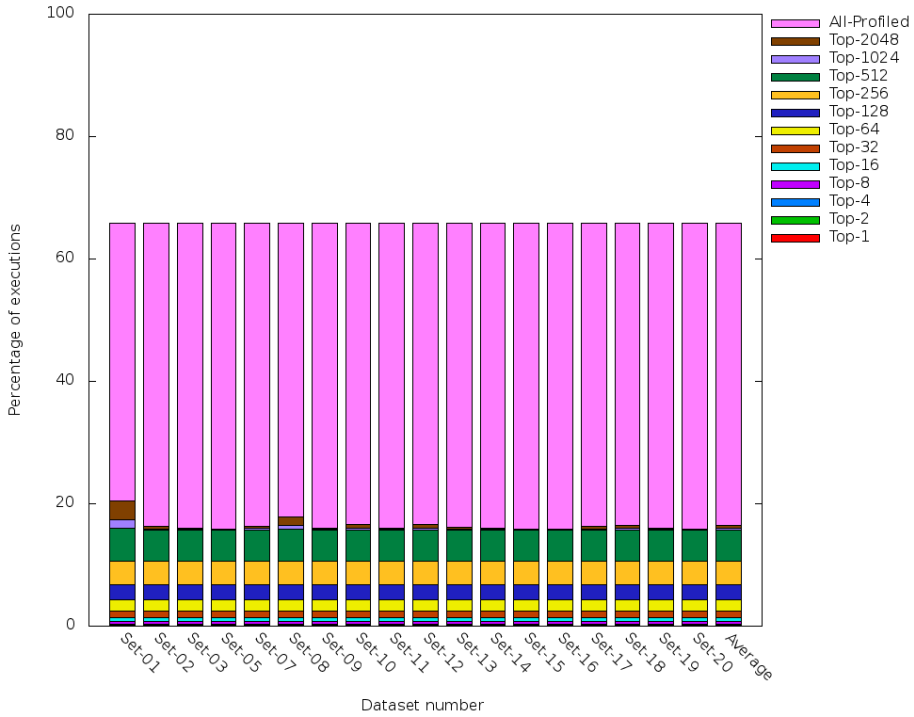

Figure 5.8: Security-sha. Percentage of all profiled instructions accounted for by the top N frequently occurring instructions at global level on LLVM.

Telecom-adpcm-c. It can be observed from the results that:

- Just under 40% of all instruction executions were of profiled instructions. The exact fraction of profiled instructions is very consistent across all datasets.
- However, the amount of Value Reuse differs slightly across each dataset. For example, Set 13 has the greatest number of instruction executions due to a single instruction/operands set, yet Set 1 has a greater number of instruction executions due to the top 512 most frequently occurring instruction/operands sets than Set 13 does.

- On average, the top 32 most frequently occurring instruction/operands sets account for just under 10% of all instruction executions.
- There is Value Reuse present throughout all datasets for this benchmark. **This supports Hypothesis 1**.
- Across all datasets, the top 32 most frequently occurring instruction/operands sets consist of arithmetic or bitwise operations. Many of these operations are found throughout the top 32 most frequently occurring instruction/operands sets for all benchmarks.
- There are some GEP operations present in the top 32 most frequently occurring instruction/operands sets. Typically there are between 2 and 8 GEP operations in the top 32 most frequently occurring instruction/operands sets for all datasets. Usually none of these operations are redundant.

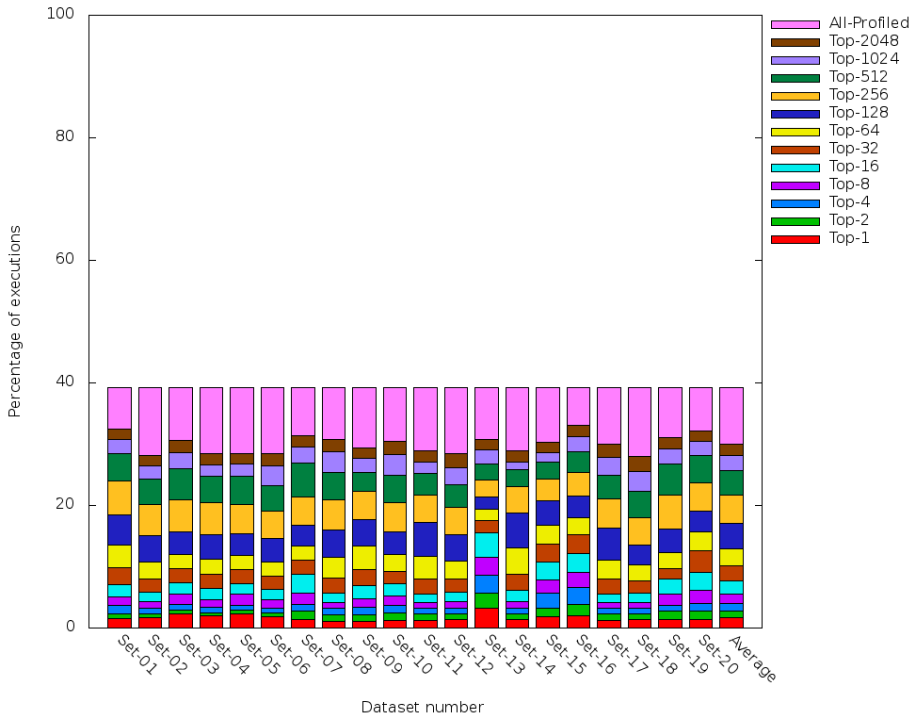

Figure 5.9: Telecom-adpcm-c. Percentage of all profiled instructions accounted for by the top N frequently occurring instructions at global level on LLVM.

Telecom-adpcm-d. It can be observed from the results that:

- Like *telecom-adpcm-c*, approximately 40% of all instruction executions for every dataset are profiled. There is slightly more variation than in *telecom-adpcm-c* across datasets.
- On average, the top 32 most frequently occurring instruction/operands sets account for approximately 10% of all instruction executions.
- **Further support is provided to Hypothesis 1 by this data**, as it can be seen that there is Value Reuse in Instruction Executions within the execution of the benchmark for all datasets.
- The top 32 most frequently occurring instruction/operands sets across all datasets mainly consist of bitwise operations, such as AND and XOR operations, operating on small integers. These operations are different to the ones seen in *telecom-adpcm-c*.
- The operations in the top 32 most frequently occurring instruction/operands sets again generally consist of a small number of GEP operations, which are not redundant.

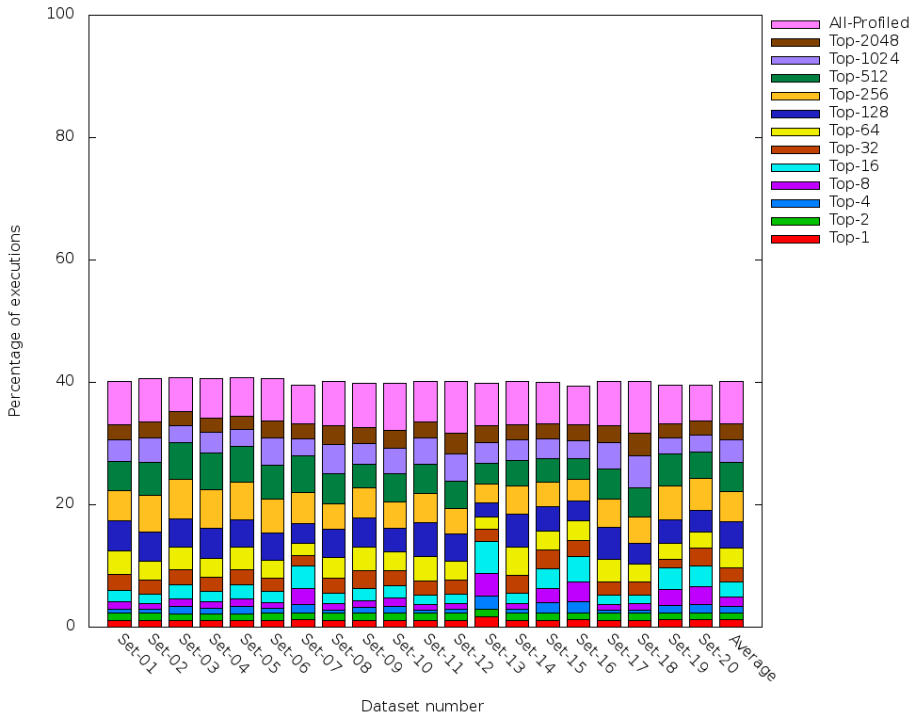

Figure 5.10: Telecom-adpcm-d. Percentage of all profiled instructions accounted for by the top N frequently occurring instructions at global level on LLVM.

Telecom-crc32. It can be observed from the results that:

- Over 50% of all instruction executions were of profiled instructions.
- However, this benchmark shows by far the lowest amount of Value Reuse. Even the top 2048 most frequently occurring instruction/operands sets account for less than 10% of all instruction executions on average.
- However, this still shows that there is some Value Reuse in Instruction Executions, **and does support Hypothesis 1**.
- All of the top 32 most frequently occurring instruction/operands sets for all datasets consist entirely of GEP operations.
- Examining the entire profile data reveals that almost all operations are XOR operations which are only executed a single time throughout the execution of the program.
- This is a characteristic of the operation of the CRC32 algorithm. The operation of the CRC32 algorithm consists of repeated XOR operations of a dataset until there is only a remainder left which cannot be XORed any more. As a result of this operation, values which are inputs to the XOR operation have a high degree of entropy, which leads to a very small amount of Value Reuse.

Comparison Across All Benchmarks

- The benchmarks, *automotive-susan-c*, *automotive-susan-e*, *telecom-adpcm-c*, and *telecom-adpcm-d* had greater levels of Value Reuse from small sets of instruction/operands sets than other benchmarks. On average, for each of these benchmarks, the top 32 most frequently occurring instruction/operands sets accounted for between 10 and 20% of all instruction executions. *telecom-adpcm-c* and *-d* also had the highest levels of Value Reuse with larger sets - on average over 30% of all instruction executions were represented by the top 2048 most frequently occurring instruction/operands sets for both of these benchmarks.

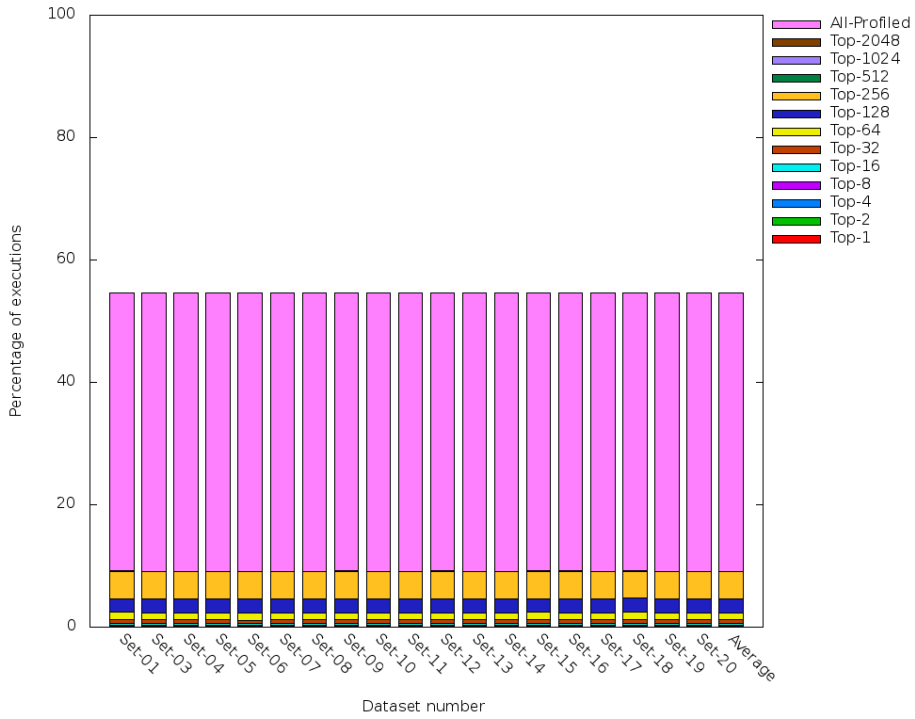

Figure 5.11: Telecom-crc32. Percentage of all profiled instructions accounted for by the top N frequently occurring instructions at global level on LLVM.

- *Consumer-jpeg-c* and *-d* had smaller levels of Value Reuse with small sets of instruction/operands sets. For these benchmarks, on average the top 32 most frequently occurring instruction/operands sets account for between 5 and 10% of all instruction executions. However, the top 2048 instruction/operands sets represents approximately 25% of all instruction executions on average, which is similar to the *automotive-susan* benchmarks.
- A third group of benchmarks consists of *network-dijkstra*, *office-stringsearch*, *security-rijndael-c*, *security-sha*, and *telecom-crc32*. These benchmarks all have very little Value Reuse due to a small number of sets of instruction/operands sets. On average, the top 32 most frequently occurring instruction/operands sets account for less than 5% of all instruction executions for all these benchmarks. Additionally, *telecom-crc32* shows the lowest level of Value Reuse in general - even when considering the amount of instruction executions due to the top 2048 most frequently occurring instruction/operands sets, less than 10% of all instruction executions are represented.
- On average across all benchmarks, it can be seen that there is a definite occurrence of Value Reuse. Even just the single most frequently executed instruction/operands set accounts for 1.4% of all instruction executions. Although this is not a significant fraction of execution time, as the lifetime of a single benchmark is typically tens or hundreds of millions of instructions, this shows that the exact same instruction with the exact same operands is executed hundreds of thousands or even millions of times during the execution of a typical benchmark.
- When extending this to include more than just a single instruction and its inputs, the number of repeated computations becomes even higher. As can be seen from the graph, considering 2048 unique instructions including the inputs shows that over 25% of computations are repeated. **This is strong support for Hypothesis 1.**

Considering only profiled instructions

Although the percentage of all instruction executions accounted for by the few most frequent computations is relatively low, it can be seen that the few most frequent computations make up a more significant

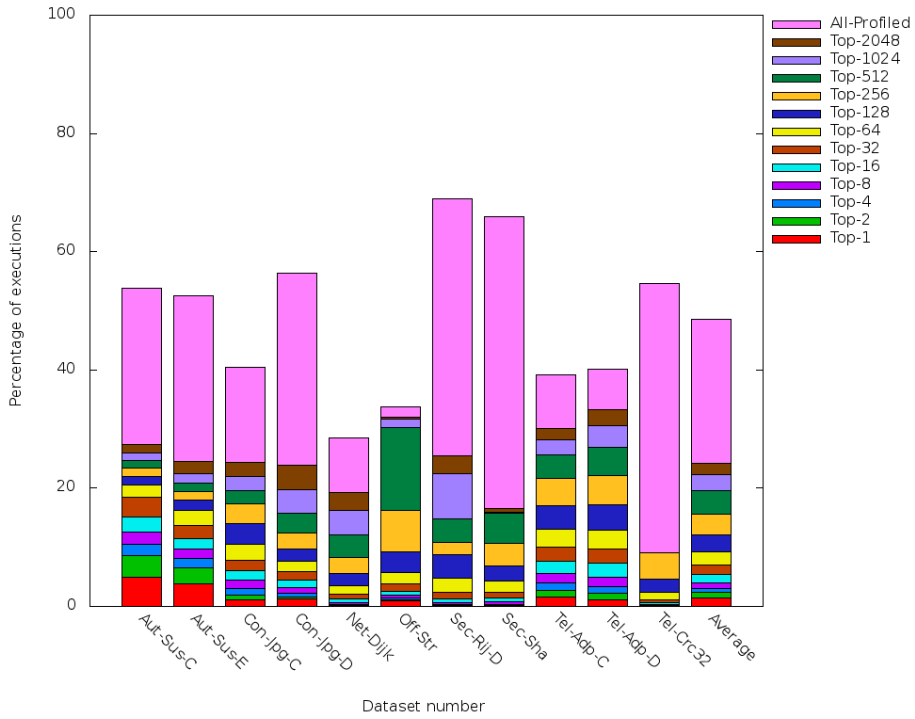

Figure 5.12: Comparison of the percentage of all instruction executions accounted for by the top N frequently occurring instructions across all benchmarks at global level on LLVM.

fraction of profiled instruction executions. Figure 5.13 shows the percentage of profiled instruction executions accounted for by the most frequent computations across all benchmarks. On average, just 1024 unique computations account for all profiled instruction executions. This is a very small fraction compared to the total number of profiled instruction executions throughout the benchmark, which is generally millions to tens of millions of profiled instructions.

It is possible that if additional instructions were profiled, the percentage of all instructions which are represented by the few most frequent computations would be greatly increased. The instructions which were chosen to be profiled were the ones which were most likely to make up the majority of all instruction executions, at the advice of the supervisor.

5.1.2 Global-level Memory Access Value Profiling

Automotive-susan-c. It can be observed from the results that:

- Up to 44.5% of all memory accesses involve the same distinct value being transferred across the memory bus. On average, around 32% of all memory accesses are due to a single distinct value.
- For most datasets, this value is either 100 or 255. A small number of other datasets transfer a different single distinct value.
- There is a large amount of Value Reuse present in memory accesses for this benchmark. It can be seen than on average, just 8 distinct values are involved in over 78% of all memory accesses. For dataset 3, almost 100% of memory accesses involved one of these 8 distinct values.
- There is a set of several values which are frequently transferred across the memory bus in all benchmarks. These are multiples of 100 between 100 and 1800.
- There is some variation in the amount of Value Reuse in Memory Accesses between datasets for this benchmark. An interesting result is that for Set 15, only two distinct values are involved in over 86% of all memory accesses, yet for set 19, the two most frequently transferred distinct values only account for 29% of all memory bus transfers.

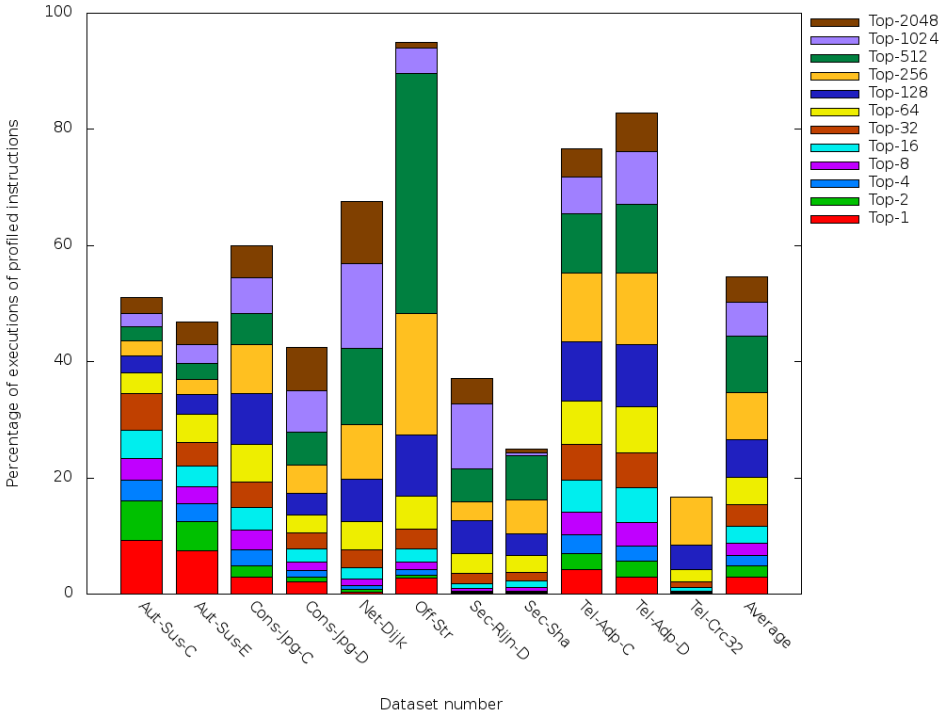

Figure 5.13: Comparison of the percentage of all profiled instruction executions accounted for by the top N frequently occurring instructions across all benchmarks at global level on LLVM.

- **The high incidence of Value Reuse in Memory Accesses for this benchmark provides strong support for Hypothesis 1**.

Automotive-susan-e. It can be observed from the results that:

- On average, there are similar levels of Value Reuse in Memory Accesses for this benchmark and *automotive-susan-c*. As with Value Reuse of Instruction Executions, this is thought to be due to the two benchmarks performing similar functions, and sharing some of the same code to perform these functions.
- Again there is a large variation in the amount of Value Reuse between datasets. For example, the variations between Set 4 and Set 15 are most pronounced for this benchmark.
- Despite this variation, the level of Value Reuse in Memory Accesses for this benchmark is still high. **This is further support for Hypothesis 1**.
- The most frequently transferred value for most datasets is again either 100 or 255 for this benchmark. However, frequent values transferred across the memory bus for most datasets also include small (< 10) integer values.

Consumer-jpeg-c. It can be observed from the results that:

- There is a more consistent level of Value Reuse in Memory Accesses across datasets than for the first two benchmarks examined. For each of the datasets, around 20% of all memory bus transfers involve a single distinct value.
- This value is always a load of the value 0 for all of the datasets. Additionally, the next most frequent bus transfer is always a store of 0. Therefore, on average, over 23% of all memory bus transfers involve a transfer of the value 0. This is shown by the height of the "Average - Top-2" bar on the graph.
- Although there is a lower level of Value Reuse in Memory Accesses for this benchmark than the previous two, the level of Value Reuse is still high. On average the set of the 32 most

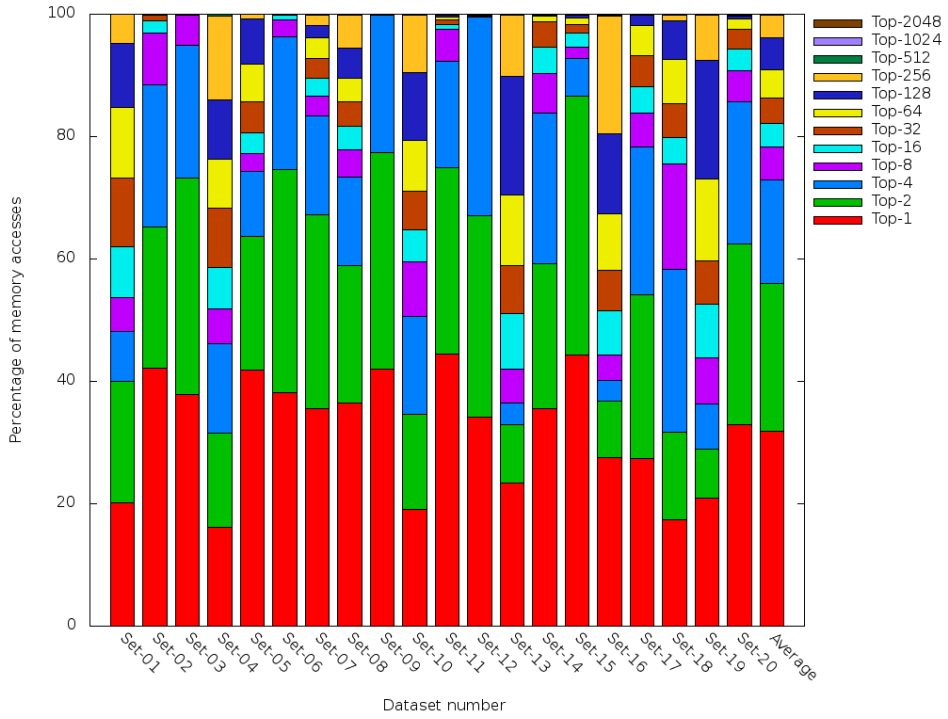

Figure 5.14: Automotive-susan-c. Percentage of all memory accesses accounted for by the top N most frequently transferred values at global level on LLVM.

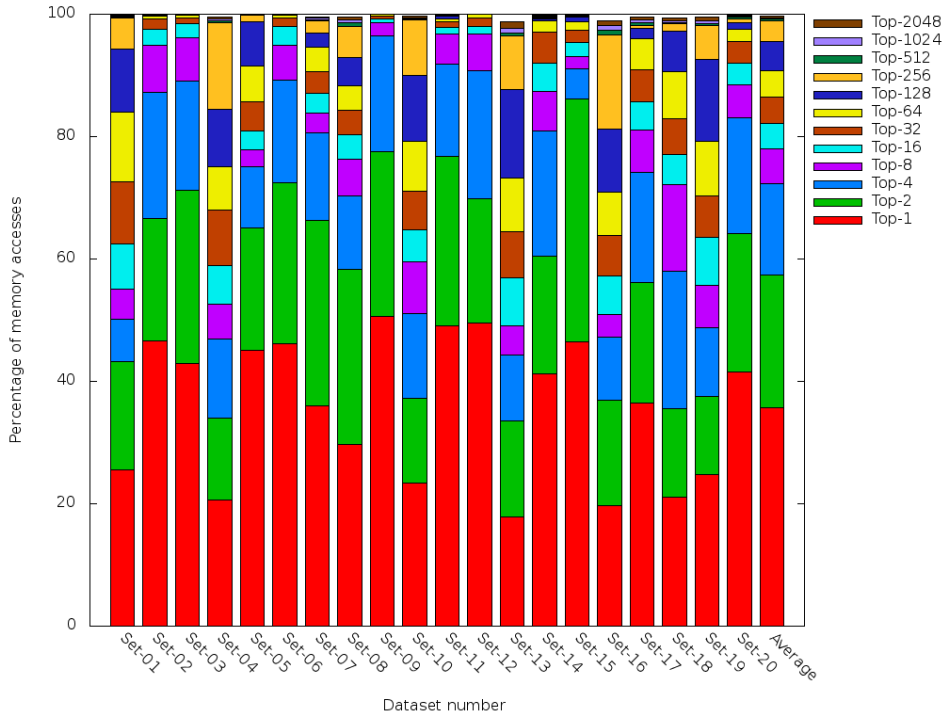

Figure 5.15: Automotive-susan-e. Percentage of all memory accesses accounted for by the top N most frequently transferred values at global level on LLVM.

frequently transferred values accounts for more than 50% of all bus traffic. **This is again strong support for Hypothesis 1**.

- Some of the most frequently transferred values which occur for most datasets again are small (< 10) integers.

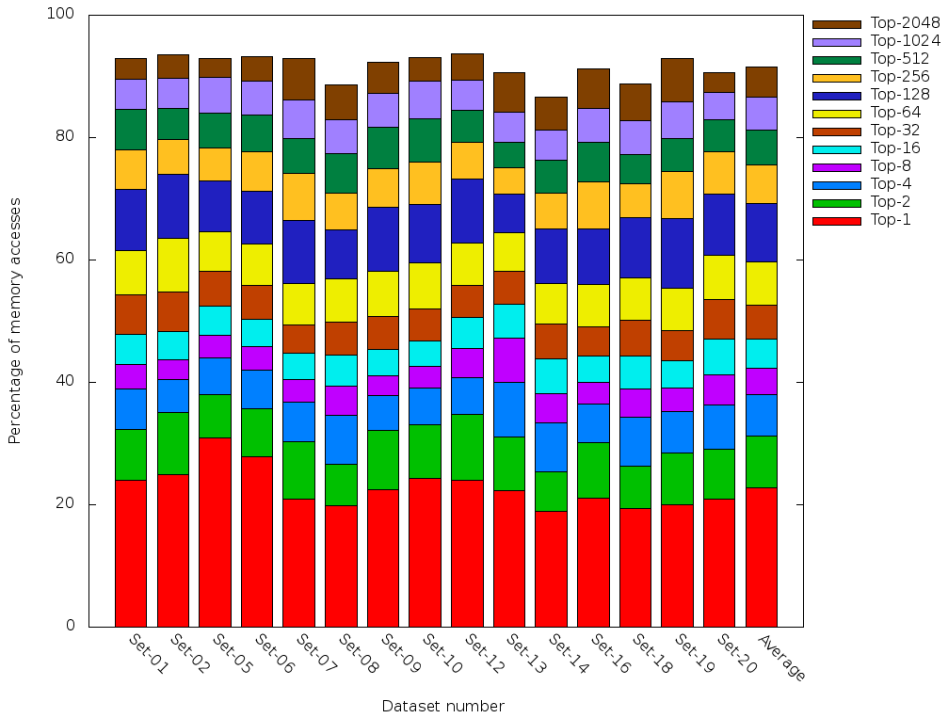

Figure 5.16: Consumer-jpeg-c. Percentage of all memory accesses accounted for by the top N most frequently transferred values at global level on LLVM.

Consumer-jpeg-d. It can be observed from the results that:

- There is a greater variation in the amount of Value Reuse for this benchmark than for *consumer-jpeg-c*. It is difficult to spot an obvious link between the amount of Value Reuse in Memory Accesses for this benchmark and the amount of Value Reuse in Memory Accesses for the *consumer-jpeg-c* benchmark for a particular dataset. This is similar to the Instruction Level Value Profile data for these same two benchmarks. Again this suggests that the way in which a JPEG image is decoded is very different to the encoding process.
- There is the least Value Reuse in Memory Accesses so far within this benchmark. On average, the top 8 most frequently transferred values are only involved in 26.8% of all memory bus transfers. **However, this is still a significant amount of Value Reuse, and provides support for Hypothesis 1**.
- As with *consumer-jpeg-c*, the most frequently transferred value is always 0, and this operation is a Load. The second most frequent transfer is almost always a store of 0.
- The to 8 most frequently transferred values for all of the datasets usually include the integers 0, 1, 2, 3, and 4.

Network-dijkstra. It can be observed from the results that:

- As with the Value Profile of Instruction Executions, there is a trend in the amount of Value Reuse in Memory Accesses against the size of the dataset.
- In this case, the size of the dataset does not have a great effect on the fraction of memory accesses involving the single most frequently transferred value, but has a larger effect on the fraction of memory transfers involving the set of the 32 most frequently transferred values.

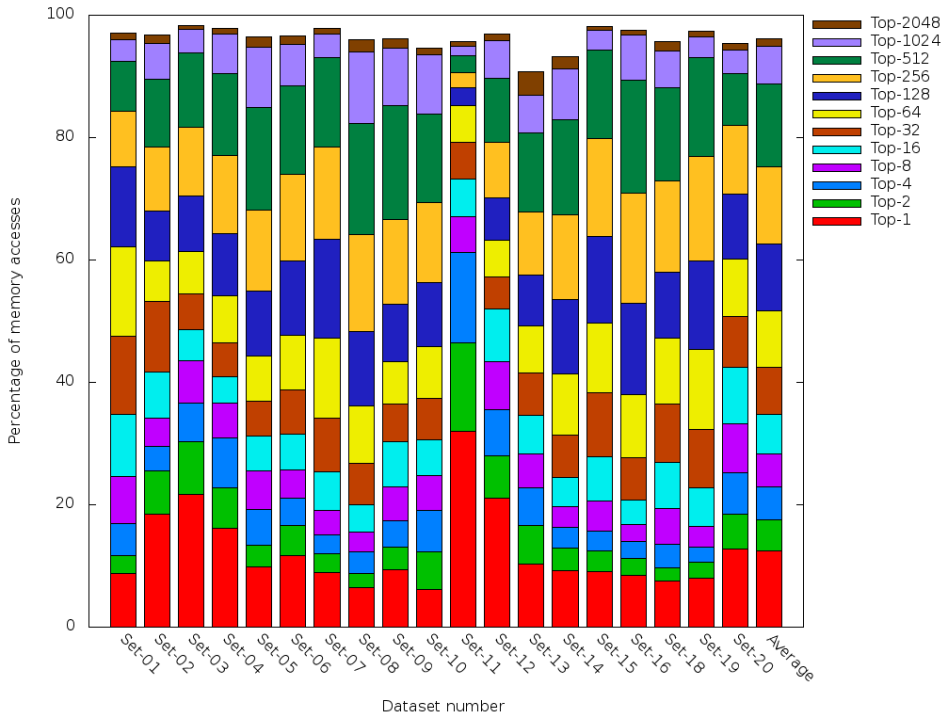

Figure 5.17: Consumer-jpeg-d. Percentage of all memory accesses accounted for by the top N most frequently transferred values at global level on LLVM.

- On average, the set of the 32 most frequently transferred values is involved in 47% of all memory bus transfers. The results for the first three datasets skew this value upwards - this average (the mean) is greater than the median value of the fraction of memory transfers involving the set of the 32 most frequently transferred values, which is approximately 40%. How representative this average is of a real-world application depends on the typical size of the input datasets to this algorithm in production use.
- There is no single value which is always the most frequently transferred value across all datasets for this benchmark. However, the top 8 most frequently transferred values generally consist of small integers, for all datasets for this benchmark.
- There is a definite occurrence of Value Reuse in Memory Accesses for this benchmark - **again this is in support of Hypothesis 1**.

Office-stringsearch. It can be observed from the results that:

- There is some variation in the amount of Value Reuse in Memory Accesses across datasets for this benchmark.
- However, there is a very high level of Value Reuse in Memory Accesses across all datasets. On average, the top 16 most frequently transferred values are involved in 88% of all memory bus transfers.
- This benchmark is dissimilar to the others in that the top 8 most frequently transferred values are always stored to, rather than loaded from memory.
- These stores are always of small (< 10) integers. As there are only a small number of integers less than 10, many of the same values are found in the top 8 most frequently transferred values across all datasets.

Security-rijndael-d. It can be observed from the results that:

- There is some Value Reuse in Memory Accesses present across all datasets for this benchmark. The amount of Value Reuse is very consistent across datasets.

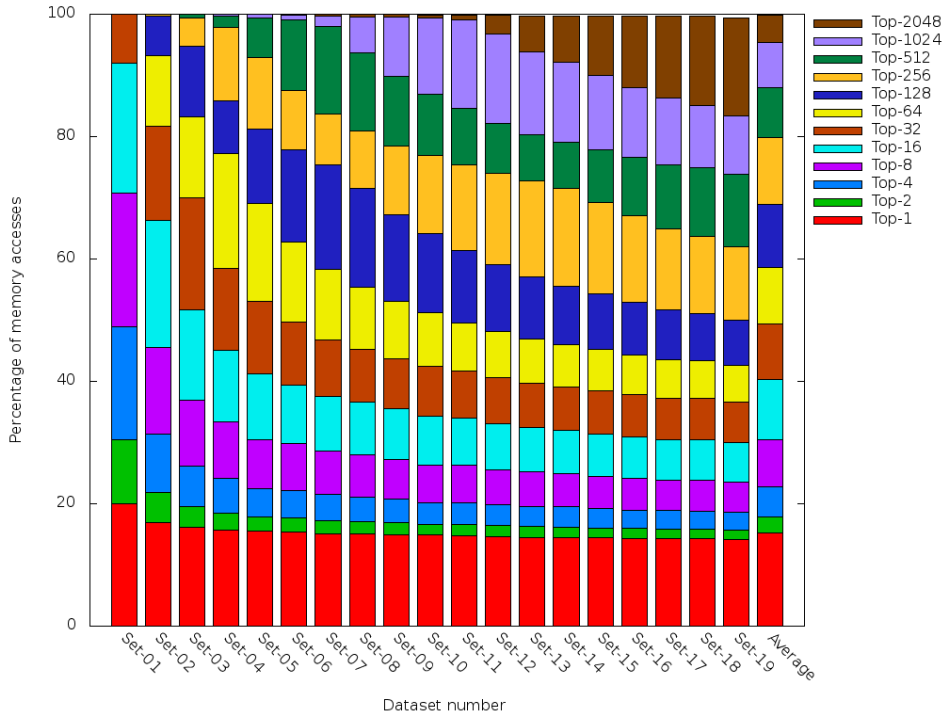

Figure 5.18: Network-dijkstra. Percentage of all memory accesses accounted for by the top N most frequently transferred values at global level on LLVM.

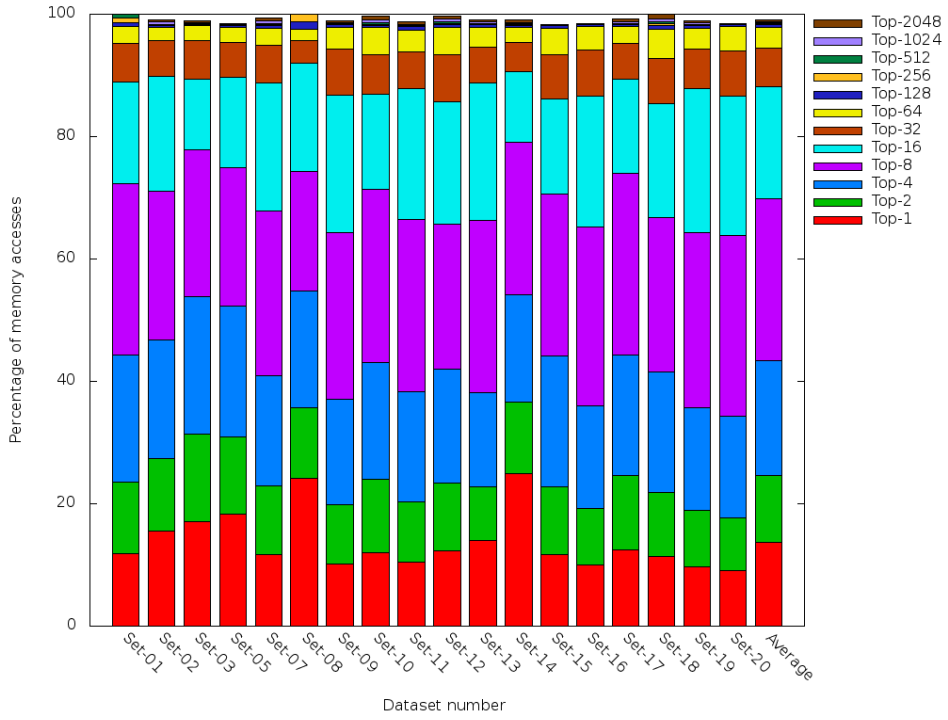

Figure 5.19: Office-stringsearch. Percentage of all memory accesses accounted for by the top N most frequently transferred values at global level on LLVM.

- Unlike other benchmarks so far, the most frequently transferred single value does not account for a large fraction of all memory bus transfers.
- However, Value Reuse is present. This can be concluded because it can be seen that larger sets of values do account for a significant fraction of memory bus transfers. For example, the 512 most frequently transferred values account for just under 50% of all memory bus transfers.
- As there is some Value Reuse present, **there is support for Hypothesis 1 provided by these results**.
- The 2048 most frequently transferred values are involved in almost all memory bus transfers.
- The two most frequently transferred values are generally 32 and 101, which are both stored to memory more frequently than they are loaded.
- Examining the full Value Profile output reveals that the majority of bus transfers are of large integers, which are only ever transferred a single time throughout the execution of the program. These generally occur as both loads and stores.

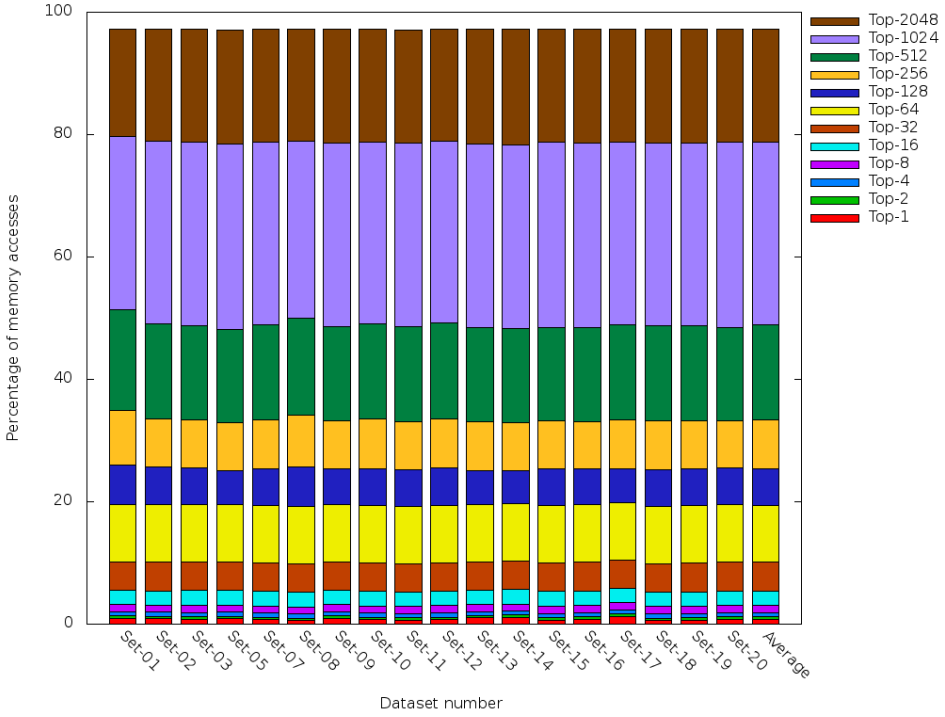

Figure 5.20: Security-rijndael-d. Percentage of all memory accesses accounted for by the top N most frequently transferred values at global level on LLVM.

Security-sha. It can be observed from the results that:

- This benchmark shows a lower level of Value Reuse in Memory Accesses than even *security-rijndael-d*. It can still be said that Value Reuse is present - for example, the top 64 most frequently transferred values do on average represent over 20% of all memory bus transfers.
- **This still provides support for Hypothesis 1**.
- The fraction of memory bus transfers represented by the top 2048 most frequently transferred values is much lower than for other benchmarks.
- The results suggest that it is likely that the majority of bus transfers involve values which are only transferred a single time or a small number of times throughout the lifetime of the benchmark. An inspection of the full profile data confirms this.

- The most frequently transferred value across all datasets is always 32, which is stored. Additionally, a load of this value is also the second most frequent transfer made. The next most frequently transferred value is generally 101. These values are the same as those most frequently transferred by the *security-rijndael-d* benchmark, which suggests that there is some similarity in their operation.

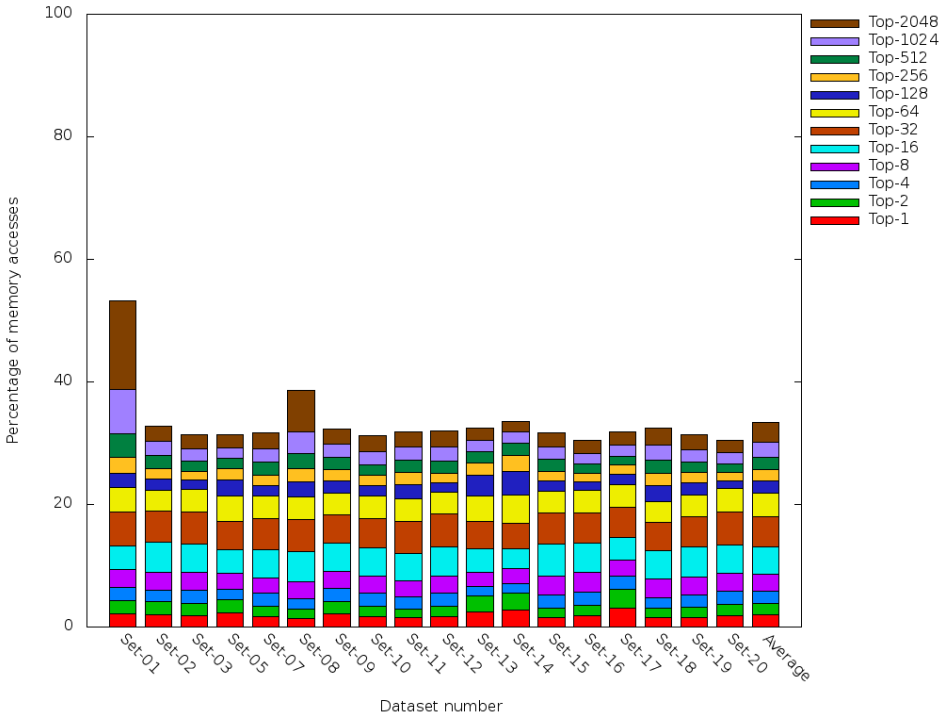

Figure 5.21: Security-sha. Percentage of all memory accesses accounted for by the top N most frequently transferred values at global level on LLVM.

Telecom-adpcm-c. It can be observed from the results that:

- There is a large amount of Value Reuse in Memory Accesses for this benchmark. **This is in support of Hypothesis 1**.
- The single most frequently transferred value is involved in 22% of all memory accesses on average.
- The most frequently transferred value across all datasets is 4294967295. However, the implementation of Value Profiling of Memory Accesses does not distinguish between signed and unsigned values. It is likely that this value actually represents -1, which is stored the same way as 4294967295 in two's complement notation, which is used by almost all modern machines.
- Other frequently transferred values include integers which can be represented within 16 bits across all the datasets. Often very small integers (< 10) are present in the top 8 most frequently transferred values, but it is difficult to argue that any of them appear frequently throughout all datasets.
- Apart from when considering the fraction of all memory bus transfers accounted for by the top most frequently transferred value, there is some variation across datasets in the amount of Value Reuse in Memory Accesses for this benchmark.

Telecom-adpcm-d. It can be observed from the results that:

- There is a large amount of Value Reuse in Memory Accesses for this benchmark. The amount of Value Reuse for a particular dataset in this benchmark generally exhibits some similarity to the amount of Value Reuse for the same dataset for *telecom-adpcm-c*.

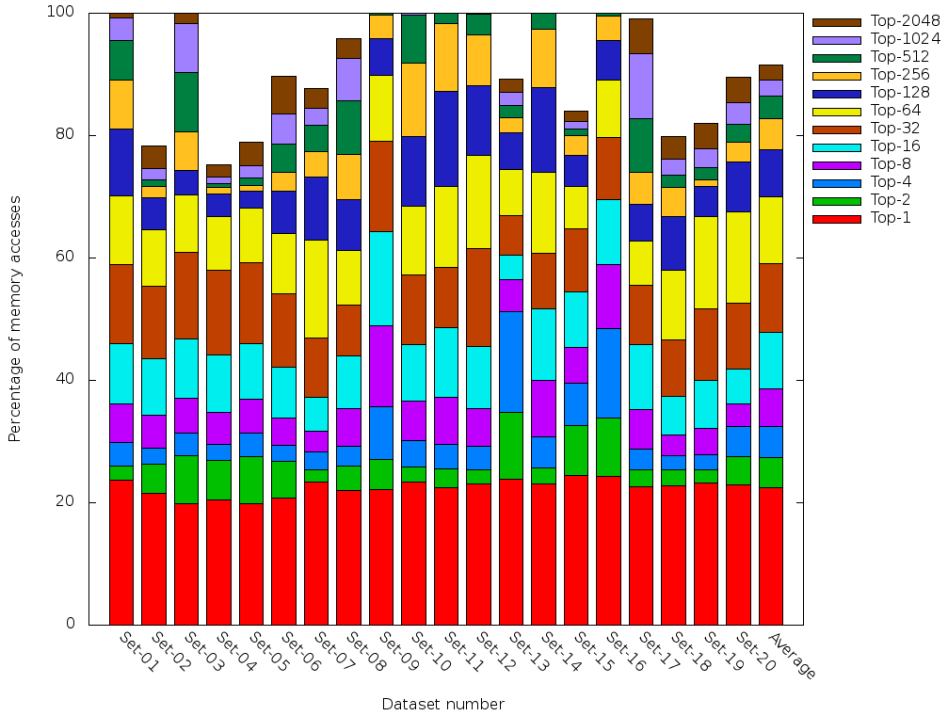

Figure 5.22: Telecom-adpcm-c. Percentage of all memory accesses accounted for by the top N most frequently transferred values at global level on LLVM.

- As with *telecom-adpcm-c*, the top most frequently transferred value is involved in 22% of all memory bus transfers. An inspection of the Value Profile data reveals that this is also -1.
- It is likely that the similarity between the Value Profiles for these two benchmarks is because the operations transcoding from PCM to ADPCM and from ADPCM to PCM are quite similar in operation. It is likely that some common code is used between these two operations.
- As with Telecom-adpcm-c, there is some variation in the amount of Value Reuse across datasets.
- As there is a large amount of Value Reuse present in this data, **additional support is provided for Hypothesis 1**.

Telecom-crc32. It can be observed from the results that:

- The level of Value Reuse in Memory Accesses is consistent across all datasets for this benchmark.
- The top most frequently transferred value does is not involved in a significant fraction of all memory bus transfers - yet all memory bus transfers involve 256 or less values.
- Examining the Value Profile data shows that of these 256 (or less values) almost all of them are transferred exactly the same number of times throughout the lifetime of the program. Therefore, the top most frequently transferred value may have only been transferred 1 or 2 more times than other values which were transferred, which is why it does not represent a significant fraction above all other values.
- **This data supports Hypothesis 1**, that Value reuse is prevalent, as almost all of the transferred values for this benchmark are frequently reused.
- Across all datasets, there does not seem to be a common set of frequently transferred values.

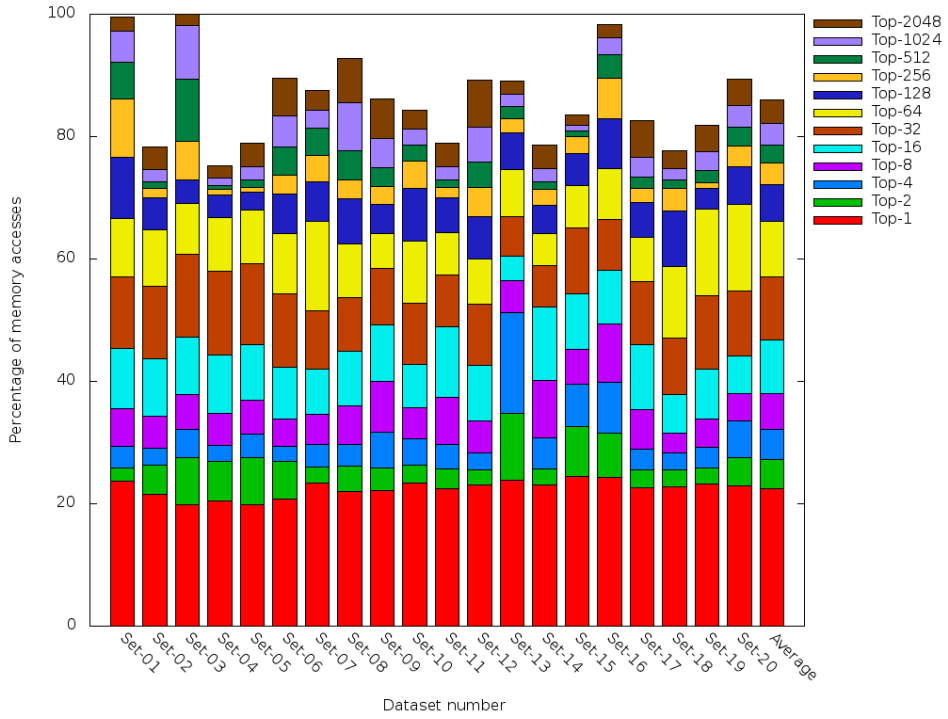

Figure 5.23: Telecom-adpcm-d. Percentage of all memory accesses accounted for by the top N most frequently transferred values at global level on LLVM.

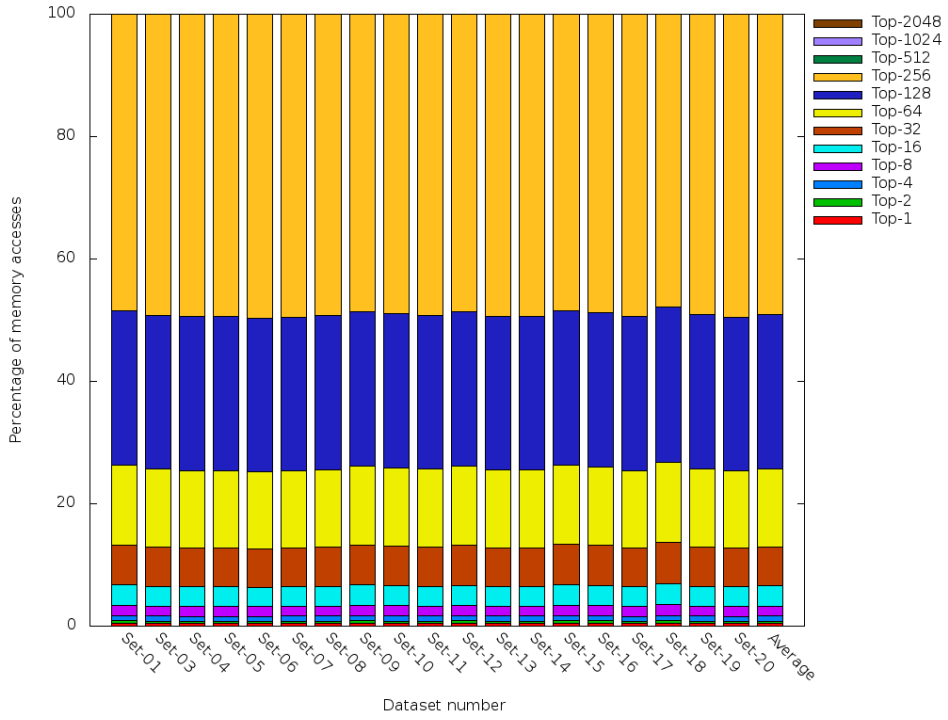

Figure 5.24: Telecom-crc32. Percentage of all memory accesses accounted for by the top N most frequently transferred values at global level on LLVM.

Comparison Across all Benchmarks

The benchmarks fall easily into two distinct groups:

- Those which have a high incidence of Value Reuse due to a very small number of frequently transferred values. These include *automotive-susan-c*, *automotive-susan-e*, *consumer-jpeg-c*, *consumer-jpeg-d*, *network-dijkstra*, *office-stringsearch*, *telecom-adpcm-c* and *telecom-adpcm-d*. All of these benchmarks have a significant fraction of memory accesses involving just one or two distinct values, and an even greater fraction if more values are considered.

- Those which do not have a high incidence of Value Reuse due to a very small number of frequently transferred values. These include *security-rijndael-d*, *security-sha* and *telecom-crc32*. However, *security-rijndael-d* and *telecom-crc32* do still have a high incidence of Value Reuse in Memory Accesses, when a larger number of distinct values are considered. As stated, all of the memory bus transfers occurring during the execution of Telecom-crc32 involved one of 256 distinct values, and almost all memory bus transfers involve one of 2048 distinct values during the execution of of *security-rijndael-d*. The only benchmark which does not have this characteristic is *security-sha*, which does not have a relatively large fraction of memory bus transfers which involve even 1 of 2048 distinct frequently transferred values.

Overall, there is generally a high level of Value Reuse in Memory Accesses. Across all the benchmarks 50% of all memory transfers involve one of 32 distinct frequently transferred values throughout the execution of the benchmark. **This provides strong support for Hypothesis 1.**

Additionally, it has been shown that for these benchmarks, programs which perform similar operations will generally have similar behaviour present in their Value Profiles. This has been seen for the pairs of *automotive-susan-c* and *automotive-susan-e*, *security-rijndael-d* and *security-sha*, and *telecom-adpcm-c* and *telecom-adpcm-d*. An exception to this pattern is the pair of *consumer-jpeg-c* and *consumer-jpeg-d*. This could be due to the algorithms for compression and decompression of a JPEG image being asymmetric, and operating in different ways to perform different functions.

It can also be observed that the benchmarks with high levels of Value Reuse in Instruction Executions also have high levels of Value Reuse in Memory Accesses. For example, *automotive-susan-c* and *-e* had high levels of Value Reuse in Instruction Executions and in Memory accesses, whereas the *security* benchmarks had low levels of Value Reuse in both Instruction Executions and Memory Accesses. **This is in support of Hypothesis 4**.

5.1.3 Local-Level Memory Access Value Profiling

On average it can be seen that there is less Value Reuse in Memory Accesses at the local level than at the global level. There are two groups which benchmarks may fall into:

- Benchmarks which have far less Value Reuse at the local level than at the global level. These include *automotive-susan-c* and *-e*, *consumer-jpeg-c* and *-d*, *network-dijkstra*, and *office-stringsearch*. These benchmarks are likely to repeatedly store a small number of distinct values throughout their memory space. The amount of Value Reuse at the local level is much lower because the same value stored in different locations is considered as a distinct value for each of the locations which it is stored in.

- Benchmarks which have a small reduction Value Reuse at the local level compared to the global level. This includes *security-rijndael-d*, *security-sha*, *telecom-adpcm-c* and *-d*, and *telecom-crc32*. These benchmarks are likely to repeatedly access the same small number of distinct values repeatedly from the same memory locations. The amount of Value Reuse at the local level would be similar to that at the global level, because these distinct values are not accessed from a wide variety of locations in memory.

These results are in support of Hypothesis 3, as there is greater Value Reuse at the global level than at the local level on average across all benchmarks.

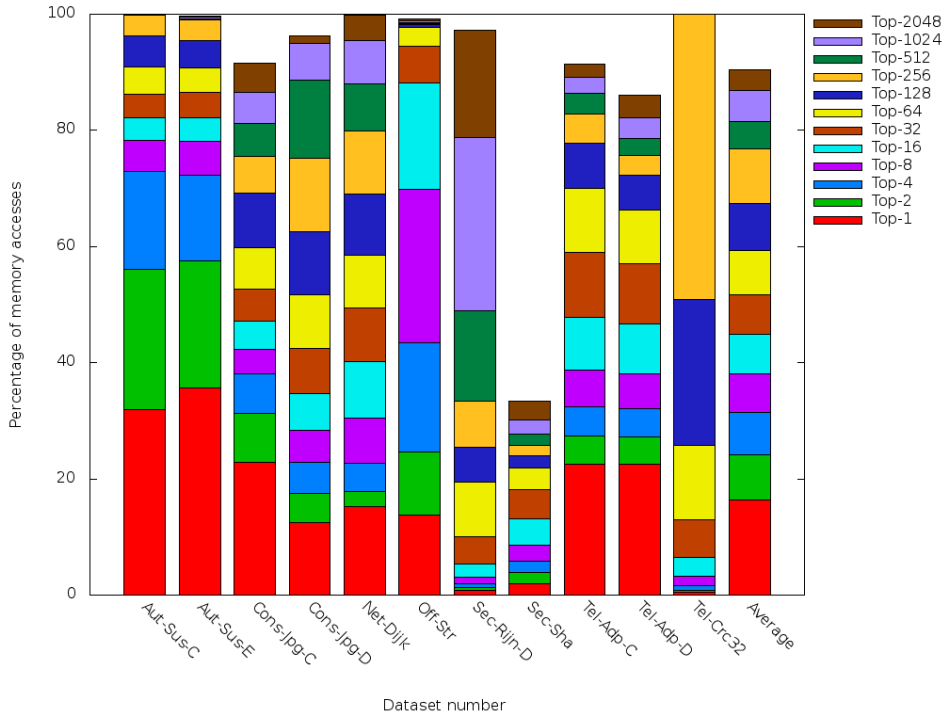

Figure 5.25: Comparison of the percentage of all memory accesses accounted for by the top N most frequently transferred values across all benchmarks at global level on LLVM.

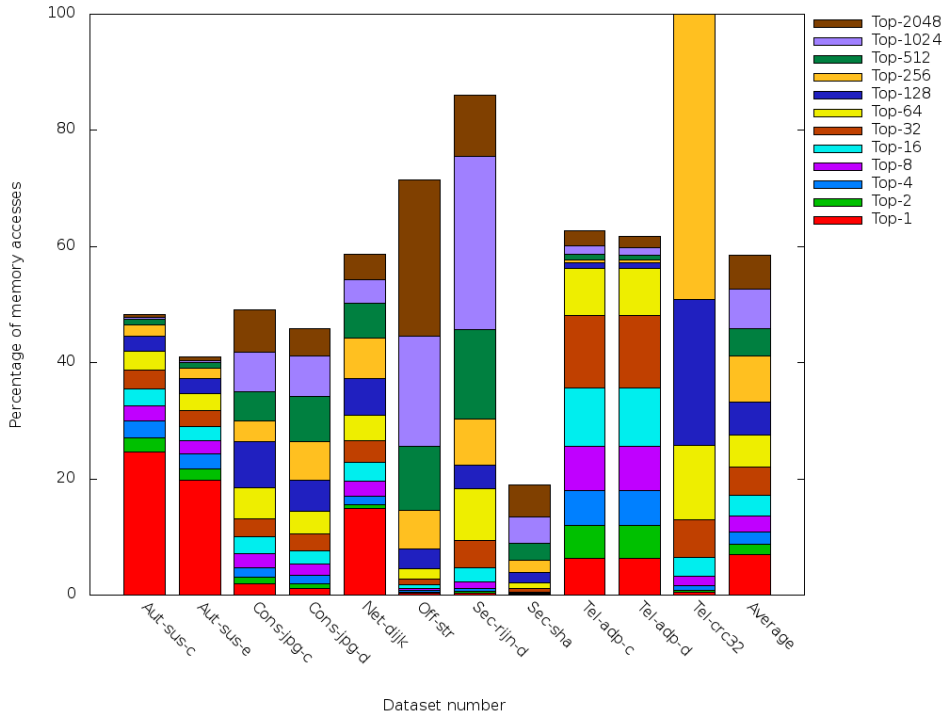

Figure 5.26: Comparison of the percentage of all memory accesses accounted for by the top N most frequently transferred values across all benchmarks at local level on LLVM.

5.2 X86 Architecture (Pin) Value Profile Data

5.2.1 Global-Level Instruction Value Profiling

Automotive-susan-c. It can be observed from the results that:

- The percentage of profiled instructions is lower using Pin than LLVM for this benchmark. However, the amount of Value Reuse within these profiled instructions appears to be similar between the x86 architecture (referred to as x86 from this point) and LLVM.
- For example, set 13 has relatively low levels of Value Reuse when executed using both LLVM and x86, and set 12 has relatively high levels of Value Reuse when executed on both LLVM and Pin.
- On average, the amount of Value Reuse is reduced approximately 33% when on the x86 compared to LLVM. For example, the top 32 most frequent computations account for around 18% of all instruction executions on LLVM, and only 12% of all instruction executions on the x86.

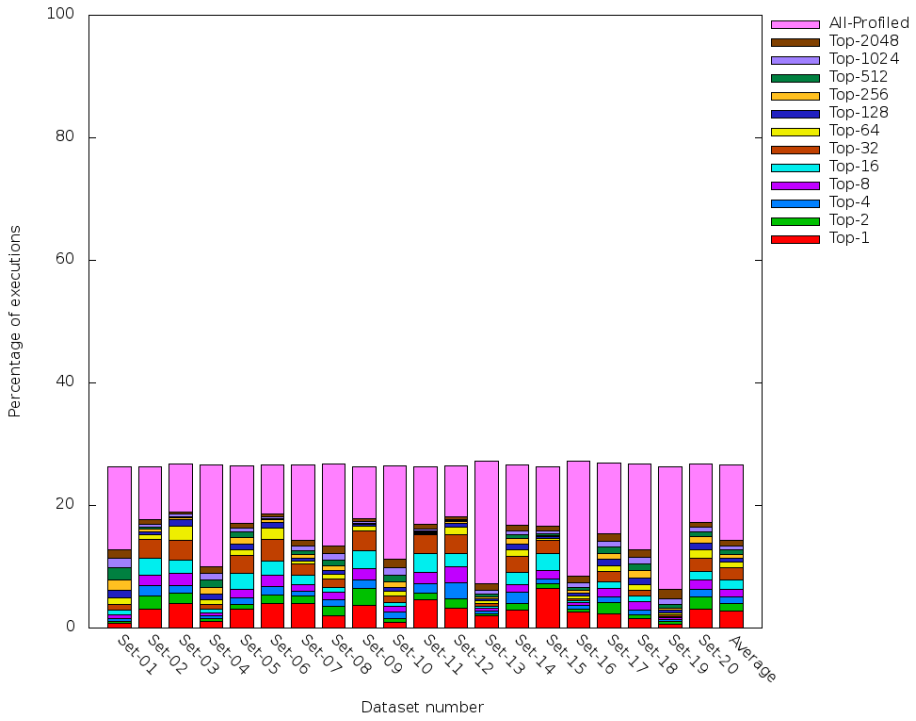

Figure 5.27: Automotive-susan-c. Percentage of all profiled instructions accounted for by the top N frequently occurring instructions at global level profiled using Pin.

Automotive-susan-e. It can be observed from the results that:

- As with the previous benchmark, the percentage of profile instructions is lower on the x86 than on LLVM for this benchmark, with a similar level of Value Reuse within these instructions.
- Again the amount of Value reuse is reduced approximately 33% on the x86 compared to on LLVM for this benchmark.

Consumer-jpeg-c. It can be observed from the results that:

- Similar observations may be made for this benchmark to the previous two, in that the levels of Value Reuse and profiled instructions are consistently lower on the x86 than on LLVM.

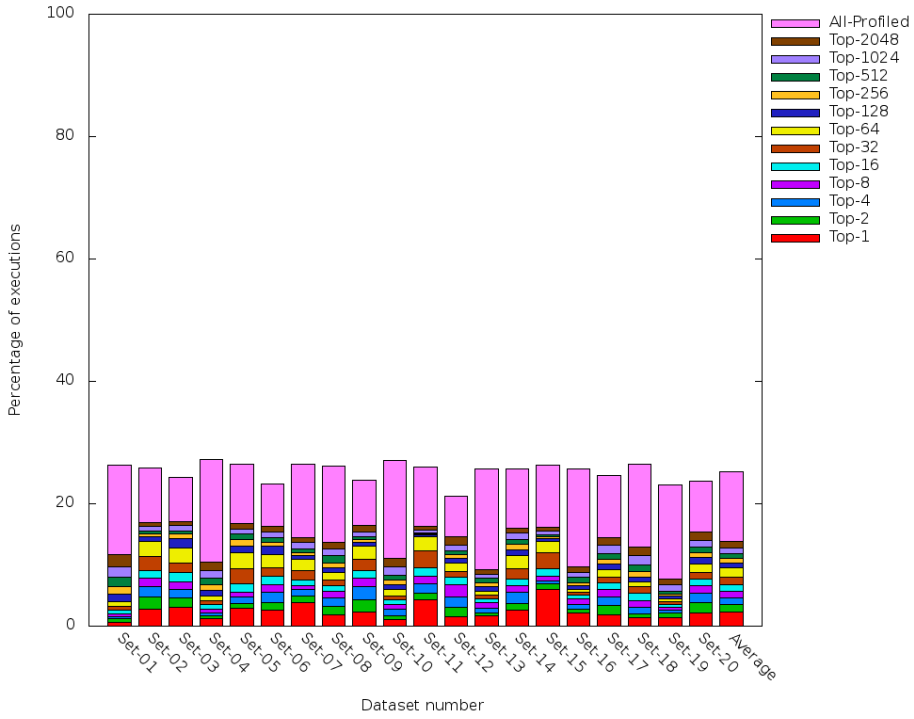

Figure 5.28: Automotive-susan-e. Percentage of all profiled instructions accounted for by the top N frequently occurring instructions at global level profiled using Pin.

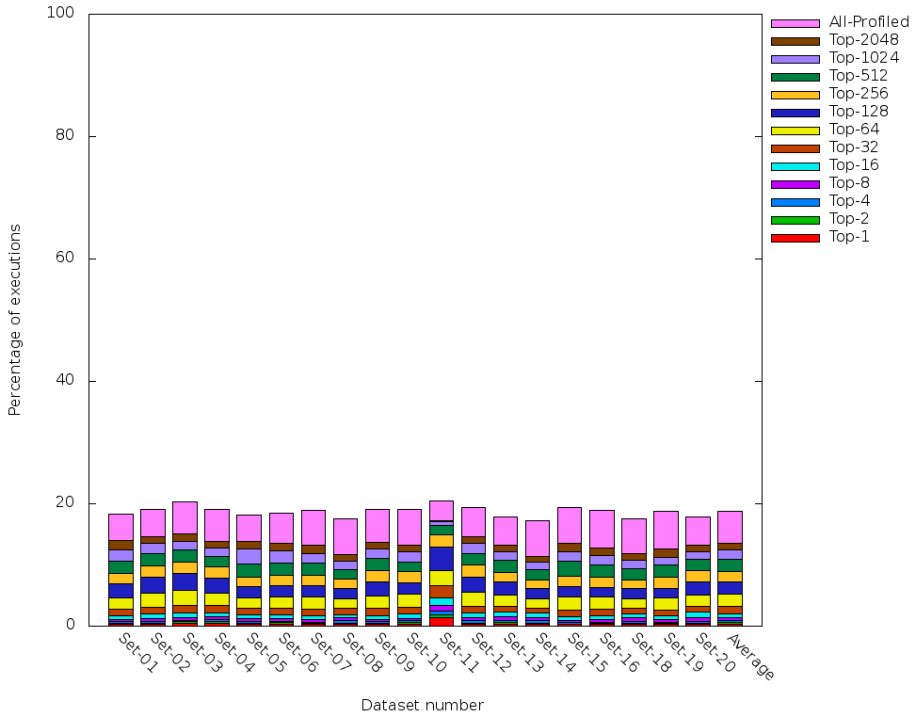

Figure 5.29: Consumer-jpeg-c. Percentage of all profiled instructions accounted for by the top N frequently occurring instructions at global level profiled using Pin.

Consumer-jpeg-d. It can be observed from the results that:

- Similar observations may be made to those of previous benchmarks.

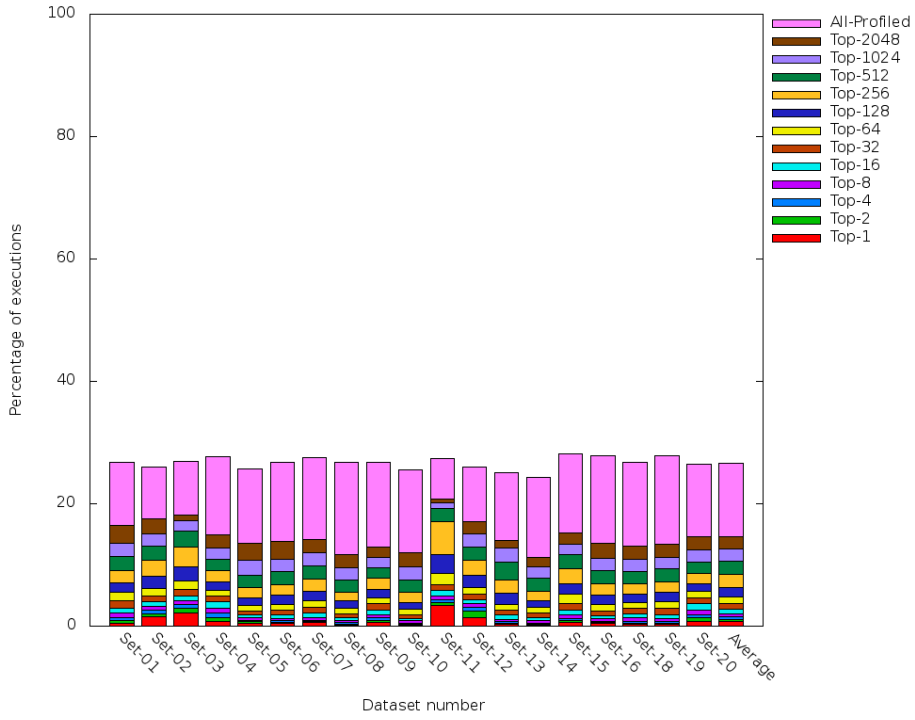

Figure 5.30: Consumer-jpeg-d. Percentage of all profiled instructions accounted for by the top N frequently occurring instructions at global level profiled using Pin.

Network-dijkstra. It can be observed from the results that:

- The percentage of profiled instructions is decreased by approximately 66% on the x86 compared to on LLVM.
- However, the amount of Value Reuse within the profiled instructions is similar for the results on the x86 compared to those on LLVM.

Office-stringsearch. It can be observed from the results that:

- Again the percentage of profiled instructions is reduced by approximately 33% on the x86 compared to on LLVM.
- Additionally the amount of Value Reuse within these profiled instructions is slightly lower on the x86 than on LLVM.

Security-rijndael-d. It can be observed from the results that:

- The amount of profiled instructions is reduced by approximately 66% on the x86 compared to on LLVM.
- The amount of Value Reuse within these profiled instructions is similar for both the x86 and LLVM.

Security-sha. It can be observed from the results that:

- As with Security-rijndael-d, the amount of profiled instructions is reduced by approximately 66%.

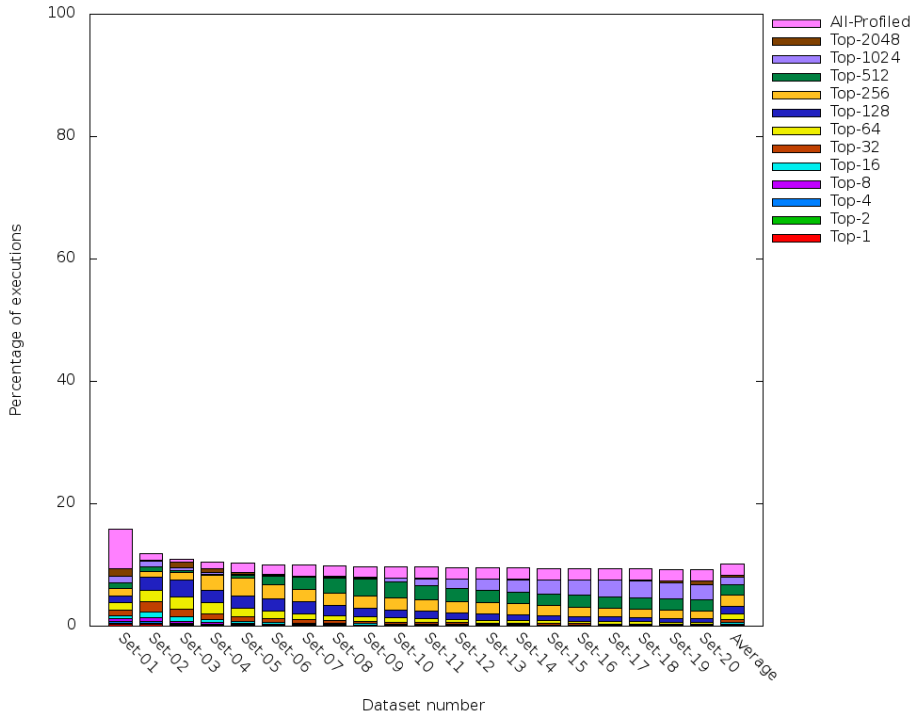

Figure 5.31: Network-dijkstra. Percentage of all profiled instructions accounted for by the top N frequently occurring instructions at global level profiled using Pin.

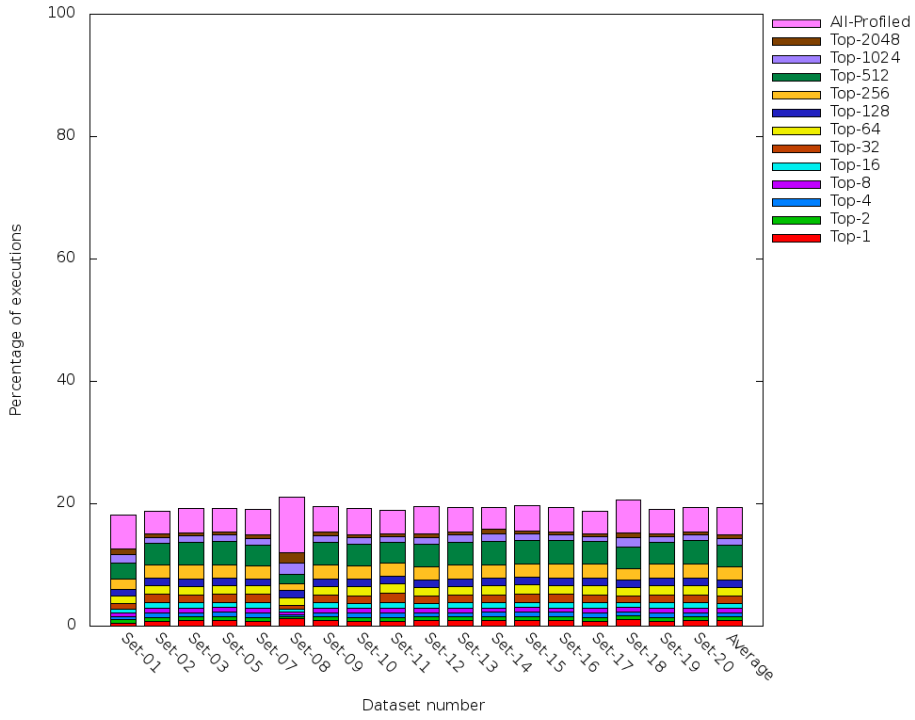

Figure 5.32: Office-stringsearch. Percentage of all profiled instructions accounted for by the top N frequently occurring instructions at global level profiled using Pin.

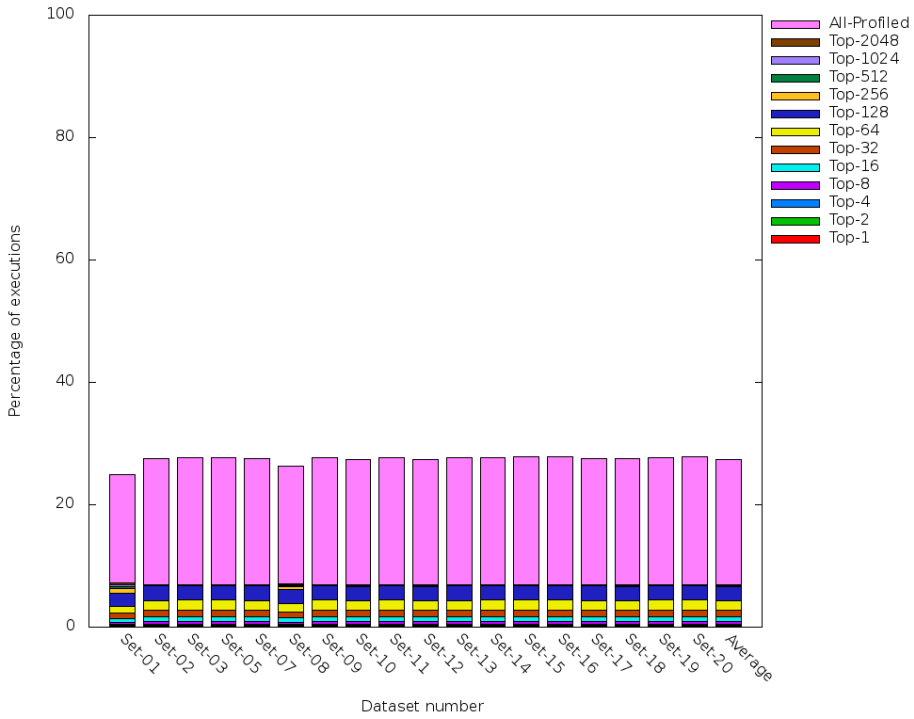

Figure 5.33: Security-rijndael-d. Percentage of all profiled instructions accounted for by the top N frequently occurring instructions at global level profiled using Pin.

- However, the amount of Value Reuse within these profile instructions is greatly increased on the x86 compared on LLVM. For example, on average the 512 most frequent computations account for almost 50% of all profiled instructions in the results on the x86, whereas they only account for around 20% of all profiled instructions in the results on LLVM.

Telecom-adpcm-c. It can be observed from the results that:

- The percentage of all instructions which are profiled are decreased by 66% on the x86 compared to on LLVM.
- The amount of Value Reuse within these profiled instructions appears to be similar on the x86 and LLVM.

Telecom-adpcm-d. It can be observed from the results that:

- Similar observations can be made for the results with this benchmark to Telecom-adpcm-c.

Telecom-crc32. It can be observed from the results that:

- The percentage of profiled instructions is decreased by approximately 75% on the x86 compared to on LLVM.
- However, the amount of Value Reuse within these instructions is greatly increased on the x86 compared to on LLVM. For example, the most frequent computation generally account for around 20% of all profiled instructions on the x86, compared to a very small fraction which cannot even be seen on the graph on LLVM.

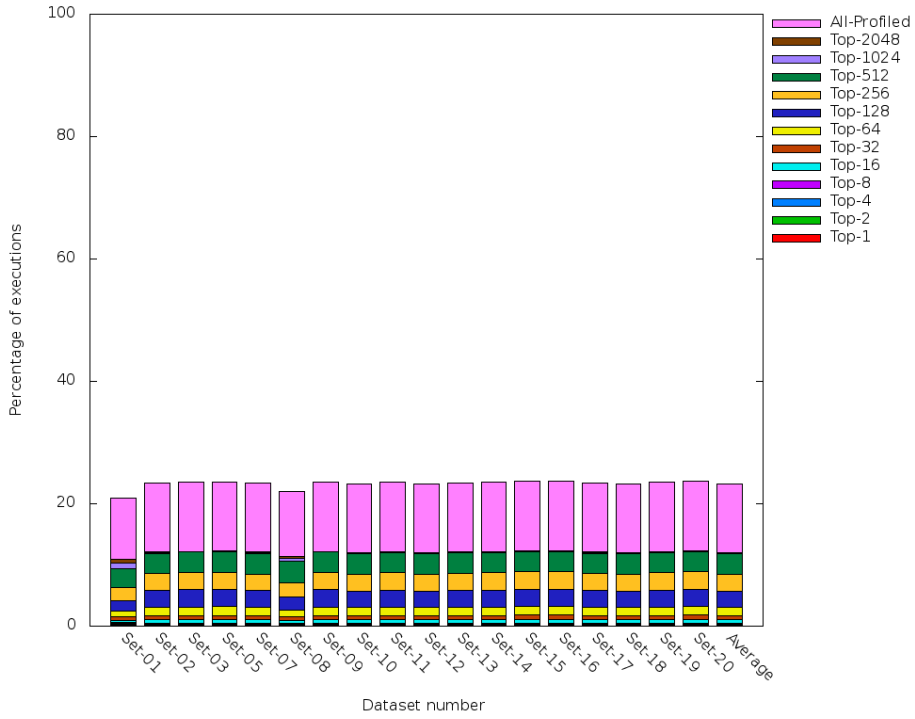

Figure 5.34: Security-sha. Percentage of all profiled instructions accounted for by the top N frequently occurring instructions at global level profiled using Pin.

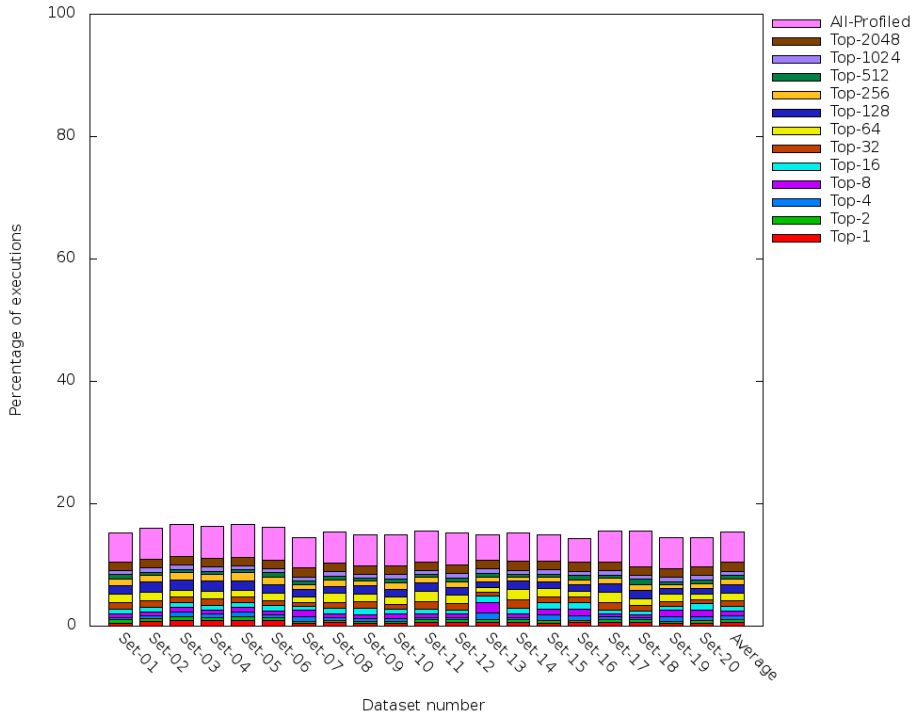

Figure 5.35: Telecom-adpcm-c. Percentage of all profiled instructions accounted for by the top N frequently occurring instructions at global level profiled using Pin.

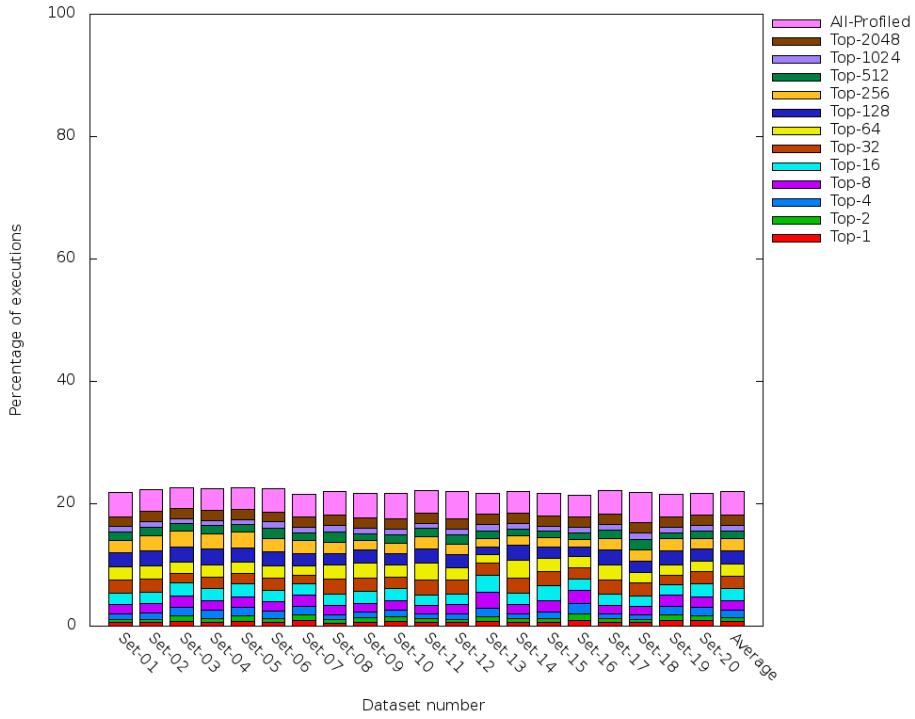

Figure 5.36: Telecom-adpcm-d. Percentage of all profiled instructions accounted for by the top N frequently occurring instructions at global level profiled using Pin.

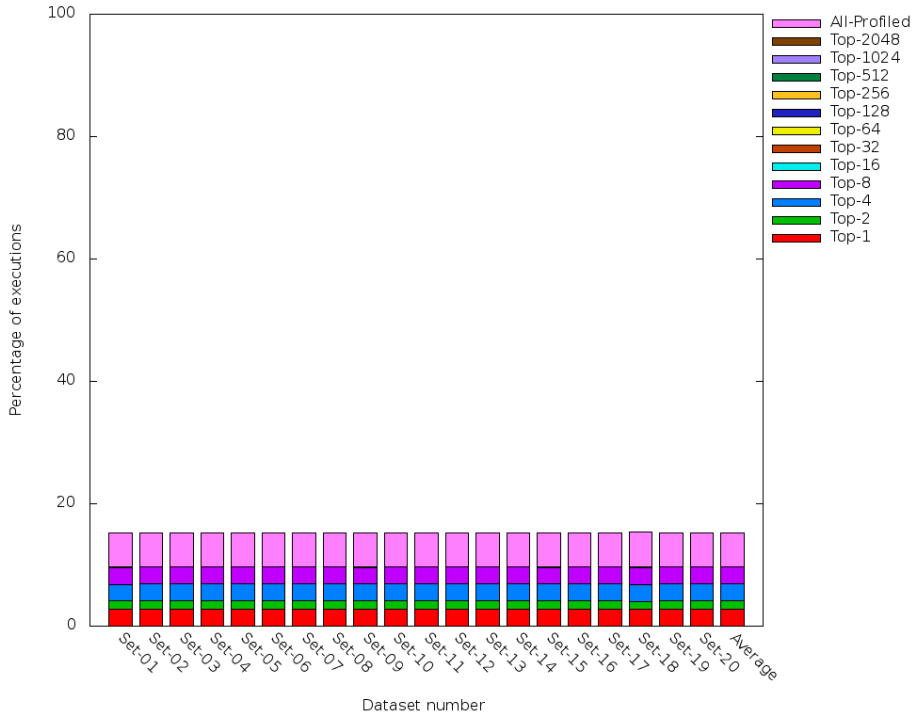

Figure 5.37: Telecom-crc32. Percentage of all profiled instructions accounted for by the top N frequently occurring instructions at global level profiled using Pin.

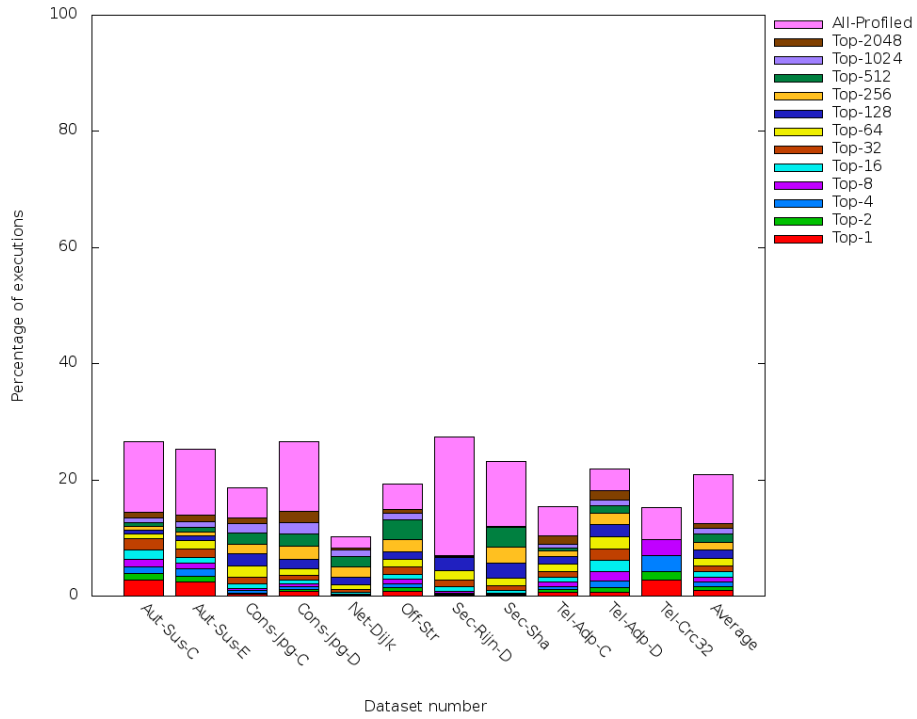

Figure 5.38: Comparison across all benchmarks of the percentage of all instructions accounted for by the top N frequently occurring instructions at global level profiled using Pin.

A Comparison Across all Benchmarks

On average it can be seen that around 20% of all instruction executions were profiled on the x86. This is less than on LLVM. Additionally, the amount of Value Reuse in Instruction Executions is increased for some benchmarks and decreased for others:

- Benchmarks which had less Value Reuse in Instruction Executions on the x86 architecture than on LLVM include: *automotive-susan-c* and *-e*, *consumer-jpeg-c* and *-d*, *network-dijkstra*, *office-stringsearch*, and *telecom-Adpcm-c* and *-d*.

- Benchmarks which had greater Value Reuse in Instruction Executions on the x86 architecture than on LLVM include: *security-rijndael-c*, *Security-sha*, and *telecom-crc32*.

This provides some support for Hypothesis 1, as there is some Value Reuse present in Instruction Executions on the x86 architecture as well as LLVM, even though the amount of Value Reuse is generally less. However, this does not support Hypothesis 7, as it does not appear that Instruction-level Value Profile data of Instruction Executions on LLVM is representative of Value Profile data of Instruction executions on the x86 architecture. This conclusion has been drawn as it is not possible to predict the amount of Value Reuse in Instruction Executions on the x86 using only the results on LLVM - this is because the amount of Value Reuse may be either increased or decreased on the x86 compared to on LLVM. Therefore, it is not possible to determine a way to transform the Value Profile of Instruction Execution data on LLVM to a form which is representative of Value Profile data of Instruction Executions on the x86.

Considering only profiled instructions

As with LLVM, the number instructions profiled was not always a large percentage of the total number of instruction executions. Figure 5.39 Examining the Value Reuse within only profiled instruction executions reveals that the amount of Value Reuse in Instruction Executions is even higher on the x86 architecture

than on LLVM - the top 1024 most frequent computations represent almost 60% of all profiled instruction executions.

The *telecom-crc32* benchmark in particular has a much higher level of Value Reuse within the profiled instructions on the x86 than on LLVM - the single most frequent computation represents almost 20% of all profiled executions, whereas on LLVM the single most frequent computation represents far less than 1% of profiled executions. The *security-sha* benchmark also has a large increase in Value Reuse on x86 compared to on LLVM. The top 512 most frequent computations account for around 50% of all profiled executions, compared to just over 20% on LLVM. The only benchmarks which have decreased levels of Value Reuse within the profiled instructions on the x86 are *office-stringsearch* and *telecom-adpcm-c*. However, the decreases in both of these benchmarks are relatively small.

Again these results are not in support of Hypothesis 7. This is due to the fact that Value Reuse within profiled instructions may be greater for one benchmark and less on another, on the x86 compared to on LLVM. This makes it impossible to predict the amount of Value Reuse in the execution of a program on the x86 based on only the amount of Value Reuse in the execution of the program on LLVM.

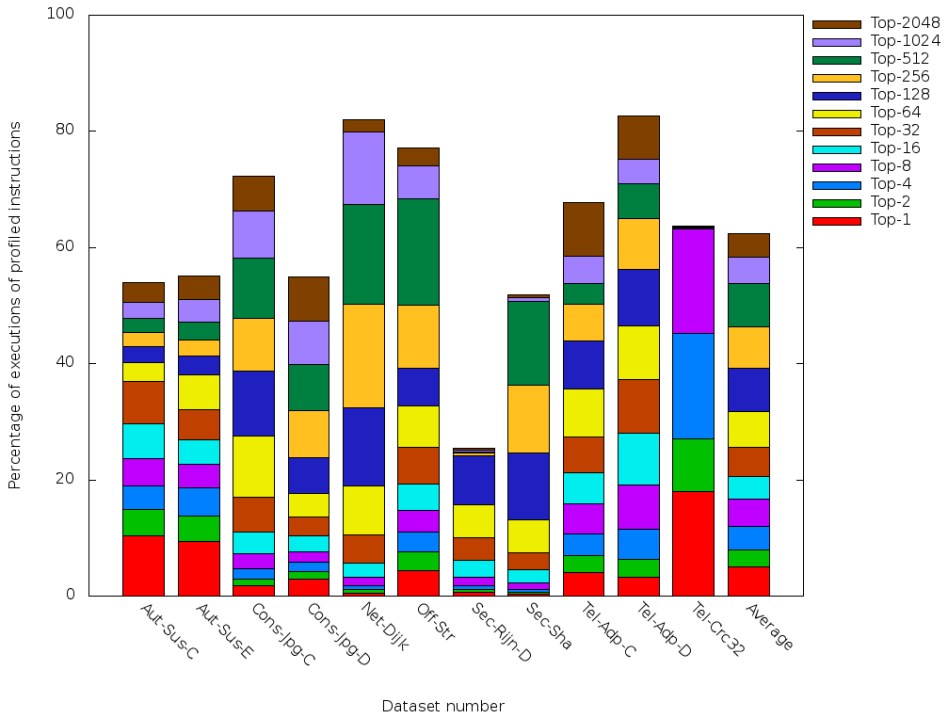

Figure 5.39: Comparison across all benchmarks of the percentage of all profiled instructions accounted for by the top N frequently occurring instructions at global level profiled using Pin.

5.2.2 Global-level Memory Value Profiling

Automotive-susan-c. It can be observed from the results that:

- There is less Value Reuse in Memory Accesses for this benchmark when executed on the x86 architecture than when executed on the LLVM Interpreter.
- For example, the top 32 most frequently transferred values only account for around 40% of all transferred values on the x86 architecture, whereas on LLVM the top 32 most frequently transferred values make up over 85% of all memory accesses.
- The amount of Value Reuse for this benchmark in each dataset follows a similar trend on the x86 architecture to the trend on LLVM. For example, set 13 has relatively low levels of Value Reuse on both the x86 architecture and LLVM.

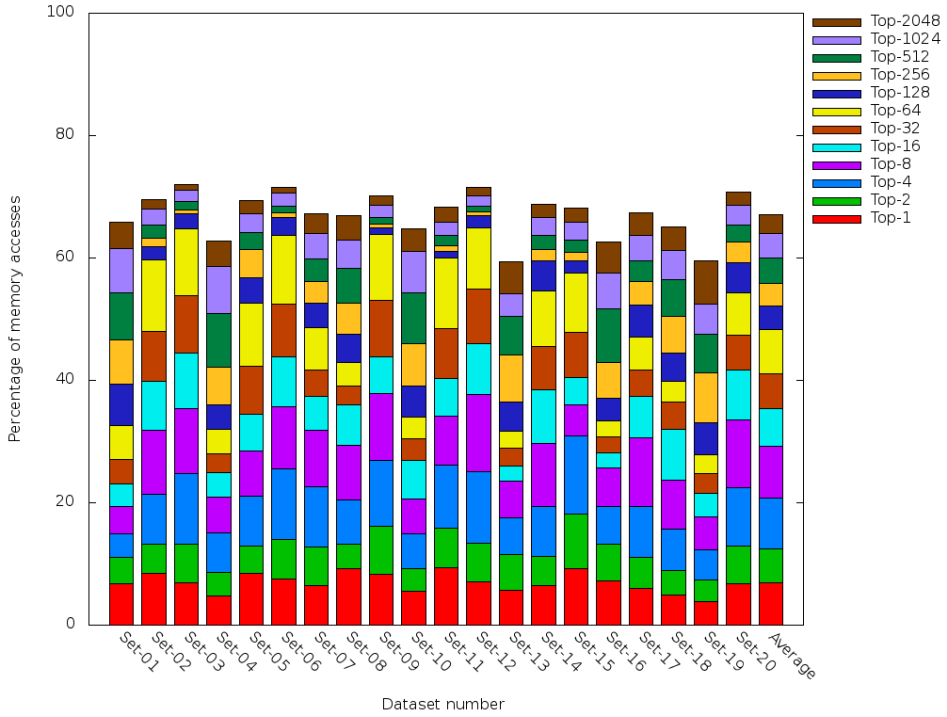

Figure 5.40: Automotive-susan-c. Percentage of all memory accesses accounted for by the top N most frequently transferred values at global level using Pin.

Automotive-susan-e. It can be observed from the results that:

- As with Automotive-susan-c, the amount of Value Reuse in Memory Accesses is greatly decreased on the x86 architecture compared to LLVM.
- There appears to be some link between the amount of Value Reuse for a particular dataset on the x86 architecture to the amount of Value Reuse on LLVM. For example, set 13 shows relatively low levels of Value Reuse on both architectures whereas set 12 shows relatively high levels of Value Reuse.

Consumer-jpeg-c. It can be observed from the results that:

- There does not appear to be a great difference in the amount of Value Reuse on the x86 architecture compared to the amount of Value Reuse on LLVM for this benchmark.
- The amount of Value Reuse is slightly reduced on the x86 architecture compared to LLVM.
- The trends in the amount of Value Reuse between datasets do not appear to be similar on the x86 architecture and LLVM.

Consumer-jpeg-d. It can be observed from the results that:

- There is a small decrease in the amount of Value Reuse on the x86 architecture compared to on LLVM. This difference is more pronounced for smaller sets of frequently transferred values. For example, the percentage of all memory accesses accounted for by the single most frequently transferred value is decreased nearly 50% of the x86 architecture compared to LLVM, but the percentage for the top 32 most frequently transferred values is only decreased 25%.
- The trend in the amount of Value Reuse between datasets appears to be similar on the x86 architecture and LLVM.

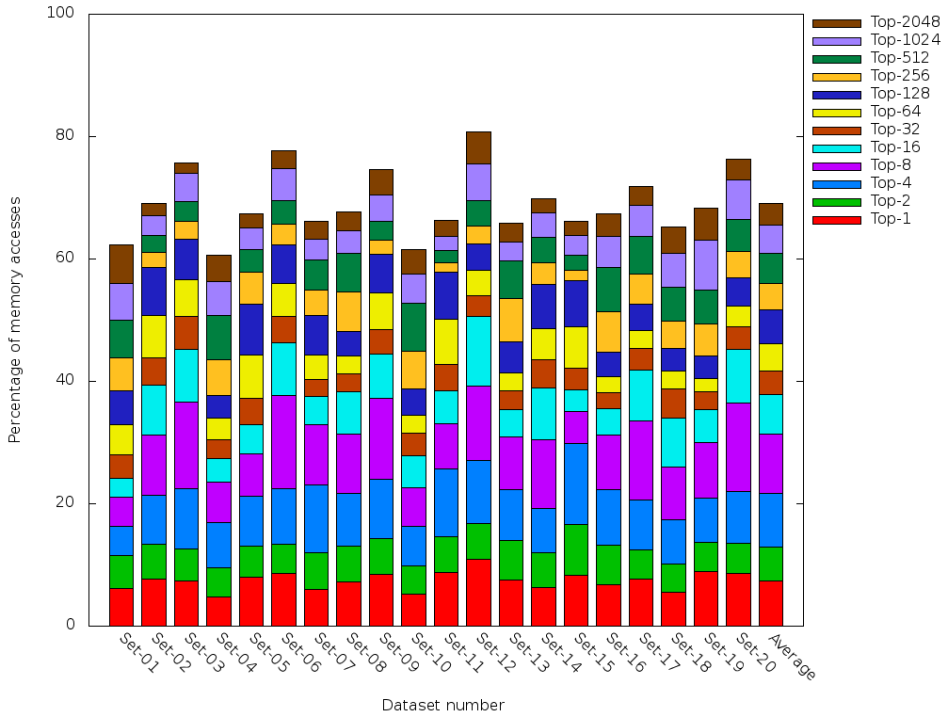

Figure 5.41: Automotive-susan-e. Percentage of all memory accesses accounted for by the top N most frequently transferred values at global level using Pin.

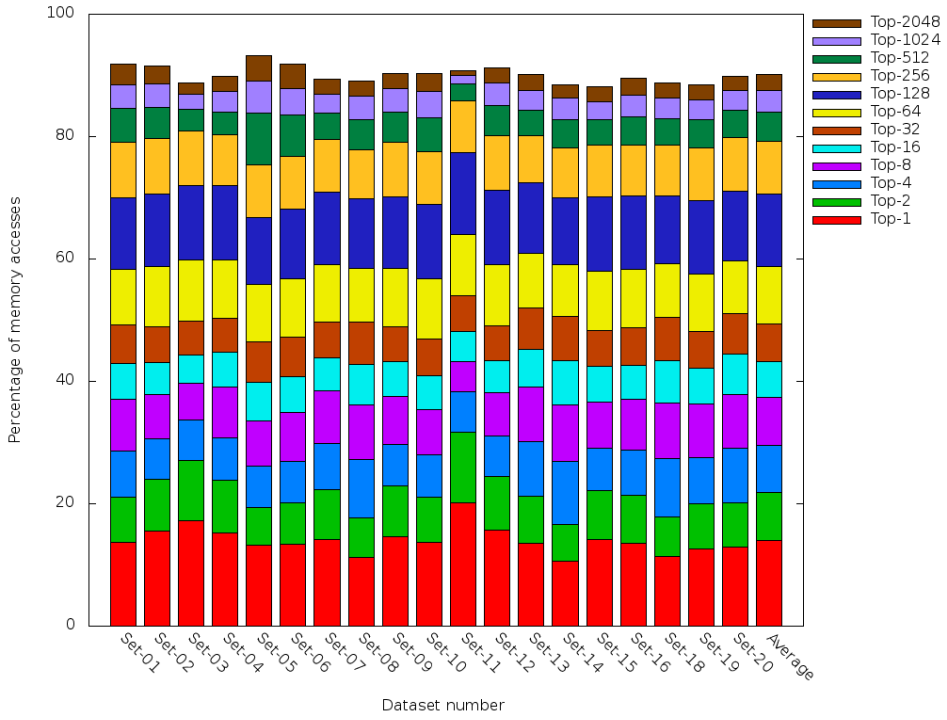

Figure 5.42: Consumer-jpeg-c. Percentage of all memory accesses accounted for by the top N most frequently transferred values at global level using Pin.

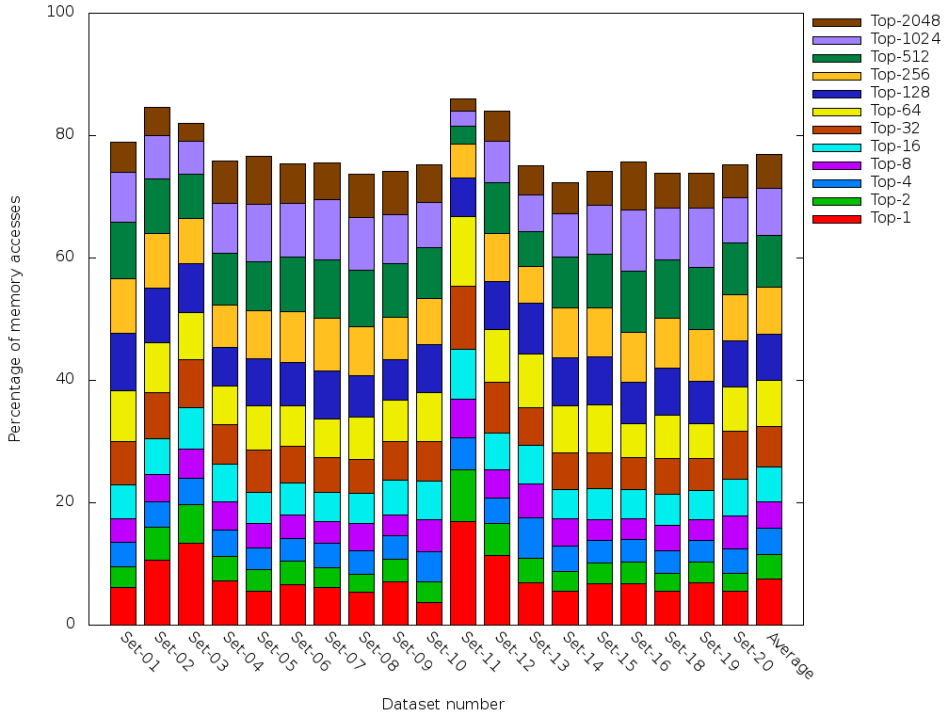

Figure 5.43: Consumer-jpeg-d. Percentage of all memory accesses accounted for by the top N most frequently transferred values at global level using Pin.

Network-dijkstra. It can be observed from the results that:

- There is a reduction in Value Reuse on the x86 architecture compared to on LLVM. As with Consumer-jpeg-c, this reduction is larger for smaller sets of frequently transferred values.
- The trend between datasets is similar on both the x86 architecture and on LLVM. The only exception to this is the first dataset, where Value Reuse is disproportionately reduced on the x86 architecture. It is likely that this is because the first dataset is so small that the values transferred when the program is first loaded and initialised affect the amount of Value Reuse. LLVM does not have this overhead, as execution begins at the first instruction in the main function.

Office-stringsearch. It can be observed from the results that:

- The amount of Value Reuse on the x86 architecture is reduced compared to on LLVM. For this benchmark the reduction is more significant. For example, the percentage of all memory accesses involving the single most frequent value is reduced by approximately 50%, whereas the percentage involving the top 16 most frequent values is reduced by around 66%.
- This is likely to be because this benchmark only has a small set of working values (upper and lower case letters, numbers and some punctuation) which creates a very high level of Value Reuse on LLVM. However, the x86 architecture will transfer other values as it has only a finite number of registers, which will dilute this small set of working values.

Security-rijndael-d. It can be observed from the results that:

- Unlike other benchmark, the amount of Value Reuse on the x86 architecture is greater than that on LLVM for small numbers of frequent values.
- It is thought that this is because the x86 architecture does not have infinite registers[1], it

[1]This is in contrast to LLVM, which does have an infinite number of registers available.

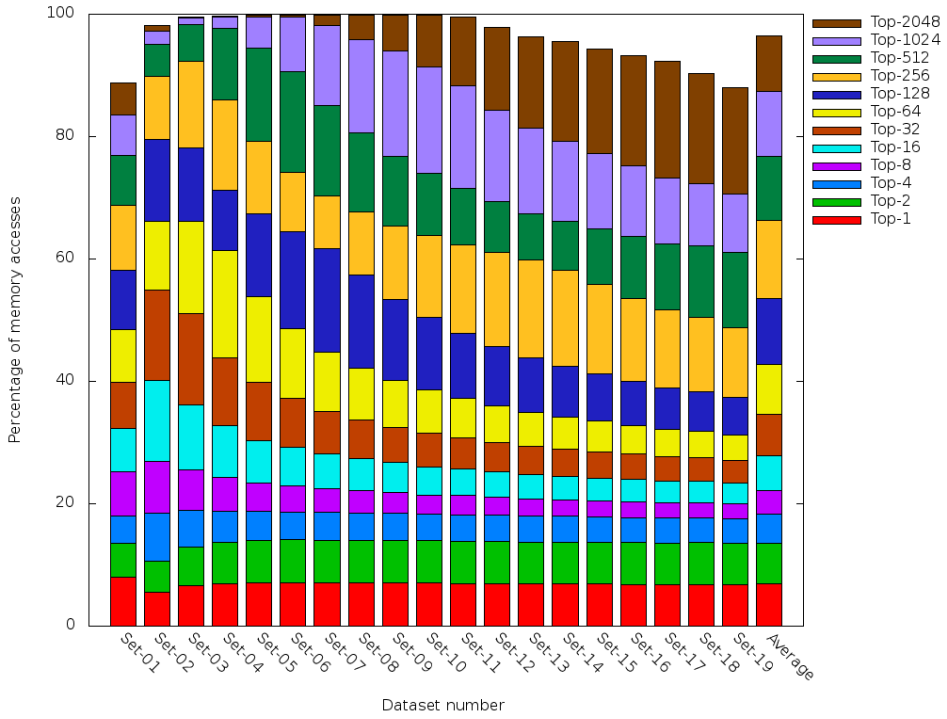

Figure 5.44: Network-dijkstra. Percentage of all memory accesses accounted for by the top N most frequently transferred values at global level using Pin.

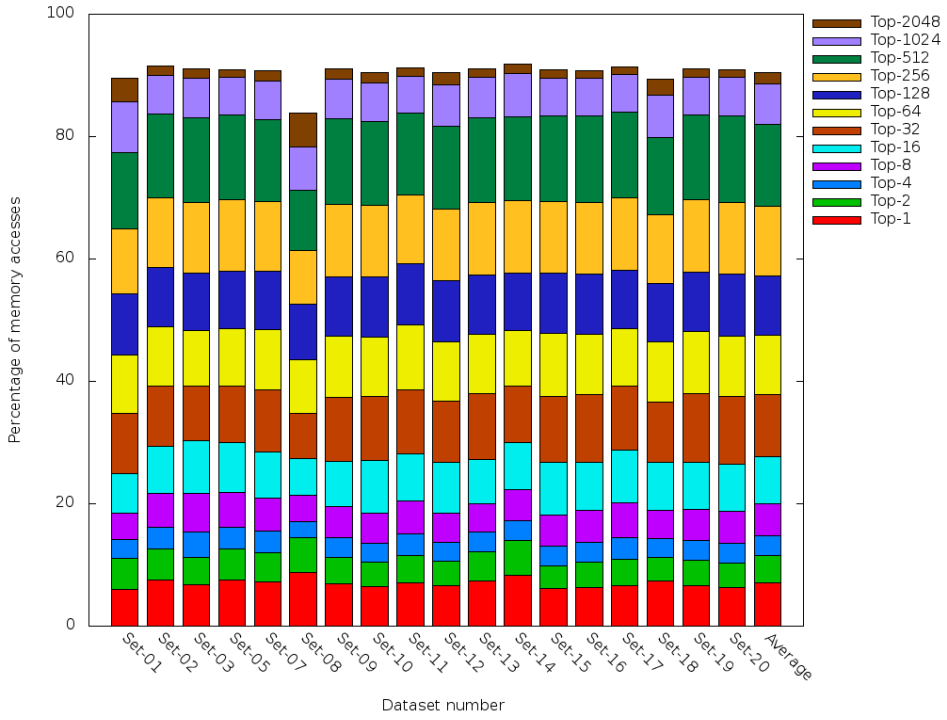

Figure 5.45: Office-stringsearch. Percentage of all memory accesses accounted for by the top N most frequently transferred values at global level using Pin.

must therefore spill values into memory. These values which LLVM does not need to store to memory are likely frequently be the same value.

- Larger sets of frequently transferred values still make up a smaller percentage of all memory accesses on the x86 architecture than on LLVM. For example, the top 2048 frequent values account for almost all memory accesses on LLVM, but only around 70% of all memory accesses on the x86 architecture.
- This is also likely to be due to the x86 architecture having to store more values to memory which LLVM would be able to store in registers.
- As the amount of Value Reuse is very similar for all datasets on both LLVM and the x86 architecture, it can also be said that the trend between datasets is similar on LLVM and the x86 architecture.

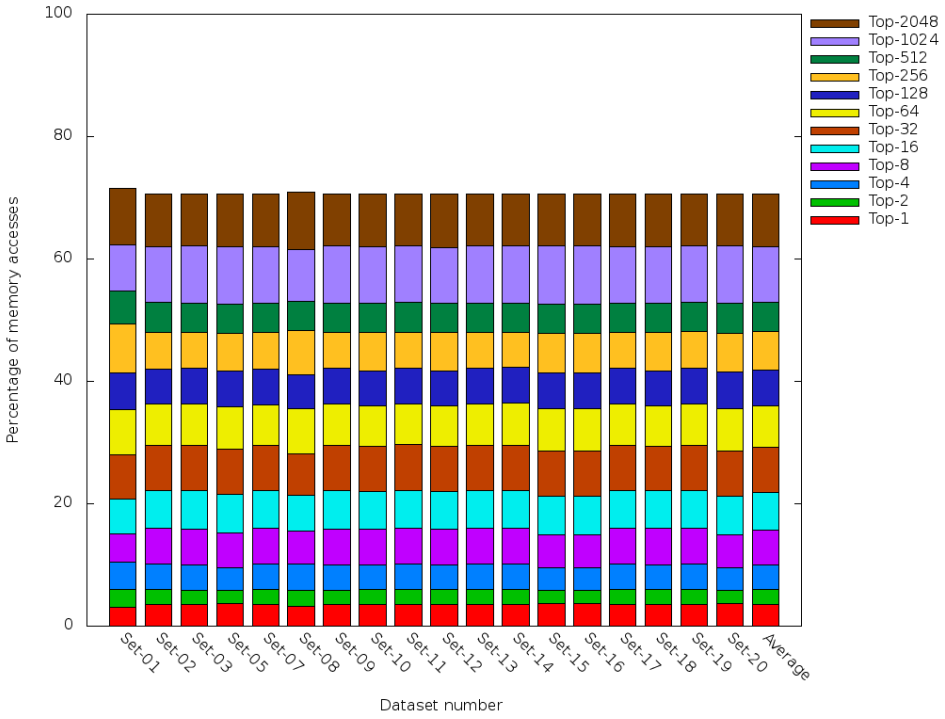

Figure 5.46: Security-rijndael-d. Percentage of all memory accesses accounted for by the top N most frequently transferred values at global level using Pin.

Security-sha. It can be observed from the results that:

- The amount of Value Reuse is increased on the x86 architecture compared to on LLVM.
- This is again likely to the x86 architecture having to store values to memory which would otherwise only be held in a register.
- The trend between datasets is similar on both the x86 architecture and LLVM. On both architectures, set 1 and set 8 have higher levels of Value Reuse than other datasets.

Telecom-adpcm-c. It can be observed from the results that:

- The amount of Value Reuse is decreased on the x86 architecture compared to on LLVM.
- Additionally, the trend across datasets differs on the x86 architecture and LLVM. Set 9, 10, 11 and 12 have relatively high Value Reuse on LLVM which is not reflected in the results on the x86 architecture.
- As the trends across datasets are quite different, it is difficult to suggest an amount by which the amount of Value Reuse on the x86 architecture is reduced compared to on LLVM.

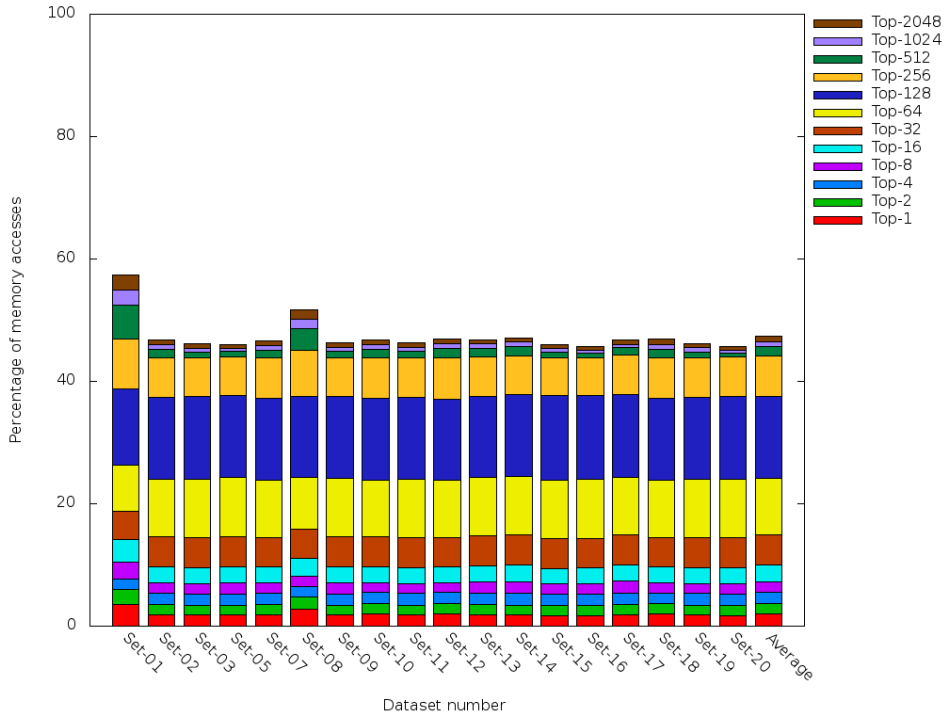

Figure 5.47: Security-sha. Percentage of all memory accesses accounted for by the top N most frequently transferred values at global level using Pin.

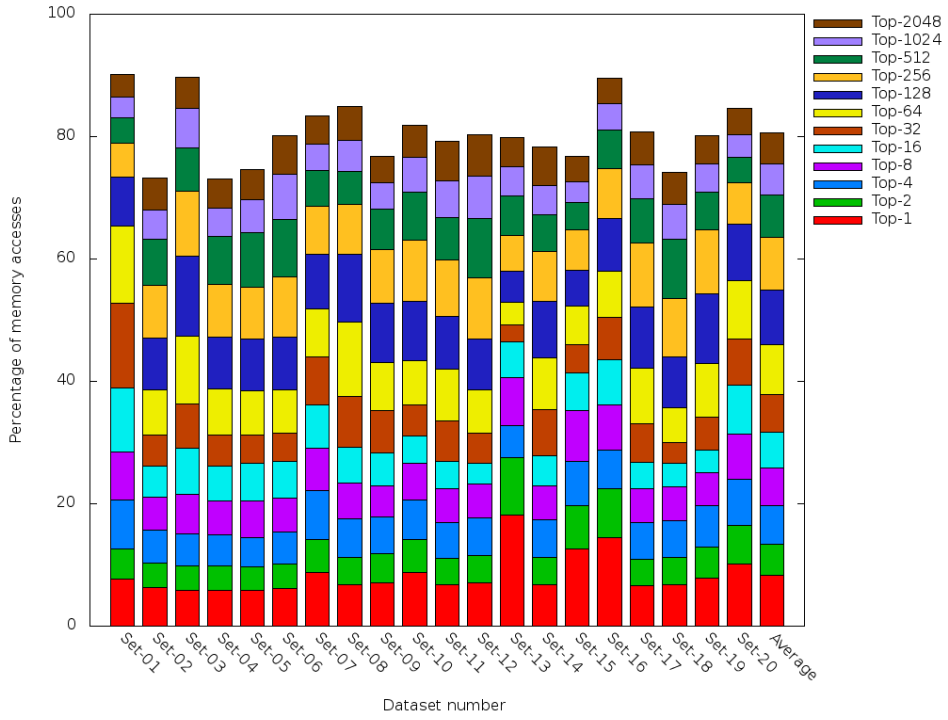

Figure 5.48: Telecom-adpcm-c. Percentage of all memory accesses accounted for by the top N most frequently transferred values at global level using Pin.

Telecom-adpcm-d. It can be observed from the results that:

- The amount of Value Reuse is slightly reduced on the x86 architecture compared to on LLVM.
- The percentage of all memory accesses involving smaller sets of frequent values is more decreased than the percentage of larger sets of frequent values. For example, the average percentage of all memory accesses involving the single most frequently transferred value is decreased by approximately 66%, whilst the average percentage involving the top 64 most frequent values is decreased by only 25%.
- Unlike *telecom-adpcm-c*, the trend across datasets is similar on the x86 architecture and LLVM. Sets 9, 10, 11 and 12 fit in with the trend for this benchmark.

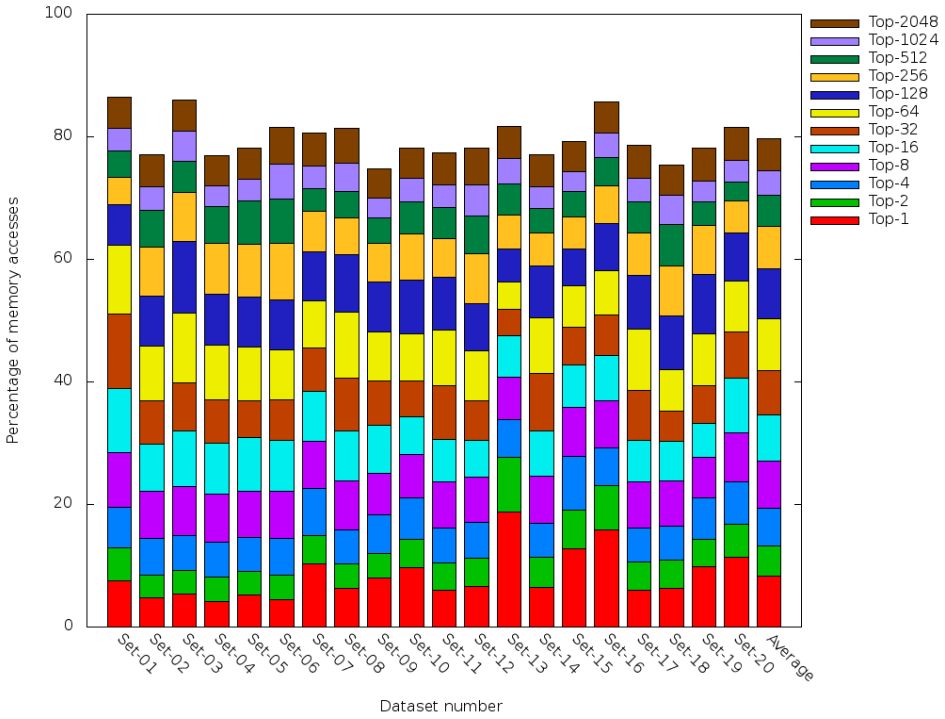

Figure 5.49: Telecom-adpcm-d. Percentage of all memory accesses accounted for by the top N most frequently transferred values at global level using Pin.

Telecom-crc32. It can be observed from the results that:

- The amount of Value Reuse for small sets of frequent values is greatly increased on the x86 architecture compared to on LLVM. On average almost 70% of all memory accesses involve the top 16 most frequent values on the x86 architecture, compared to less than 10% on LLVM.
- However, on LLVM almost all memory accesses involve one of 256 values. On the x86 architecture, there are more than 2048 distinct values transferred. This can be observed because the Top-2048 bar on the graph does not reach 100%.

A Comparison Across all Benchmarks to LLVM Value Profile Data

As on LLVM, there is a high level of Value Reuse in Memory Accesses on the x86. However, the amount is slightly lower than on LLVM. For example, the top 32 most frequently transferred values on average account for around 40% of all memory accesses, compared to around 50% on LLVM. As with Value Reuse in Instruction executions, the amount of Value Reuse may be higher or lower on the x86 than on LLVM depending on which benchmark is examined. For example, Security-sha has higher levels of Value Reuse on the x86 than on LLVM, whereas Automotive-susan-c has lower levels of Value Reuse on the x86 than on LLVM. This again does not support Hypothesis 7.

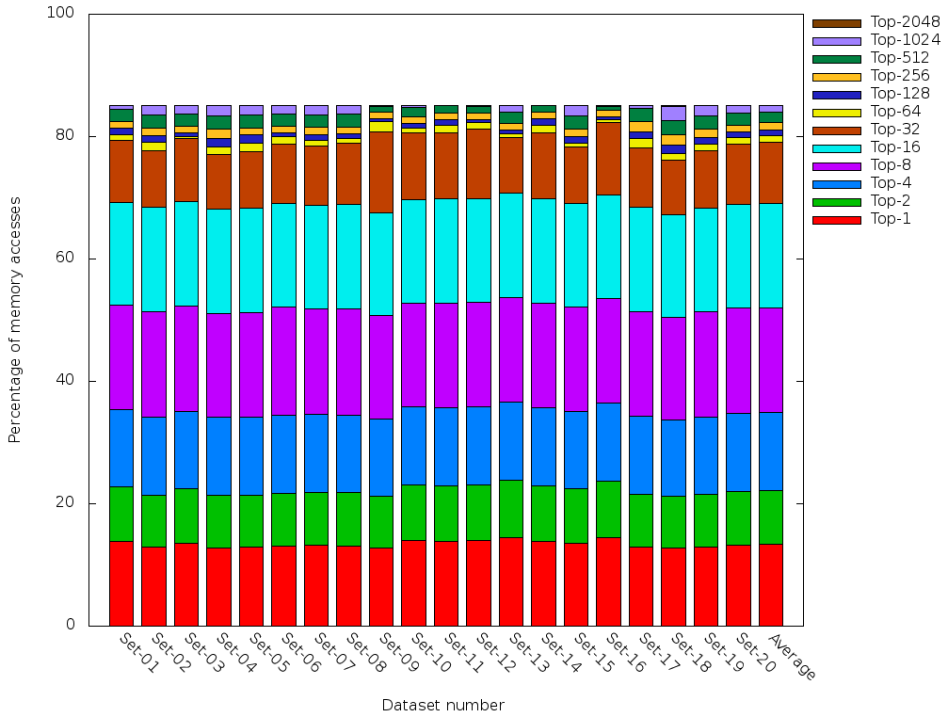

Figure 5.50: Telecom-crc32. Percentage of all memory accesses accounted for by the top N most frequently transferred values at global level using Pin.

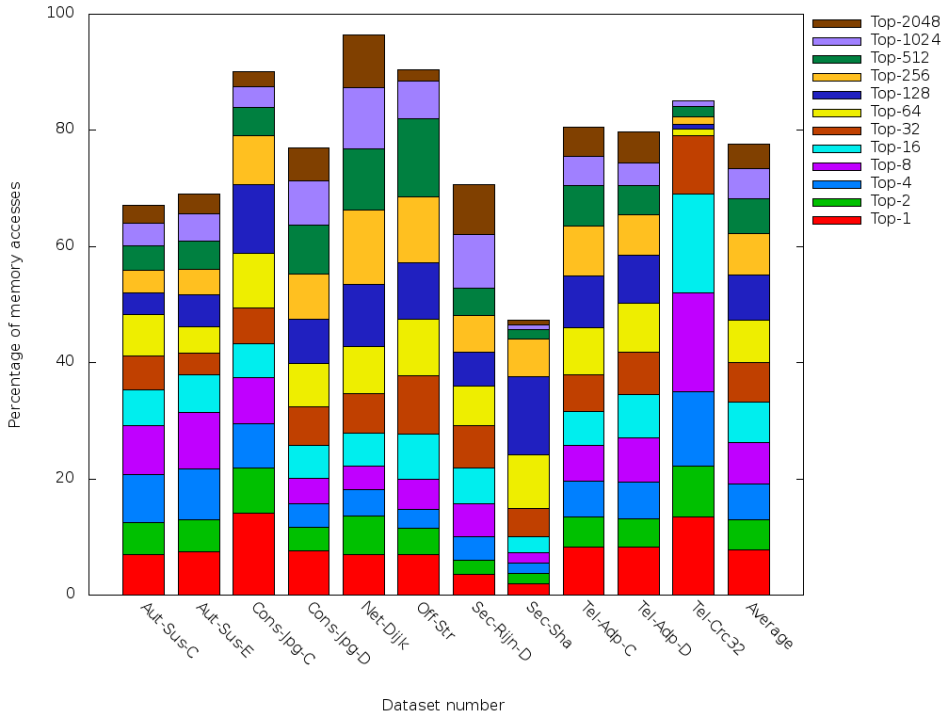

Figure 5.51: Comparison of the percentage of all memory accesses accounted for by the top N most frequently transferred values across all benchmarks at global level using Pin.

5.2.3 Local-level Memory Access Value Profiling

As on LLVM, there is a lower level of Value Reuse in Memory Accesses as local level. The top 32 most frequent values only account for around 30% of all memory accesses at the local level, compared to around 40% at the global level. Again this is in support of Hypothesis 3.

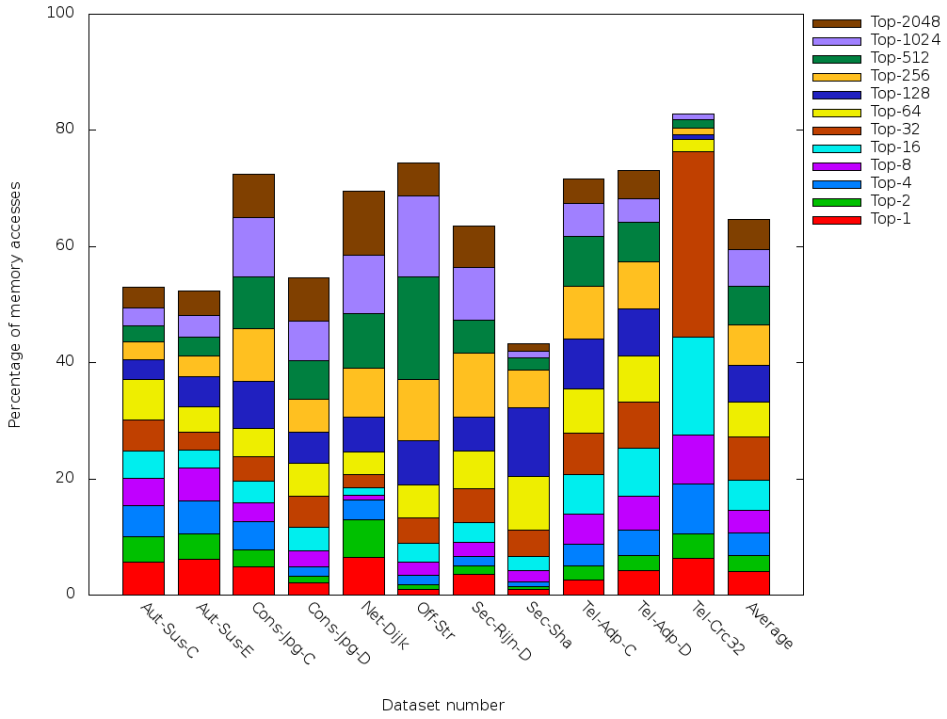

Figure 5.52: Comparison of the percentage of all memory accesses accounted for by the top N most frequently transferred values across all benchmarks at local level using Pin.

5.3 Conclusion to the Results and Analysis

Several of the proposed hypotheses have been examined in this chapter. The outcomes of these results with regard to the hypotheses are as follows:

Hypothesis 1. It has been seen that this hypothesis is well-supported by the results, and that Value Reuse is prevalent throughout the execution of most programs.

Hypothesis 2. It has not been possible to test this hypothesis.

Hypothesis 3. It has been shown that this hypothesis is also well-supported by the results. The amount of Value Reuse in Memory Accesses is greater at the global level than at the local level.

Hypothesis 4. It has been seen that this hypothesis is supported by the results. There is a correlation between the amount of Value Reuse in Memory Accesses and the amount of Value Reuse in Instruction Executions.

Hypothesis 7. It has been seen that this hypothesis is not correct. Value Profile data recorded on LLVM is not generally representative of Value Profile data recorded on the x86 architecture. This could in part be due to the differences between the way that LLVM and Pin record Value Profile data. For example, Value Profiling on LLVM does not include profiling of executed library code, whereas Value Profiling on Pin does include library code. For a more complete comparison of Value Profiling using LLVM and Pin, see Chapter 9.

As it has been shown that Value Reuse in Instruction Executions is prevalent throughout the execution of most programs, a scheme could be developed to exploit this Value Reuse. Such a scheme is presented in Chapter 6. Additionally, the Value Profiling of Memory Accesses may be further refined by simulating more accurately the values which are transferred across the data bus. A method to produce a more representative Value Profile of Memory Accesses is presented in Chapter 7.

Chapter 6

Exploiting Value Reuse in Instruction Executions - A Value Reuse Cache

6.1 Background

(Sodani & Sohi, 1997) implemented a scheme which exploits Value Reuse in Instruction Executions in order to increase performance by decreasing the total execution time of a program. Hypothesis 5 in this report states that "It is possible to exploit value reuse in instruction executions to improve performance in terms of decreasing execution time...", which can be confirmed by implementing a *Value Reuse Cache* (termed an *Instruction Reuse Buffer* in (Sodani & Sohi, 1997)). Additionally, Hypothesis 2 is untested, as it was not possible to gather Local-level Value Profile data of Instruction executions either on LLVM or on the x86 architecture. This hypothesis can be tested by implementing and testing two Value Reuse Caches: one which exploits Global-level Value Reuse in Instruction Executions, and another which exploits Local-level Value Reuse in Instruction Executions. If the Global-level Value Reuse Cache is able to exploit Value Reuse more successfully than the Local-level Value Reuse Cache, then this would serve to support Hypothesis 2.

The purpose of the Value Reuse Cache is to reduce the total amount of instruction executions by allowing the bypass of certain instruction executions. Instruction executions which reuse the same inputs as earlier computations may be bypassed, as the results of these computations were stored in the Value Reuse Cache at the point that they were first executed. When an instruction to be bypassed is encountered, the result of the computation is retrieved from the cache. The instruction is discarded and the processor can continue with the next instruction to be executed.

A Value Reuse Cache must be implemented in hardware, on the processor, for it to provide any benefit. The overhead of a software scheme to reuse computations at this small granularity will have an overhead many times that of the potential benefit of this scheme. It should be noted that at a much larger granularity, a software scheme can improve performance by exploiting repeated computations - an example of this is seen in (Kumar, 2003). In this chapter, a Pin Tool will be presented which simulates a Value Reuse Cache on the x86 architecture, to determine the potential performance benefits. It is far beyond the scope of this project to implement the Value Reuse Cache in hardware.

6.2 Design of the Value Reuse Cache

The Value Reuse cache will hold a finite number of results of Instruction Executions. (Sodani & Sohi, 1997) tested Instruction Reuse Buffers which held 32, 128 or 1024 entries. This implementation will consider the following sizes of Value Reuse Cache:

- 32-entry
- 128-entry

- 512-entry
- 2048-entry

These sizes have been chosen as each cache is four times larger than the next smallest cache. This will make it easier to examine a trend in the hit rates of the caches. Ideally more caches would have been implemented, so that the hit rate can be compared to doubling the size of the cache. However, it would have taken twice as long to produce the results in this case, as twice as many caches would have to be tested.

As the Value Reuse Cache is limited in size, there will frequently be times when an instruction is executed which is not already present in the Value Reuse Cache, and the cache is already full. At this point, a decision has to be made about which entry to evict (remove) from the cache to make space for the new entry. It is stated in (Handy, 1998) that the optimal replacement policy for a cache is generally a Least Recently Used (LRU) policy. This policy always chooses the entry in the cache which was accessed least recently for eviction. This policy is also straightforward to implement in software. Therefore, an LRU eviction policy has been chosen for the cache.

Another major factor in the design of a cache is its *associativity*. Many caches implement a scheme for storage of entries which maps each item to be stored using a hashing function to specific areas in the cache. A cache which does not use a hashing function to map entries to specific locations, but rather allows an entry to be stored at any location in the cache, is termed *fully associative* (Handy, 1998). The Value Reuse Cache will be implemented as a fully associative cache, as this is the most straightforward cache to implement.

The computations which the cache will store, and will be termed *cacheable instructions*, will be the same as those profiled in Instruction-level Value Profiling on the x86 architecture. These instructions are listed in Table 4.3.

This implementation of the Value Reuse Cache will not modify the execution of the programs that it is tested with. Instead, the contents of the cache will merely be tracked in order to determine whether there is a hit or miss for each cacheable instruction execution. Statistics will be output at the end of the execution. Additionally, the total number of instructions executed will also be output so that the overall hit rate can be computed to assess the effectiveness of the cache.

6.3 Implementation of the Value Reuse Cache

A UML diagram of the classes to implement the Value Reuse Cache is shown in Figure 6.1.

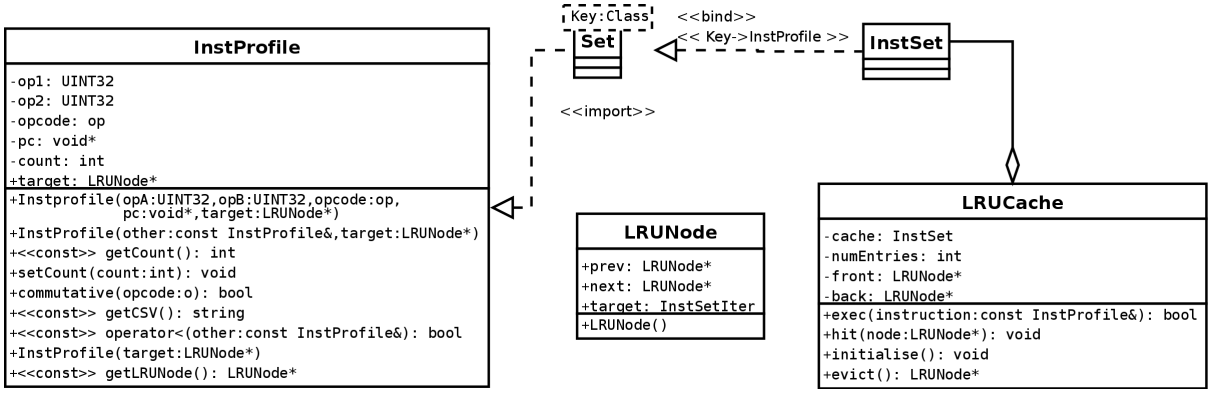

Figure 6.1: Classes involved in the implementation of the Value Reuse Cache.

The InstProfile object used in Value Profiling of Instruction executions is used to store the data regarding each individual entry in the cache. The only modification to this class is the addition of an additional member variable, `target`, which is of type `LRUNode*`. The LRUNode class is intended to implement a doubly-linked list, called the *LRU queue*. The LRU queue is in order of the least recently used to most recently used. This is to implement the LRU eviction policy - when a node must be evicted, the `LRUNode` at the front of the queue is least recently used, and can be chosen for eviction. When an

entry is accessed, the LRUNode corresponding to that entry is moved to the back of the queue, as it has been most recently accessed. As entries are moved to the back of the queue over time, nodes which are not accessed will eventually work their way towards the front of the queue.

The LRUCache class implements the cache. An STL `set` of InstProfile objects is used to hold all the entries in the cache. The `numEntries` variable stores the number of entries which the cache stores - it is ensured that more than `numEntries` are never present in the cache at any one time. `front` and `back` are pointers to the front and back of the LRU queue. The `exec()` method takes an InstProfile object storing the details of the instruction to be executed. This method returns `true` on the event of a cache hit, and `false` otherwise. The `hit()` method is used by the `exec()` method in the event of a cache hit. The specified `LRUNode` is moved to the back of the queue by this method. The `initialise()` method is called at program start-up, to allow the cache to fill itself with dummy entries. Finally the `evict()` method removes the `LRUNode` at the front of the LRU queue and returns a pointer to it.

The queue of LRU nodes is implemented separately from the InstProfile object due to the constraints placed on objects inserted into an STL `set` - specifically, iterators pointing to objects in a `set` cannot call non-`const` member functions. If the LRU queue were implemented in the InstProfile objects, the pointers to the next and previous elements in the queue would be members of the InstProfile class and would need to be updated each time a cache hit occurs. A function which modifies any member variable of an object cannot be a `const` function. Therefore, it would not be possible for the state of the queue to be changed once it had initially been set.

The solution to this is for each InstProfile object to store a pointer to an LRUNode, which is set before the InstProfile is inserted into the `set`. The `LRUNode` which the InstProfile object points to also has a pointer back to the corresponding InstProfile object. Therefore, when a cache hit occurs, the pointer to the LRUNode from the InstProfile can be followed, and appropriate changes made to the LRUNode object. When an entry is to be evicted, the pointer from the LRUNode at the front of the queue to the corresponding InstProfile object is used to determine which InstProfile to remove from the `set`.

6.4 Testing the Value Reuse Cache

The Test Cases (see Appendix B) were executed with an 8-entry Global-level Value Reuse Cache and an 8-entry Local-level Value Reuse Cache to test the functionality of the caches. Additionally, the smallest possible size of Global-level Value Reuse Cache (2-entry) was tested to ensure correctness of the eviction policy of the cache. The "Expected Result" column contains the expected output statistics at the end of the execution.

Table 6.1: Testing of Global-level 8-entry Value Reuse Cache

Test Case	Expected result	Pass
1	Cacheable instructions: 4 Cache hits: 0 Cache misses: 4	✓
2	Cacheable instructions: 5 Cache hits: 1 Cache misses: 4	✓
3	Cacheable instructions: 5 Cache hits: 0 Cache misses: 5	✓
4	Cacheable instructions: 13 Cache hits: 0 Cache misses: 13	✓
5	Cacheable instructions: 4 Cache hits: 0 Cache misses: 4	✓
6	Cacheable instructions: 3 Cache hits: 0 Cache misses: 3	✓

7	Cacheable instructions: 4 Cache hits: 0 Cache misses: 4	✓
8	Cacheable instructions: 3 Cache hits: 0 Cache misses: 3	✓
9	Cacheable instructions: 3 Cache hits: 0 Cache misses: 3	✓
10	Cacheable instructions: 3 Cache hits: 0 Cache misses: 3	✓
11	Cacheable instructions: 7 Cache hits: 1 Cache misses: 6	✓
12	Cacheable instructions: 4 Cache hits: 0 Cache misses: 4	✓
13	Cacheable instructions: 5 Cache hits: 1 Cache misses: 4	✓
14	Cacheable instructions: 50003 Cache hits: 30000 Cache misses: 20003	✓
15	Cacheable instructions: 23 Cache hits: 12 Cache misses: 11	✓
16	Cacheable instructions: 93 Cache hits: 43 Cache misses: 50	✓
17	Cacheable instructions: 5 Cache hits: 1 Cache misses: 4	✓
18	Cacheable instructions: 3 Cache hits: 0 Cache misses: 3	✓
19	Cacheable instructions: 4 Cache hits: 0 Cache misses: 4	✓
20	Cacheable instructions: 3 Cache hits: 0 Cache misses: 3	✓
21	Cacheable instructions: 4 Cache hits: 0 Cache misses: 4	✓

Table 6.2: Testing of Local-level 8-entry Value Reuse Cache

Test Case	Expected result	Pass
1	Cacheable instructions: 4 Cache hits: 0 Cache misses: 4	✓
2	Cacheable instructions: 5 Cache hits: 0	✓

	Cache misses: 5	
3	Cacheable instructions: 5 Cache hits: 0 Cache misses: 5	✓
4	Cacheable instructions: 13 Cache hits: 0 Cache misses: 13	✓
5	Cacheable instructions: 4 Cache hits: 0 Cache misses: 4	✓
6	Cacheable instructions: 3 Cache hits: 0 Cache misses: 3	✓
7	Cacheable instructions: 4 Cache hits: 0 Cache misses: 4	✓
8	Cacheable instructions: 3 Cache hits: 0 Cache misses: 3	✓
9	Cacheable instructions: 3 Cache hits: 0 Cache misses: 3	✓
10	Cacheable instructions: 3 Cache hits: 0 Cache misses: 3	✓
11	Cacheable instructions: 7 Cache hits: 1 Cache misses: 6	✓
12	Cacheable instructions: 4 Cache hits: 0 Cache misses: 4	✓
13	Cacheable instructions: 5 Cache hits: 0 Cache misses: 5	✓
14	Cacheable instructions: 50003 Cache hits: 29998 Cache misses: 20005	✓
15	Cacheable instructions: 23 Cache hits: 12 Cache misses: 11	✓
16	Cacheable instructions: 93 Cache hits: 39 Cache misses: 54	✓
17	Cacheable instructions: 5 Cache hits: 0 Cache misses: 5	✓
18	Cacheable instructions: 3 Cache hits: 0 Cache misses: 3	✓
19	Cacheable instructions: 4 Cache hits: 0 Cache misses: 4	✓
20	Cacheable instructions: 3 Cache hits: 0 Cache misses: 3	✓

21	Cacheable instructions: 4 Cache hits: 0 Cache misses: 4	✓

Table 6.3: Testing of Global-level 2-entry Value Reuse Cache

Test Case	Expected result	Pass
1	Cacheable instructions: 4 Cache hits: 0 Cache misses: 4	✓
2	Cacheable instructions: 5 Cache hits: 1 Cache misses: 4	✓
3	Cacheable instructions: 5 Cache hits: 0 Cache misses: 5	✓
4	Cacheable instructions: 13 Cache hits: 0 Cache misses: 13	✓
5	Cacheable instructions: 4 Cache hits: 0 Cache misses: 4	✓
6	Cacheable instructions: 3 Cache hits: 0 Cache misses: 3	✓
7	Cacheable instructions: 4 Cache hits: 0 Cache misses: 4	✓
8	Cacheable instructions: 3 Cache hits: 0 Cache misses: 3	✓
9	Cacheable instructions: 3 Cache hits: 0 Cache misses: 3	✓
10	Cacheable instructions: 3 Cache hits: 0 Cache misses: 3	✓
11	Cacheable instructions: 7 Cache hits: 1 Cache misses: 6	✓
12	Cacheable instructions: 4 Cache hits: 0 Cache misses: 4	✓
13	Cacheable instructions: 5 Cache hits: 1 Cache misses: 4	✓
14	Cacheable instructions: 50003 Cache hits: 10002 Cache misses: 40001	✓
15	Cacheable instructions: 23 Cache hits: 0 Cache misses: 23	✓
16	Cacheable instructions: 93 Cache hits: 1	✓

	Cache misses: 92	
17	Cacheable instructions: 5 Cache hits: 1 Cache misses: 4	✓
18	Cacheable instructions: 3 Cache hits: 0 Cache misses: 3	✓
19	Cacheable instructions: 4 Cache hits: 0 Cache misses: 4	✓
20	Cacheable instructions: 3 Cache hits: 0 Cache misses: 3	✓
21	Cacheable instructions: 4 Cache hits: 0 Cache misses: 4	✓

6.5 Results & Effects of the Value Reuse Cache

When considering the results of the Value Reuse Cache, the cost of each size of Value Reuse Cache is considered against its performance benefit. The cost of implementing a Value Reuse cache is based on its size - the larger the cache, the more space an implementation would require on the die of a processor. Additionally, as the number of entries increases, the complexity of managing these entries will increase. The performance benefit is measured in terms of the hit rate of the cache. A higher hit rate is considered to provide better performance. As the Value Reuse Caches are simulated, it is only possible to speculate on the cost of a given size of Value Reuse Cache. Therefore, the cache size which provides the best cost/performance balance is only estimated.

6.5.1 Global-Level Value Reuse Cache

Automotive-susan-c. It can be observed from the results that:

- A 32-entry Value Reuse Cache performs poorly compared to the other three implemented sizes of cache, as it has a much lower hit rate across all datasets.
- A 128-entry Value Reuse Cache performs almost as well as both the larger caches across all datasets.
- It can be argued at a 128-entry cache is the optimal size for this benchmark to balance the resources required to implement the cache against the benefits it provides. Implementing a smaller cache greatly reduces the benefit of the cache, and implementing a larger cache provides little additional benefit.
- The larger caches have a hit rate of just under 20% - therefore, just under 20% of all instruction executions could be bypassed with the implementation of these caches.

Automotive-susan-e. It can be observed from the results that:

- As with *automotive-susan-c*, the 32-entry cache performed poorly in comparison to the other caches, and the larger caches provide little benefit over the 128-entry cache.
- Again it can be argued that the 128-entry cache provides the best cost/performance balance.
- There is a trend in the hit rates for the same dataset across this benchmark and Automotive-susan-c. For example, the hit rates for set 13 were lower than the average for both benchmarks. This is consistent with the benchmarks sharing some code and performing similar functions.
- The average hit rates were slightly lower than for *automotive-susan-c*. However, over 15% of all instruction executions could still be bypassed on average using a 128-entry Value Reuse Cache.

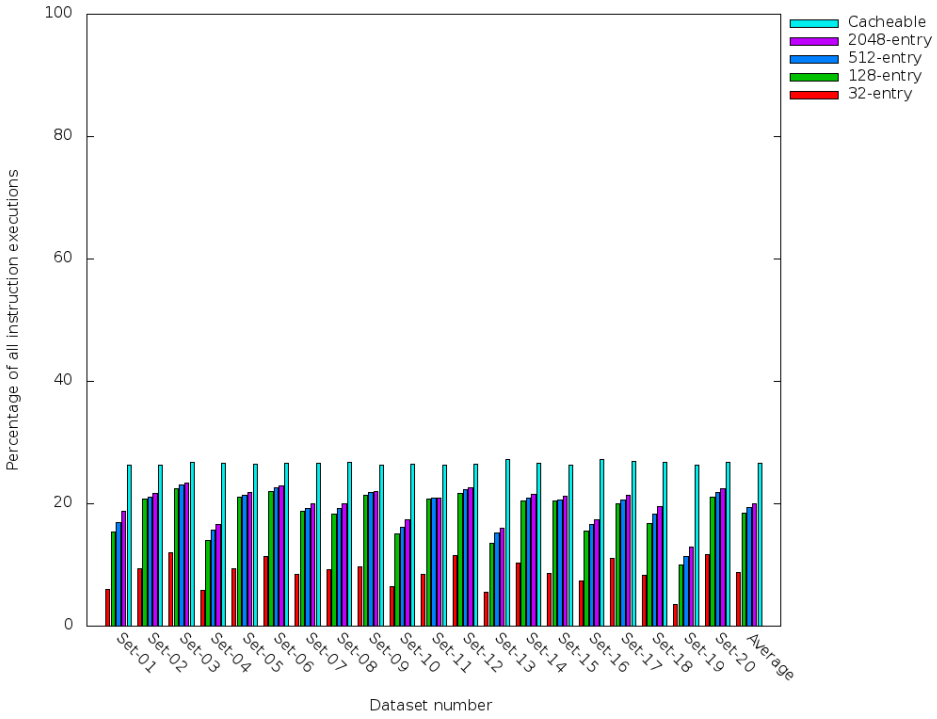

Figure 6.2: Automotive-susan-c. Cache hit rate for specified sizes of Global-Level Value Reuse Cache, and total percentage of cacheable instructions.

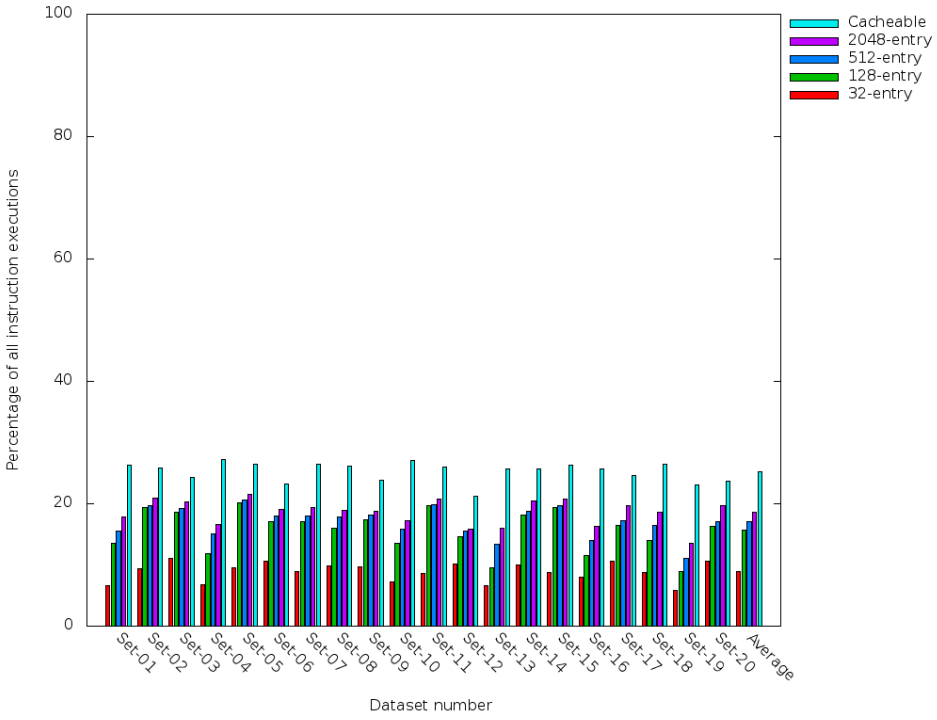

Figure 6.3: Automotive-susan-e. Cache hit rate for specified sizes of Global-Level Value Reuse Cache, and total percentage of cacheable instructions.

Consumer-jpeg-c. It can be observed from the results that:

- Across all instruction executions, fewer than 20% of all instruction executions were of cacheable instructions on average. Therefore, the average hit rate will never be able to reach 20%, even for a "perfect" cache which has a 100% hit rate.
- There is less difference between the 32- and 128-entry Value Reuse Caches than seen in the previous two benchmarks.
- The additional benefit over the 512- and 2048-entry caches over the 128-entry cache is greater than for the previous two benchmarks.
- It is difficult to make an obvious choice about which cache provides the best cost/performance balance without considering any additional factors.
- On average, the 128-entry cache will allow the bypass of just under 10% of all instruction executions for this benchmark.

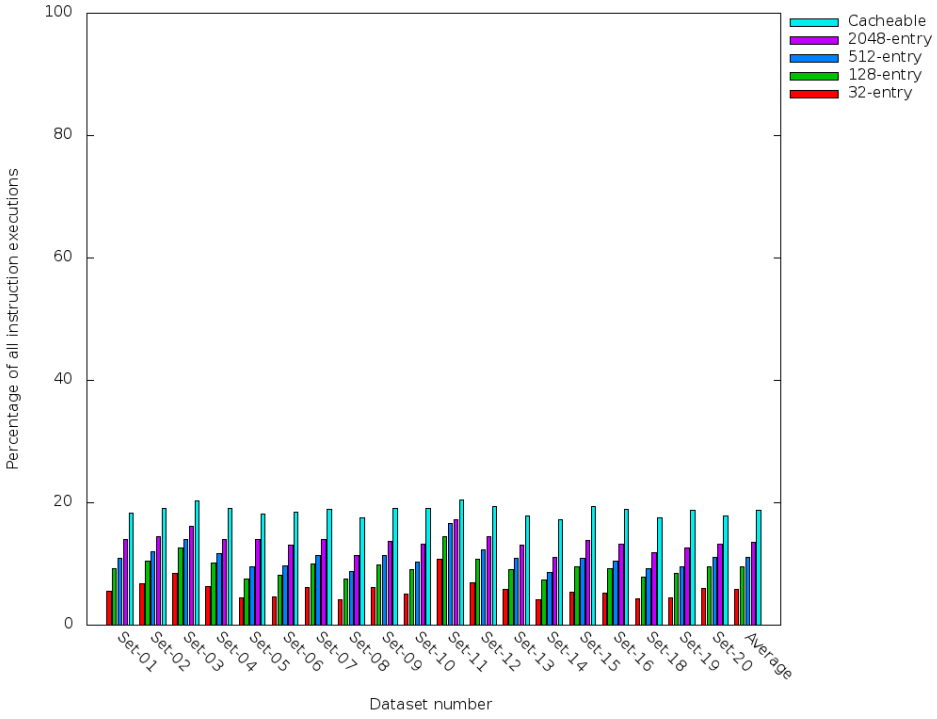

Figure 6.4: Consumer-Jpeg-C. Cache hit rate for specified sizes of Global-Level Value Reuse Cache, and total percentage of cacheable instructions.

Consumer-jpeg-d. It can be observed from the results that:

- For this benchmark, on average there is a relatively large (over 25%) percentage of cacheable instructions.
- However, this large percentage of cacheable instructions appears to be more difficult to exploit. The percentage of cacheable instructions can be seen to be much greater than the percentage of cache hits for all of the implementations of the Value Reuse Cache.
- Again it is difficult to see which cache provides the best cost/performance balance.
- On average, the 128-entry cache allows for the bypass of around 8% of all instruction executions.
- The large difference in the percentage of cacheable instructions between this benchmark and *consumer-jpeg-c* is again consistent with the JPEG encoding and decoding processes being quite different.

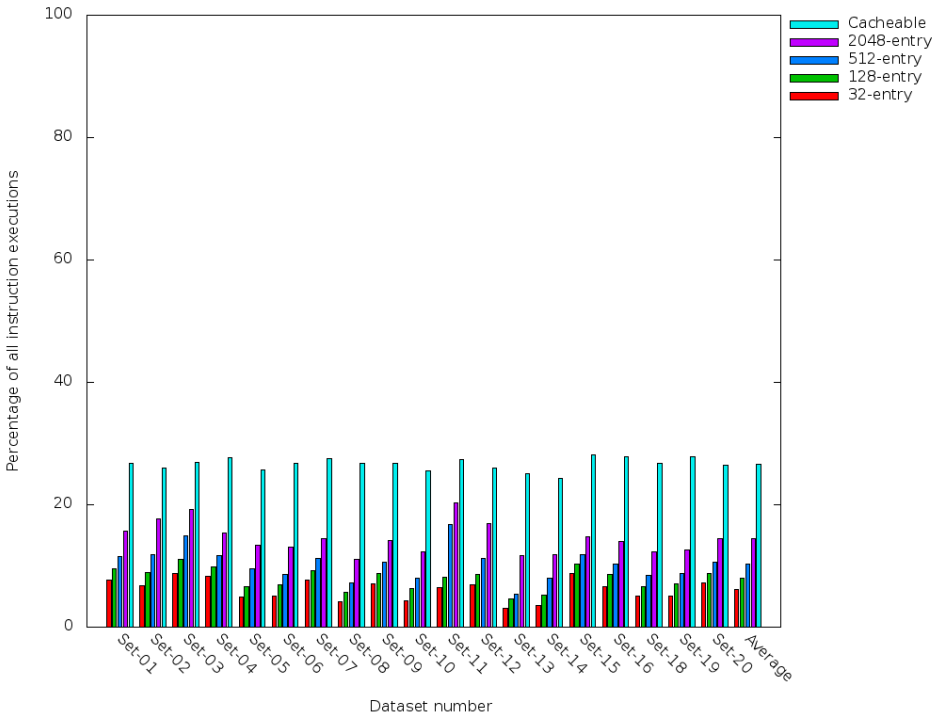

Figure 6.5: Consumer-Jpeg-D. Cache hit rate for specified sizes of Global-Level Value Reuse Cache, and total percentage of cacheable instructions.

Network-dijkstra. It can be observed from the results that:

- There is a very small fraction of cacheable instructions executed in this benchmark.
- For datasets 10 to 20, the 128- and 512-entry caches do not provide any additional benefit over the 32-entry cache. For datasets 1 to 9, the 512- and 2048-entry caches provide a large benefit over the 32- and 128-entry caches.
- Although there is only a small fraction of cacheable instructions executed, the 2048-entry Value Reuse Cache hits on almost every execution of a cacheable instruction. This is consistent with there being a set of around 2048 unique computations which account for a large fraction of all cacheable instruction executions.
- From these results it is again difficult to choose a cache which gives a better cost/performance balance over the others. However the 128- and 512-entry caches are most likely to provide a better cost/performance balance than the other two caches, because they both have a large advantage over smaller cache sizes for certain datasets.

Office-stringsearch. It can be observed from the results that:

- There is a modest percentage of all instruction executions which are cacheable instruction executions, at around 20%.
- The 32- and 128-entry Value Reuse Caches perform poorly in comparison to the 512- and 2048-entry Value Reuse Caches.
- The 2048-entry Value Reuse cache provides little additional benefit over the 512-entry Value Reuse Cache.
- The 512-entry Value Reuse Cache provides the best cost/performance balance for this benchmark.
- The 512-entry Value Reuse Cache allows for the bypass of just under 15% of all instruction executions on average for this benchmark.

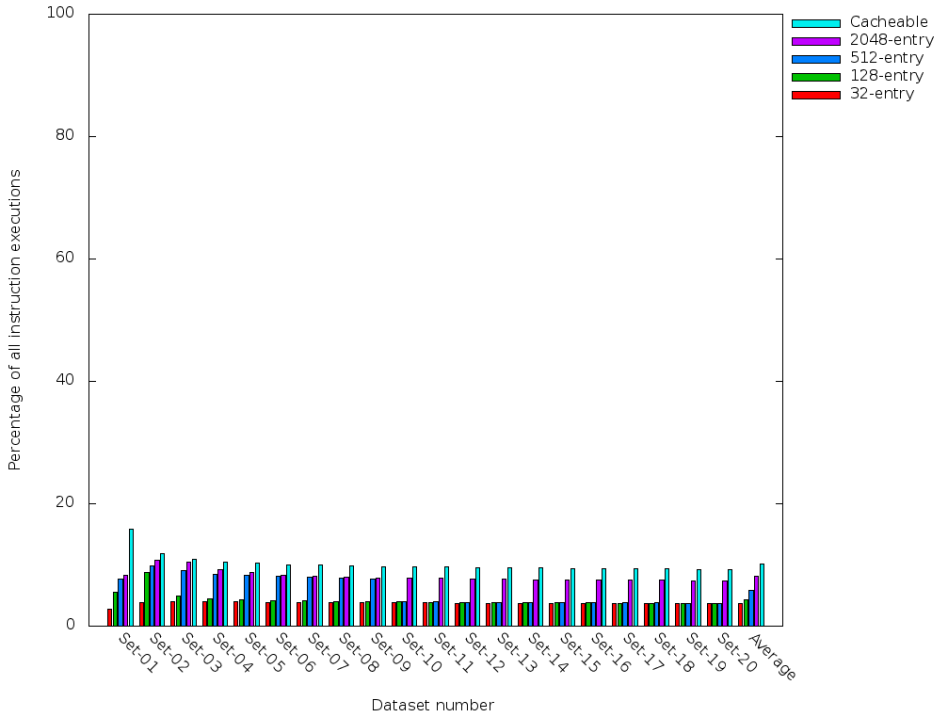

Figure 6.6: Network-Dijkstra. Cache hit rate for specified sizes of Global-Level Value Reuse Cache, and total percentage of cacheable instructions.

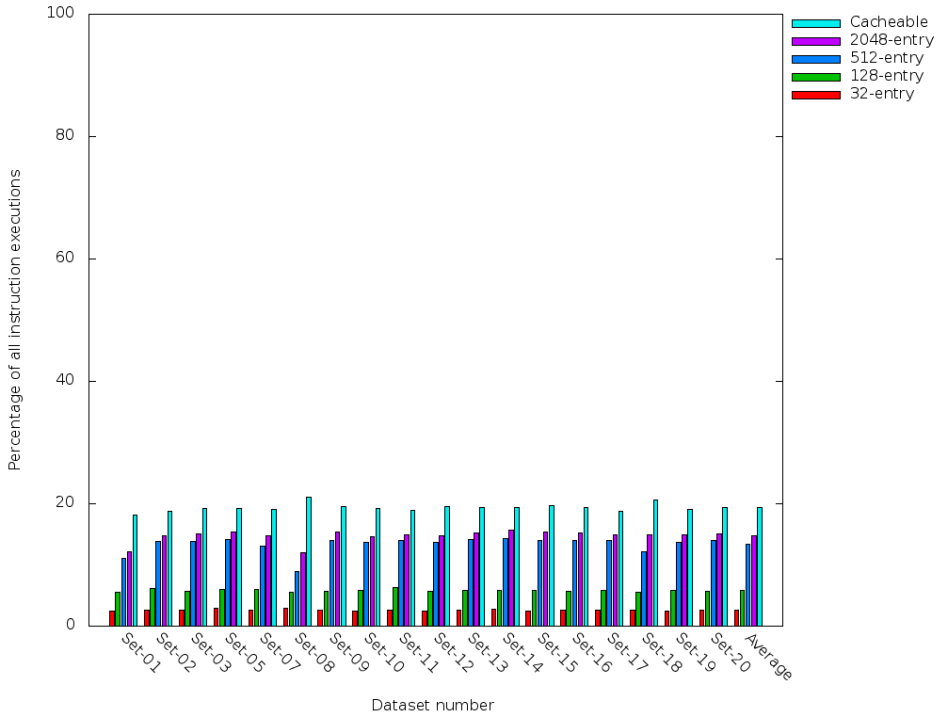

Figure 6.7: Office-Stringsearch. Cache hit rate for specified sizes of Global-Level Value Reuse Cache, and total percentage of cacheable instructions.

Security-rijndael-d. It can be observed from the results that:

- There is a large fraction of instruction executions which are of cacheable instructions.
- However, it is difficult for the Value Reuse Cache to exploit any Value Reuse in this Benchmark. This is consistent with there being a low level of Value Reuse found in the Value Profiling of Instruction Executions and Memory Accesses discussed previously.
- It is difficult to suggest that the implementation of a Value Reuse Cache provides a significant benefit in this case - even the 2048-entry Value Reuse Cache only allows the bypass of around 5% of all instruction executions. The smaller caches provide even less benefit.

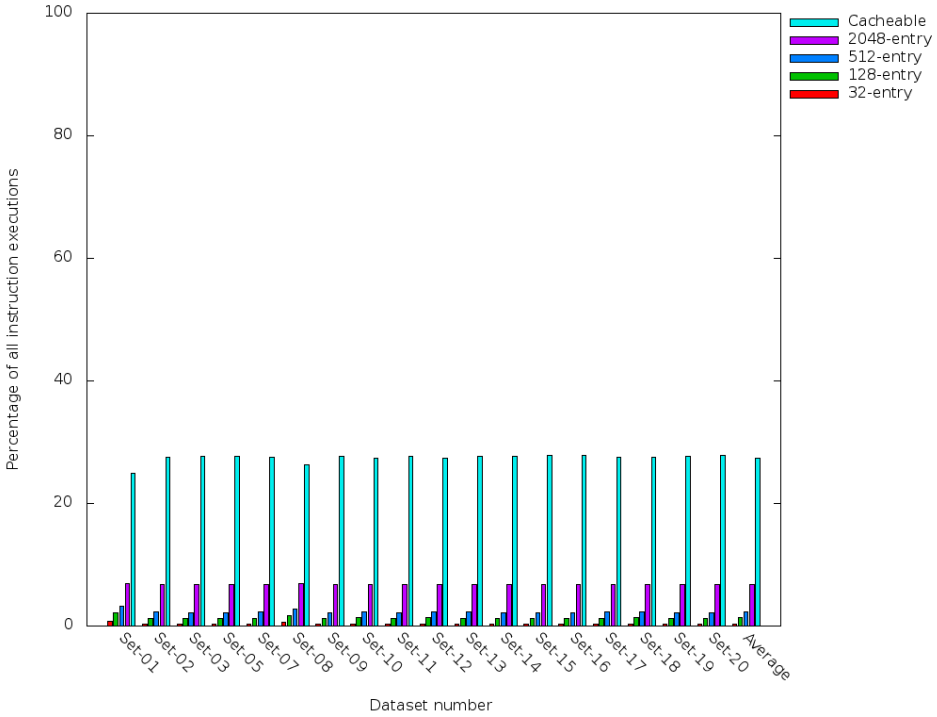

Figure 6.8: Security-Rijndael-D. Cache hit rate for specified sizes of Global-Level Value Reuse Cache, and total percentage of cacheable instructions.

Security-sha. It can be observed from the results that:

- As with Security-rijndael-d, there is a relatively large percentage of all instruction executions which are of cacheable instructions. Over 20% of all instruction executions are cacheable.
- Surprisingly for this benchmark, the 2048-entry Value Reuse Cache provides a reasonable hit rate. Over 10% of all instruction executions can be bypassed using a 2048-entry Value Reuse Cache.
- This result is surprising because Security-sha generally shows one of the lowest levels of Value Reuse across all benchmarks for all types of Value Profiling.
- However, the Value Reuse Cache is able to exploit values which are reused only a small number of times, but the occurrences of the reuse all occur temporally local to each other. These values which are reused only a small number of times would not normally show up on an Instruction Execution or Memory Access Value Profile, as these two profiling techniques consider the reuse of values throughout the lifetime of the program.
- It is clear the the 2048-entry Value Reuse Cache would provide the best cost/performance balance for this benchmark.

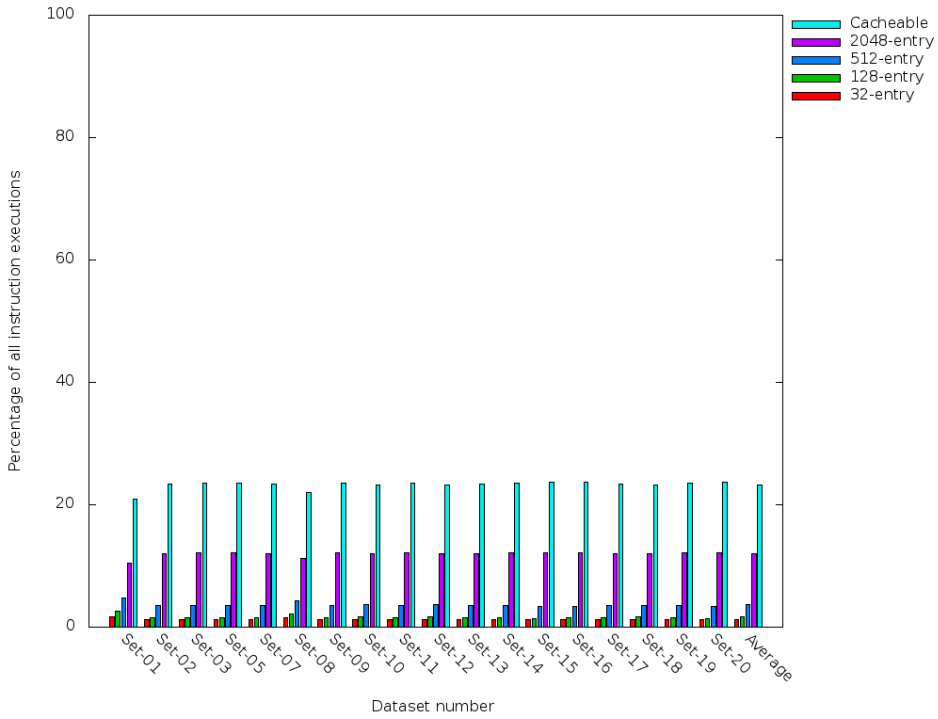

Figure 6.9: Security-Sha. Cache hit rate for specified sizes of Global-Level Value Reuse Cache, and total percentage of cacheable instructions.

Telecom-adpcm-c. It can be observed from the results that:

- There is a modest (around 16%) percentage of all instruction executions which are of cacheable instructions.
- All of the Value Reuse Caches exploit the Value Reuse in Instruction Executions in this benchmark to a small extent. The largest cache, a 2048-entry Value Reuse Cache allows for the bypass of around 8% of all instruction executions on average.
- In most cases, the 512-entry Value Reuse Cache has a hit rate almost as great as that of the 2048-entry Value Reuse Cache. Therefore, the 512-entry cache is a good candidate for being the cache which provides the best cost/performance balance.
- The level of exploitation of Value Reuse in Instruction Executions is lower than would be expected for this benchmark. This benchmark has shown a higher than average level of Value Reuse in Instruction Executions and Memory Accesses in the Value Profiling investigation discussed earlier. A reason for this poor exploitation could be that instruction values are reused temporally distantly from one another. This would cause the record of the first execution of an instruction and its input values to be evicted from the cache by the time it is re-executed.

Telecom-adpcm-d. It can be observed from the results that:

- There is a higher proportion of cacheable instruction executions in this benchmark than *telecom-adpcm-c*. This is surprising because in the earlier Value Profiling, the results for these two benchmarks had generally been very similar.
- Additionally, the Value Reuse Caches are able to exploit the Value Reuse in this benchmark successfully.
- The 512-entry Value Reuse Cache performs almost as well as the 2048-entry Value Reuse Cache across all datasets. Additionally the 512-entry cache has a much higher hit rate than the 32- and 128-entry caches across all datasets.

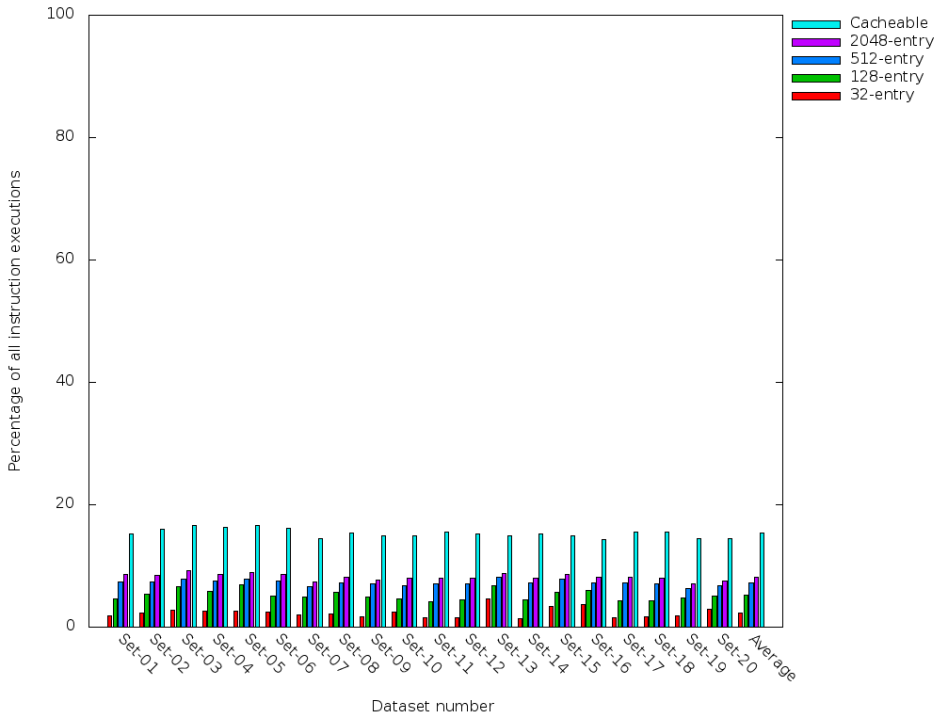

Figure 6.10: Telecom-Adpcm-C. Cache hit rate for specified sizes of Global-Level Value Reuse Cache, and total percentage of cacheable instructions.

- As a result, it is likely that a 512-entry Value Reuse Cache will provide the best cost/performance balance for this benchmark.
- The 512-entry cache allows the bypass of around 16% of all instruction executions.

Telecom-crc32. It can be observed from the results that:

- There is a reasonable (15%) percentage of all instruction executions which are of cacheable instructions.
- All of the caches are able to exploit exactly the same amount of Value Reuse. Just under 10% of all instruction executions can be bypassed with all of the caches.
- This is likely to be because the Telecom-crc32 benchmark reuses the same values in instruction executions several times within a short space of time (less than 32 cacheable instructions apart). Instruction values not reused in a short space of time are likely to not be reused until much later, or never again reused.
- These results are consistent with those found in the Value Profiling of Instruction executions, which suggested that no particular values were reused a large number of times, but that there were many values which were reused a small number of times.
- The 32-entry cache obviously provides the best cost/performance balance for this benchmark.

6.5.2 A comparison across all benchmarks

- There is less variation in the amount of Value Reuse which the Value Reuse Caches are able to exploit than there is in the amount of Value Reuse in Instruction Executions found across all the benchmarks. The Value Reuse present in the *security-sha*, *security-rijndael-d* and *telecom-crc32* benchmarks was even successfully exploited to some extent.
- As would be expected, the largest cache was able to exploit the most Value Reuse, and the smallest cache was able to provide the lowest amount of exploitation.

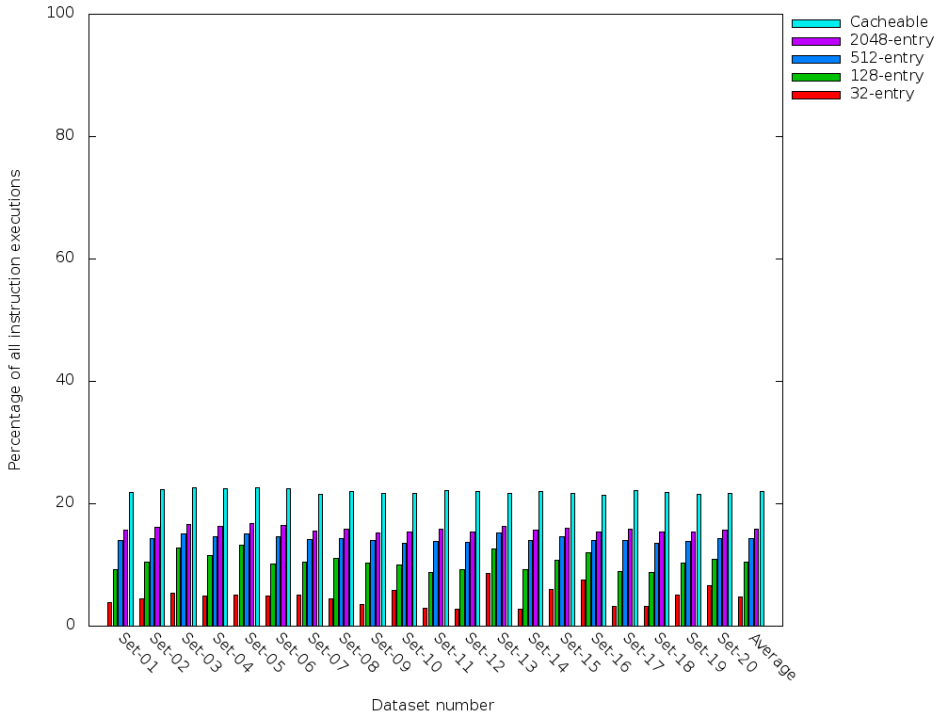

Figure 6.11: Telecom-Adpcm-D. Cache hit rate for specified sizes of Global-Level Value Reuse Cache, and total percentage of cacheable instructions.

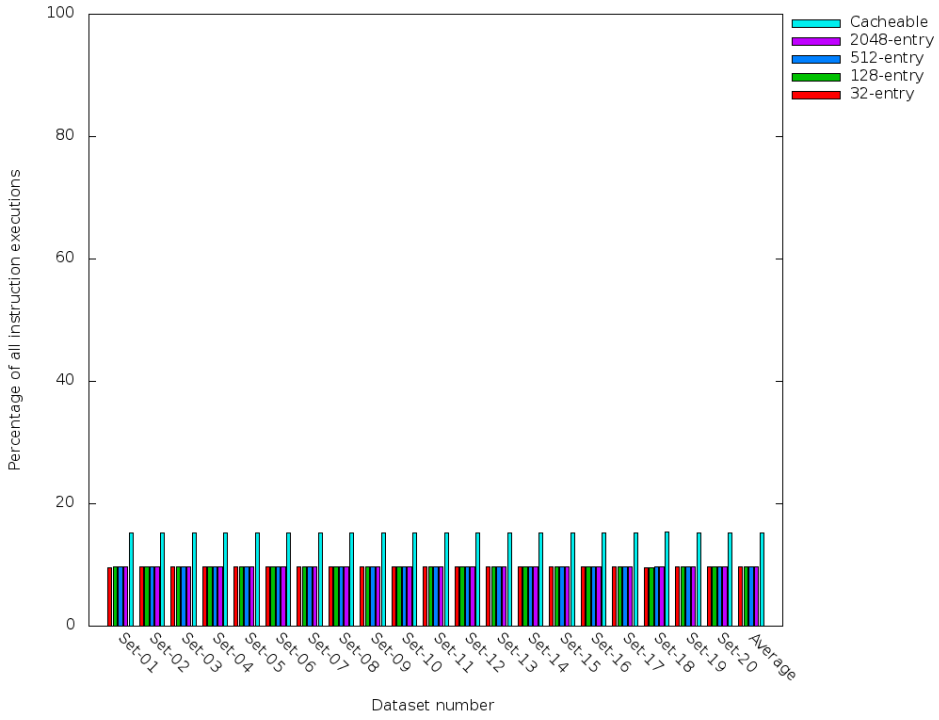

Figure 6.12: Telecom-Crc32. Cache hit rate for specified sizes of Global-Level Value Reuse Cache, and total percentage of cacheable instructions.

- There is no cache which clearly provides a better cost/performance balance than any other.
- Examining the average percentage of cache hits for each cache, it appears that quadrupling the number of entries in the cache provides a linear increase in the percentage of cache hits.
- This trend would not continue if larger cache sizes were used, such as 8192, 32768, etc. On average, the percentage of cacheable instructions which could be exploited is only just over 20%. Therefore the exploitation of Value Reuse in Instruction Executions could never allow for the bypass of more than 20% of all instruction executions on average.
- It is expected that as further increases in size of the Value Reuse Cache were made, the increase in hit rate compared to the previous size of Value Reuse Cache would diminish.

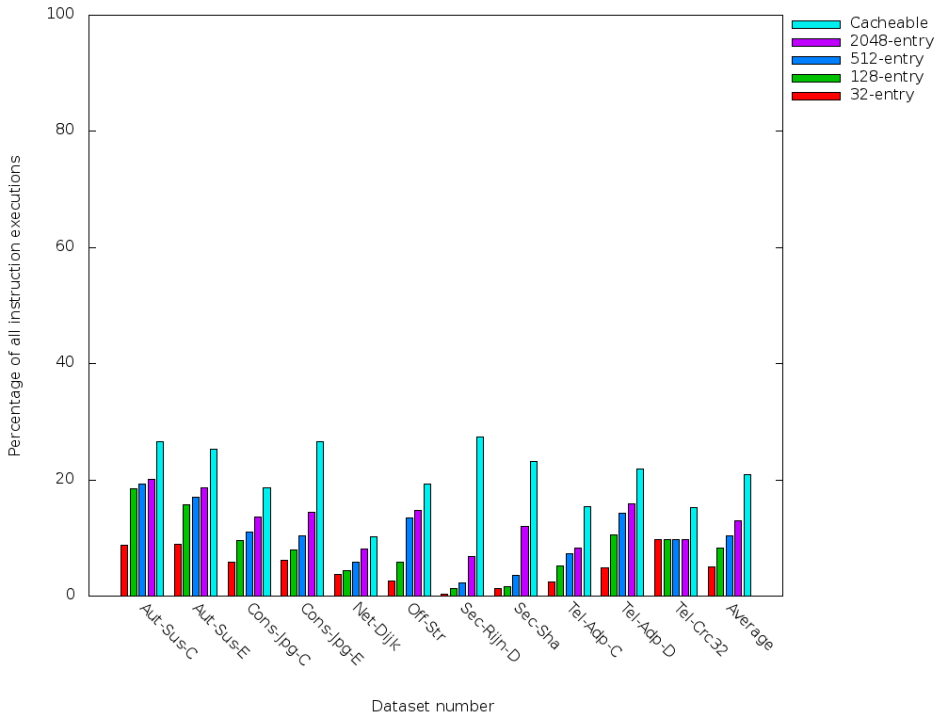

Figure 6.13: Comparison of the cache hit rate for specified sizes of Global-Level Value Reuse Cache and total percentage of cacheable instructions across all benchmarks.

6.5.3 Considering Only Cacheable Instruction Executions

Whilst the total hit rate within all instruction executions appears to be relatively low for this cache, the hit rates within the set of cacheable instructions are relatively high. Figure 6.14 shows the hit rates across all benchmarks for the Value Reuse Caches. These results are more encouraging - for example, the 2048 entry Value Reuse Cache had an average hit rate of over 60% of all cacheable instructions, and in better cases over 80%. An interesting result is that the larger caches provided good hit rates within cacheable instructions on the Security benchmarks, which have previously shown low levels of Value Reuse in all areas of Value Profiling. The 2048-entry Value Reuse cache managed a hit rate of over 50% within the cacheable instructions for Security-sha.

It is possible that implementation of a Value Reuse Cache on another architecture (perhaps one with fewer instruction opcodes) would yield higher levels of cacheable instructions - as there appear to be high hit rates within the cacheable instructions, this would give higher overall hit rates. Alternatively, if the Value Reuse Cache supported a larger subset of instruction opcodes, then it is quite possible that the overall hit rates would be higher. However, as the x86 architecture has literally hundreds of opcodes, it would require a large effort to design a Value Reuse Cache capable of supporting a majority of all opcodes.

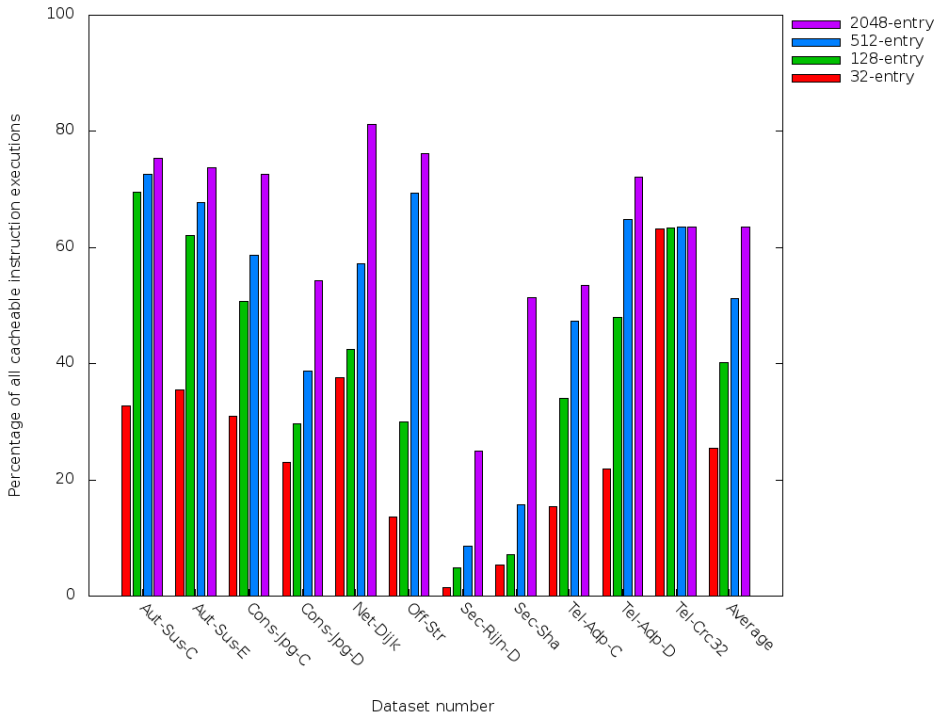

Figure 6.14: Comparison of the cache hit rate for specified sizes of Global-Level Value Reuse Cache within only cacheable instructions across all benchmarks.

6.5.4 Local-level Value Reuse Cache

Automotive-susan-c. It can be observed from the results that:

- The Local-level Value Reuse Cache is less able to exploit Value Reuse than the Global-level Value Reuse Cache. This can be seen as the cache hit rate is decreased compared to the Global-level Value Reuse Cache for all cache sizes for all datasets.
- However, the percentage of cacheable instructions remain constant, as exactly the same instructions were executed for each run of the benchmark as when the Global-level Value Reuse Cache was tested.

Automotive-susan-e. It can be observed from the results that:

- Again the amount of exploitation of Value Reuse (percentage of cache hits) is decreased when compared to the Global-level Value Reuse Cache.

Consumer-jpeg-c. It can be observed from the results that:

- Again the amount of exploitation of Value Reuse is decreased when compared to the Global-level Value Reuse Cache.

Consumer-jpeg-d. It can be observed from the results that:

- There is a slight decrease in the exploitation of Value Reuse when compared to the Global-level Value Reuse Cache.
- The 32-entry cache has a more decreased hit rate than the other caches in this case.
- It is likely for this benchmark that the main source of Value Reuse is the exact same instruction executing with the same inputs.

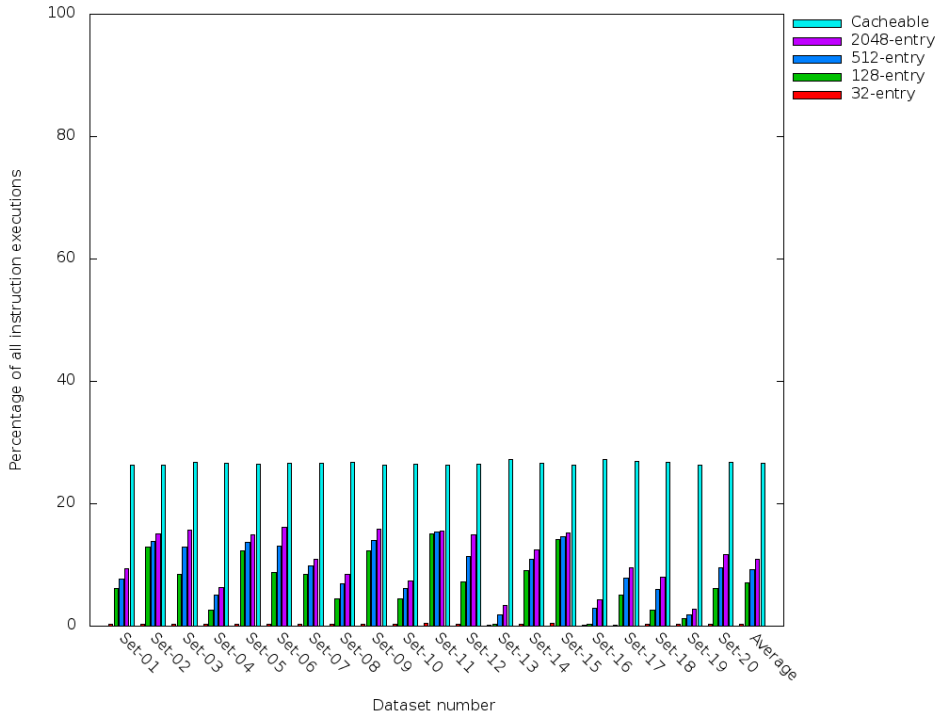

Figure 6.15: Automotive-susan-c. Cache hit rate for specified sizes of Local-Level Value Reuse Cache, and total percentage of cacheable instructions.

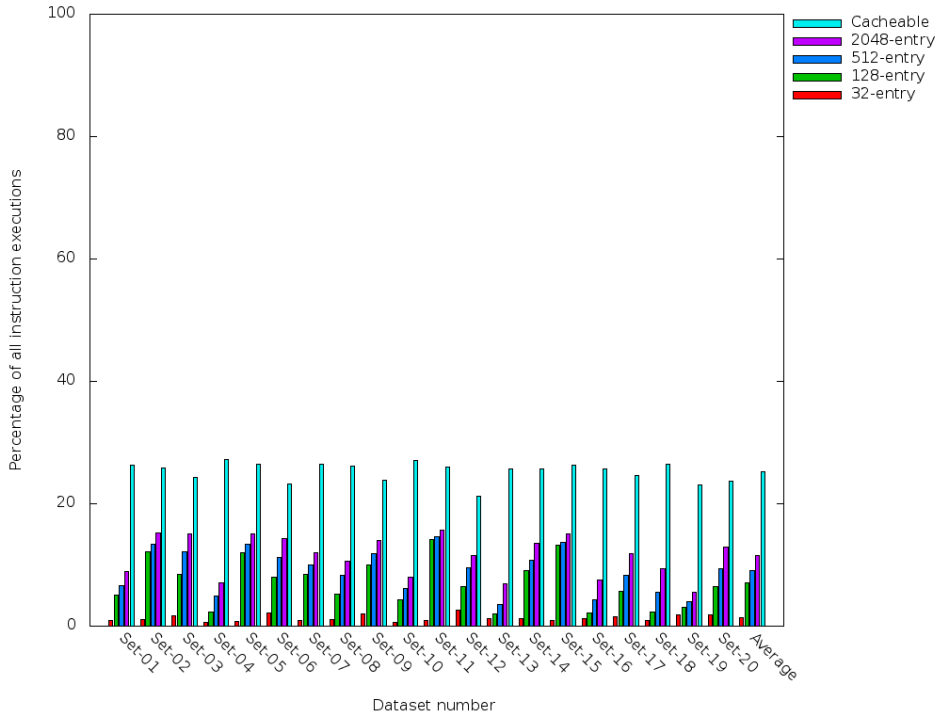

Figure 6.16: Automotive-susan-e. Cache hit rate for specified sizes of Local-Level Value Reuse Cache, and total percentage of cacheable instructions.

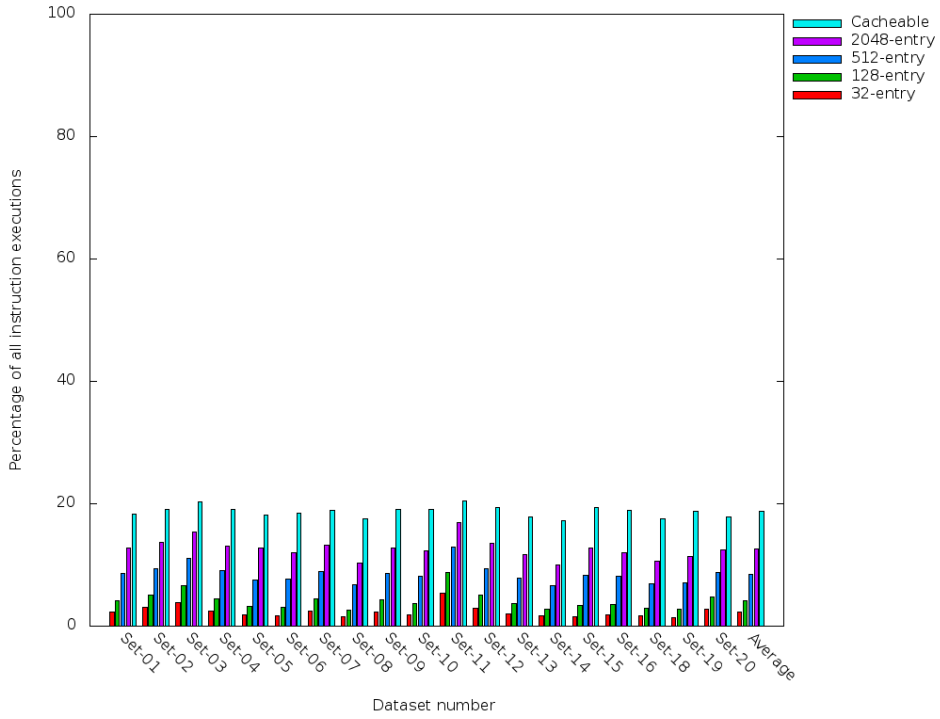

Figure 6.17: Consumer-jpeg-c. Cache hit rate for specified sizes of Local-Level Value Reuse Cache, and total percentage of cacheable instructions.

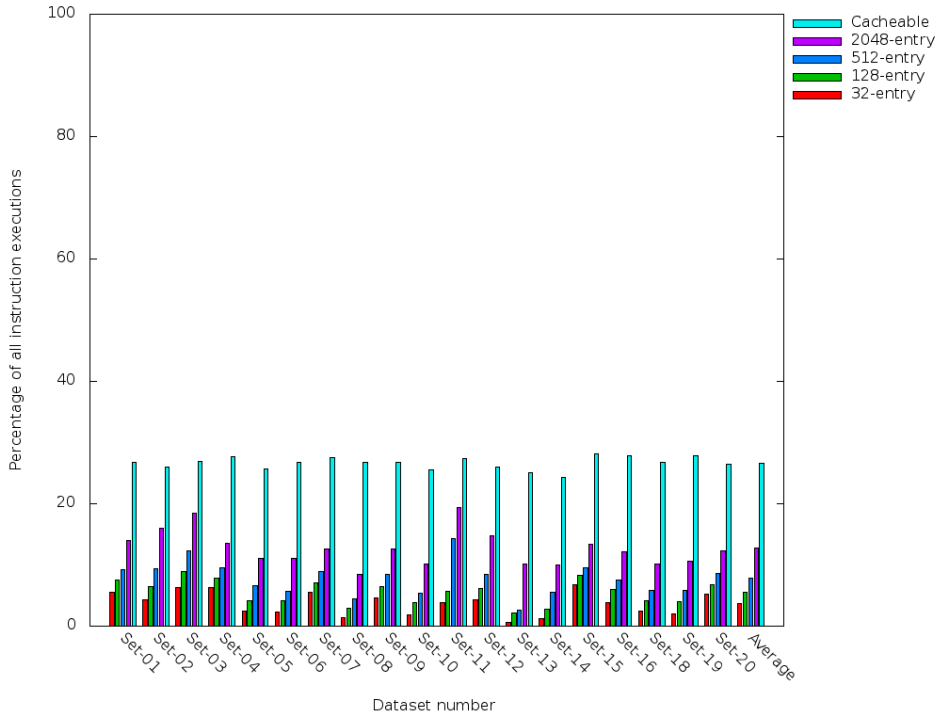

Figure 6.18: Consumer-jpeg-d. Cache hit rate for specified sizes of Local-Level Value Reuse Cache, and total percentage of cacheable instructions.

Network-dijkstra. It can be observed from the results that:

- Again the amount of exploitation of Value Reuse is generally decreased when compared to the Global-level Value Reuse Cache.
- An exception to this is the 2048-entry Value Reuse Cache, which provides a similar hit rate for both the Local-level and Global-level Value Reuse Cache implementations.

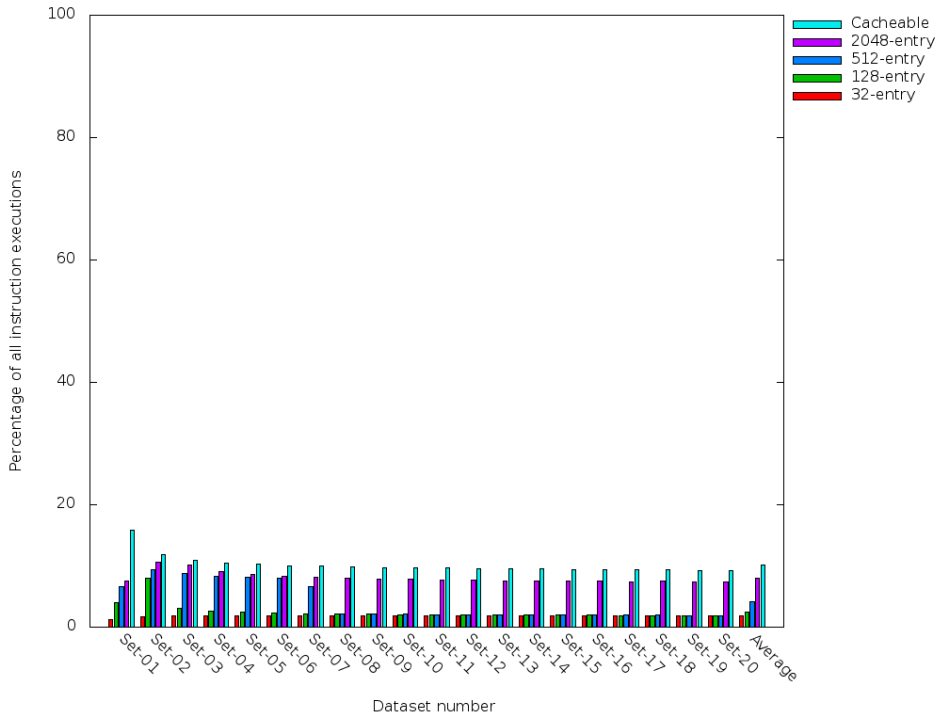

Figure 6.19: Network-dijkstra. Cache hit rate for specified sizes of Local-Level Value Reuse Cache, and total percentage of cacheable instructions.

Office-stringsearch. It can be observed from the results that:

- The amount of exploitation of Value Reuse is generally decreased when compared to the Global-level Value Reuse Cache.
- However, the 2048-entry Value Reuse cache provides similar hit rates for both the Local-level and Global-level Value Reuse Caches.

Security-rijndael-d. It can be observed from the results that:

- The amount of Value Reuse is generally decreased when compared to the Global-level Value Reuse cache.
- The 32- 128- and 512-entry Value Reuse Caches exhibit a greater decrease in hit rate than the 2048-entry Value Reuse Cache.

Security-sha. It can be observed from the results that:

- There is generally a decrease in the amount of exploitation of Value Reuse compared to the Global-level Value Reuse Cache.
- However, the 2048-entry Value Reuse Cache provides similar levels of exploitation for both Local-level and Global-level Value Reuse Caches.

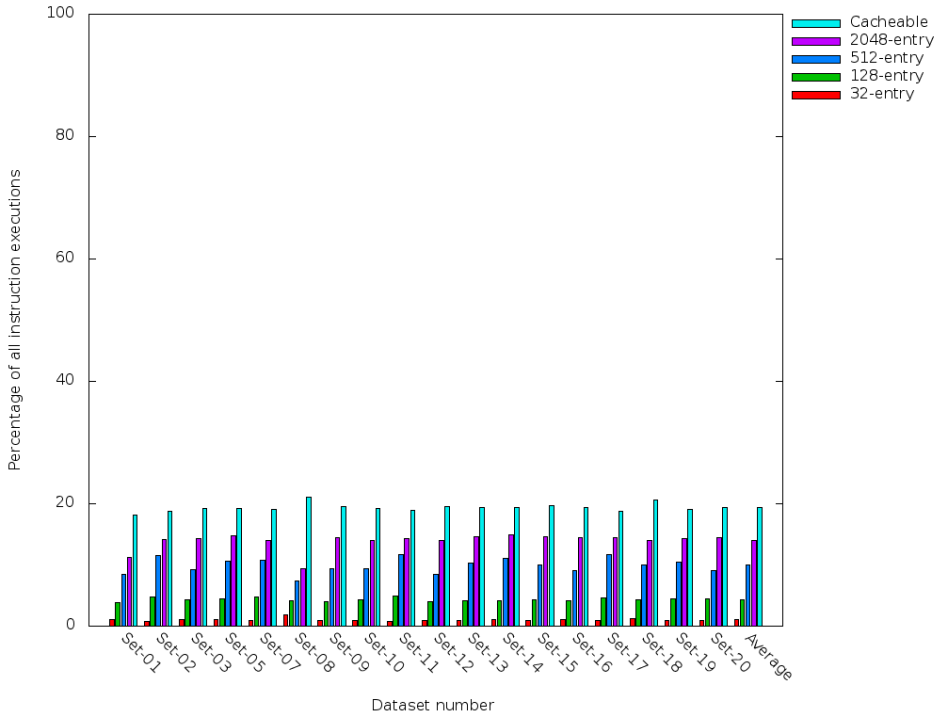

Figure 6.20: Office-stringsearch. Cache hit rate for specified sizes of Local-Level Value Reuse Cache, and total percentage of cacheable instructions.

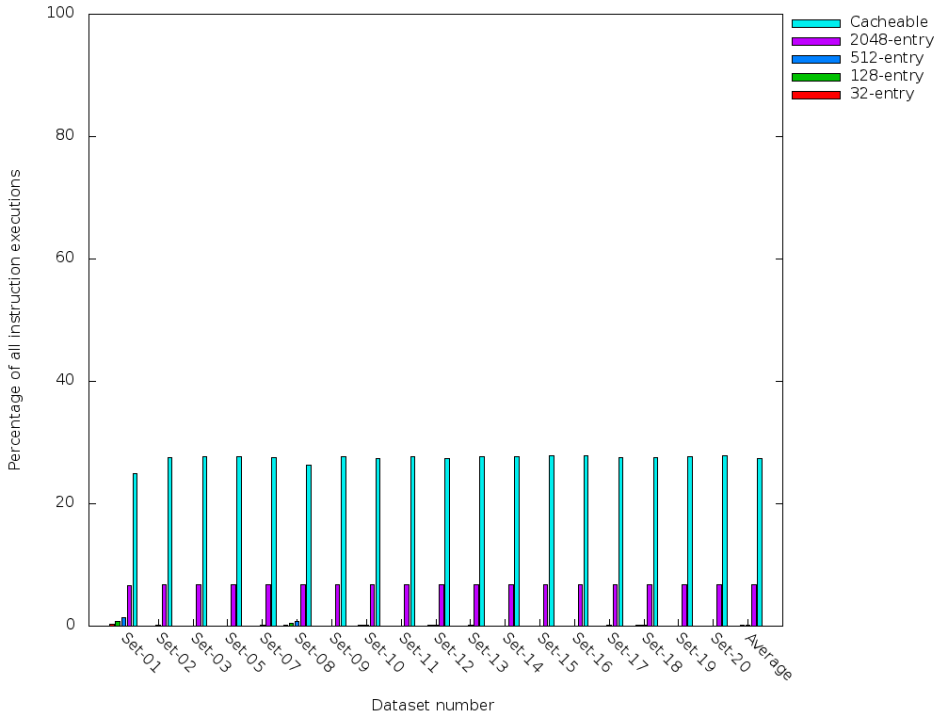

Figure 6.21: Security-rijndael-d. Cache hit rate for specified sizes of Local-Level Value Reuse Cache, and total percentage of cacheable instructions.

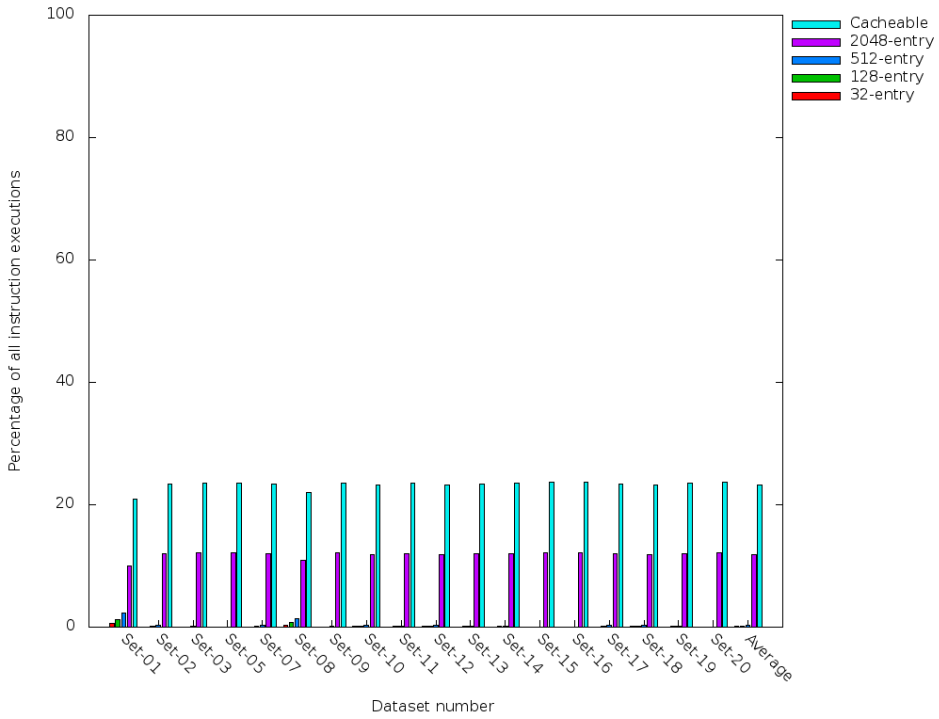

Figure 6.22: Security-sha. Cache hit rate for specified sizes of Local-Level Value Reuse Cache, and total percentage of cacheable instructions.

Telecom-adpcm-c. It can be observed from the results that:

- There is a similar amount of exploitation of Value Reuse for both Local-level and Global-level Value Reuse Caches.
- This suggests that most Value Reuse in this benchmark comes from the exact same instruction executing with the same inputs rather than instructions at different locations in memory executing with the same inputs.

Telecom-adpcm-d. It can be observed from the results that:

- Similar observations to those made for *telecom-adpcm-c* can be made for this benchmark.
- This result is likely to be due to the similarity between the functions performed by both benchmarks.

Telecom-crc32. It can be observed from the results that:

- The Local-level and Global-level Value Reuse caches provide a similar level of exploitation of Value Reuse for this benchmark.
- This would be expected behaviour for this benchmark as the CRC32 algorithm is likely to be implemented using a small loop where the same instruction repeatedly performs the CRC32 calculation, which would lead to the same instruction being responsible for most of the Value Reuse.

A Comparison Across all Benchmarks

It can be seen that for over half of the benchmarks, the amount of Value Reuse exploited by the Local-level Value Reuse Cache is less that the amount of Value Reuse exploited by the Global-level Value Reuse Cache.

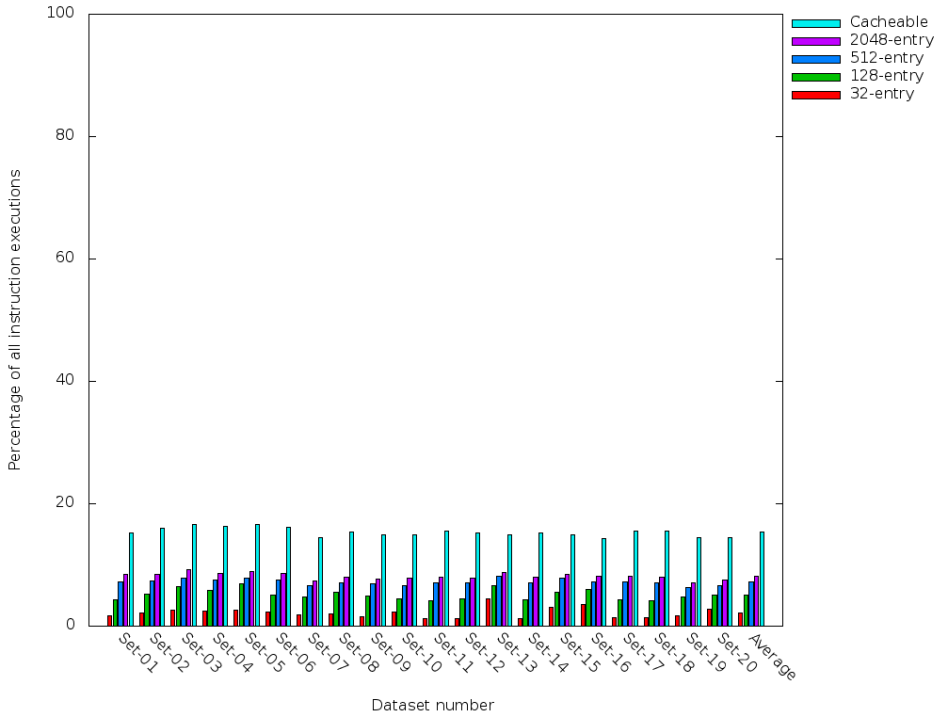

Figure 6.23: Telecom-adpcm-c. Cache hit rate for specified sizes of Local-Level Value Reuse Cache, and total percentage of cacheable instructions.

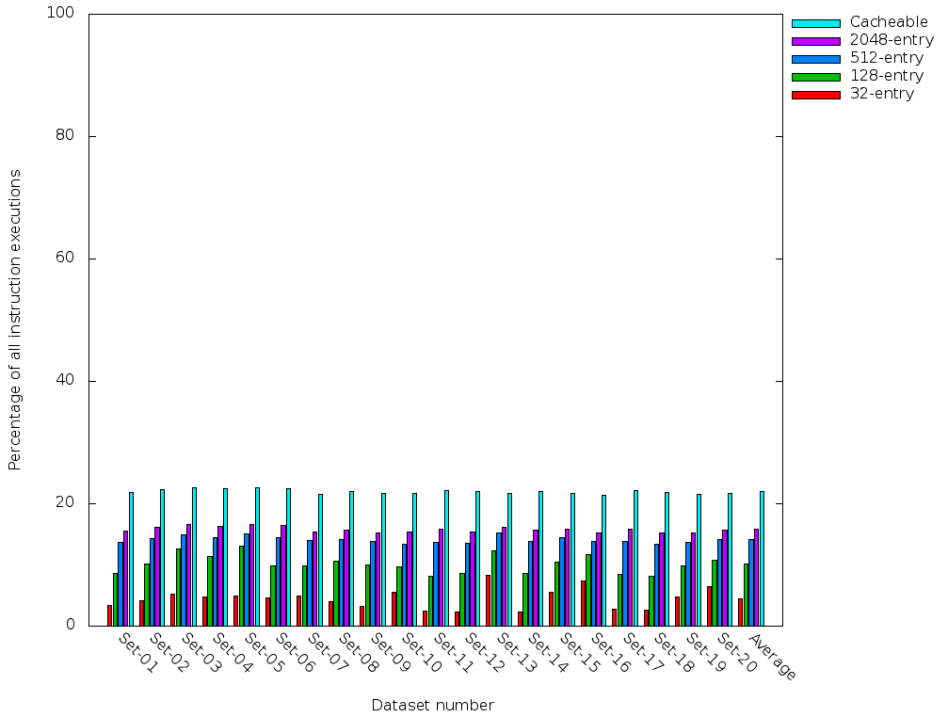

Figure 6.24: Telecom-adpcm-d. Cache hit rate for specified sizes of Local-Level Value Reuse Cache, and total percentage of cacheable instructions.

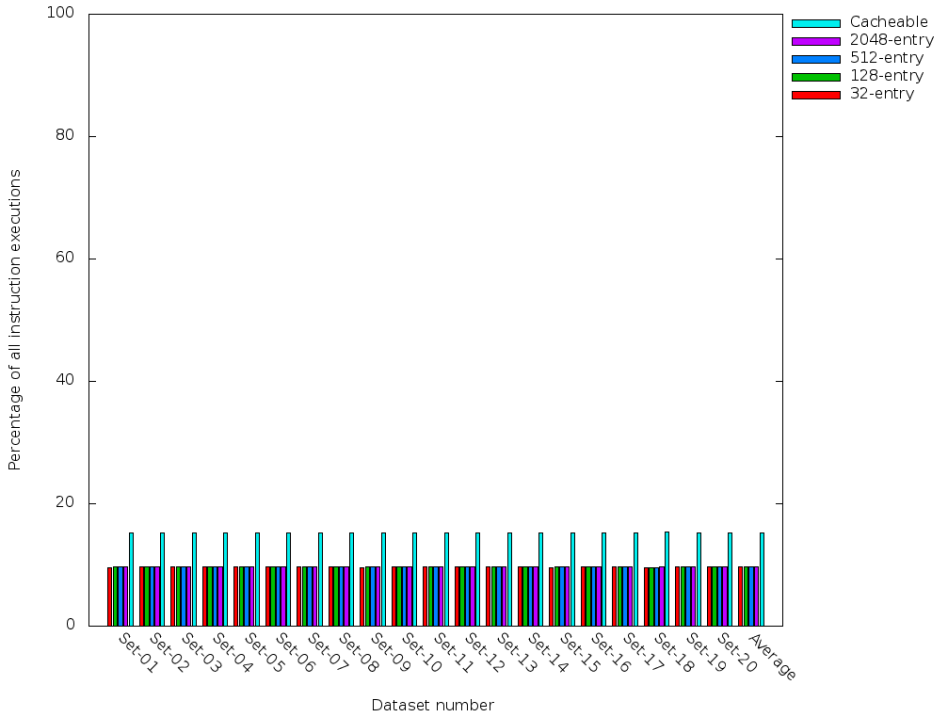

Figure 6.25: Telecom-crc32. Cache hit rate for specified sizes of Local-Level Value Reuse Cache, and total percentage of cacheable instructions.

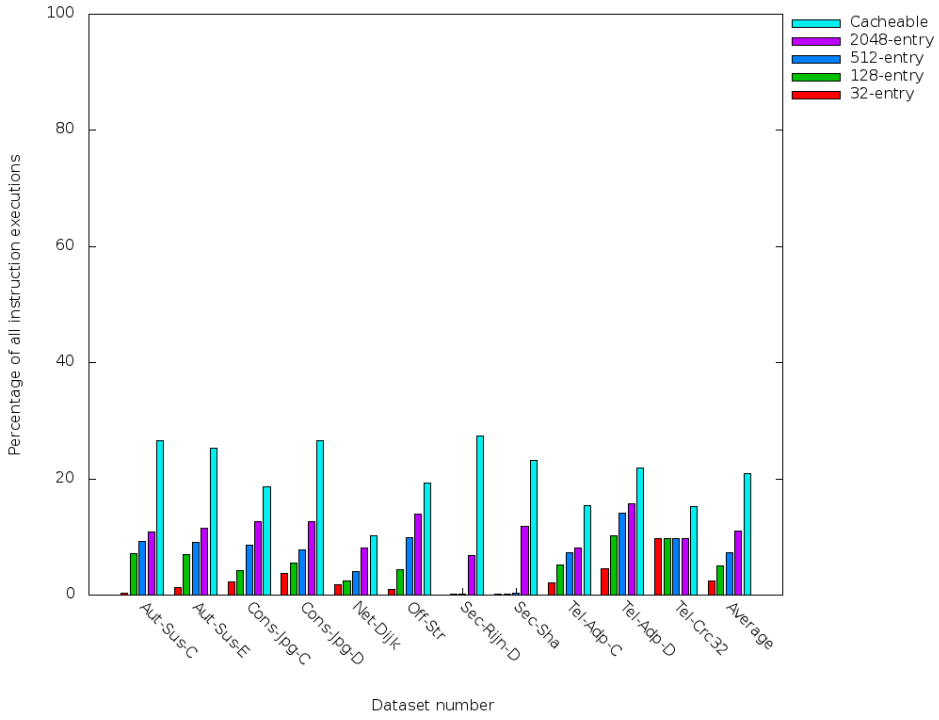

Figure 6.26: Comparison of the cache hit rate for specified sizes of Local-Level Value Reuse Cache and total percentage of cacheable instructions across all benchmarks.

6.6 Conclusion

Hypotheses 2 and 5 have been investigated in this chapter.

- The results presented in this chapter are in support of Hypothesis 2. The Global-level Value Reuse Cache was able to exploit Value Reuse in Instruction Executions more successfully that the Local-level Value Reuse Cache. Although there is no Local-level Value Profile data to confirm this result, this does agree with Hypothesis 3 (that there is a greater level of Value Reuse in Memory Accesses at the global level than at the local level) given that Hypothesis 4 (that there is a correlation between the amount of Value Reuse in Memory Accesses and Instruction Executions) is considered to be correct.
- The results are also in support of Hypothesis 5. It has been shown that the Value Reuse in Instruction Executions can be exploited to improve performance.

Although the results are in support of Hypothesis 5, there has been a relatively low hit rate exhibited for all caches across all benchmarks due to the percentage of cacheable instructions being relatively small. The instruction opcodes which were chosen to be supported by the Value Reuse Cache were intended to represent a significant fraction of all instructions. This situation could be rectified if the Value Reuse Cache supported more instruction opcodes. Alternatively, a Value Reuse Cache implemented on another architecture with a smaller number of instruction opcodes may provide a higher hit rate.

Chapter 7

A More Representative Memory Access Value Profile - Using a Cache Simulator

7.1 Background

Modern processors work at a much higher clock-rate, or speed than the main memory. As a result, if the processor requires a value from memory to perform its next computation, it can be idle for many cycles whilst it waits for the value to be retrieved. In order to reduce the impact of this disparity, all modern processors implement a cache, which stores the contents of frequently accessed memory locations close to the processor core. The cache performs operations much more quickly than the main memory. As a consequence, retrieving a value from the cache requires much less time than retrieving a value from main memory. This improves the overall performance of the processor.

Where a cache is implemented, it is not always necessary that the values required by the processor are transferred across the memory bus. Only the values which are stored in memory locations which are accessed relatively infrequently will be transferred across the bus. The Value Profiles of Memory Accesses collected so far have been representative of all the values which a program may load from or store to memory. These profiles are not representative of the values which are transferred across the data bus when a cache is implemented. In order to gather a more realistic Value Profile of Memory Accesses, Value Profiling must be performed in conjunction with simulating a cache.

7.2 A Pin Tool to Simulate a Cache

In the standard distribution of Pin, a cache simulator class is included in the folder `/Memory/cache.H`. An example use of this cache class is provided in the same folder, called `allcache.cpp`. This example implements the following:

- An Instruction Translation Lookaside Buffer.
- A Data Translation Lookaside Buffer.
- A Level 1 Instruction Cache.
- A Level 1 Data Cache.
- A Level 2 Unified Cache.
- A Level 3 Unified Cache.

The following parameters of each cache can be configured:

- Total size of cache.

- Size of a single line.
- Associativity of the cache.

The eviction policy implemented by the cache is a round-robin eviction policy. The default settings of these caches are unrealistic for a desktop machine. Sensible settings for the parameters were decided to be those which are most like the processor used in the machine used to run the Value Profiling. The processor in this machine is a Pentium 4 3GHz. The Intel IA-32 microarchitecture supports an instruction called `CPUID`, which can be used to gather information about the processor installed in the machine. A tool, called *CPUID* (Allen, 2008), was used to gather information about the caches in this machine. Using this tool the caches in this machine were determined to be:

- Level 1 Instruction Cache:
  - Total size: 16Kbytes.
  - Line Size: 64 bytes.
  - Associativity: 8-way set associative.
- Level 1 Data Cache
  - Total size: 16Kbytes.
  - Line Size: 64 bytes.
  - Associativity: 8-way set associative.
- Level 2 Unified Cache:
  - Total size: 1Mbyte.
  - Line size: 64 bytes.
  - Associativity 8-way set associative, sectored.

The instruction and data translation lookaside buffers are not being considered, as they have been determined to be outside the scope of this project. Additionally, a Trace Cache (Rotenberg *et al.*, 1996) is implemented in the machine, which the cache simulator class included with Pin does not implement, so this is disregarded. Another limitation of the cache simulator class is that it does not implement a sectored cache. The following changes were made to the cache class to gather Value Profile data for Memory Accesses.

- Removal of the level 3 cache.
- Changing the parameters of the level 1 and 2 caches to match those in the project machine.
- The code which implements instruction and data translation lookaside buffers was not modified as it is outside the scope of this project. It was not removed, as this was considered an unnecessary change. Making additional unrequired changes was avoided to reduce the potential for error.
- Inclusion of the MemProfile and ProfileData classes into the source.
- Inclusion of the `RecordMem()` function from the modified `pinatrace.cpp`, which records the values transferred across the bus.
- Insertion of code to instrument memory accesses.
- Addition of command line switches to control the output profile name, and addition of code to output the profile at the end of the execution.

7.3 Insertion of Instrumentation Code

A memory access will be necessary upon the event of a miss on the Level 2 unified cache. Therefore, the point at which the operation of the Level 2 cache is implemented will be a good candidate for instrumentation. The function `ul2Access` implements the operation of the Level 2 cache:

```
LOCALFUN VOID Ul2Access(ADDRINT addr, UINT32 size, CACHE_BASE::ACCESS_TYPE accessType)
{
    // second level unified cache
    const BOOL ul2Hit = ul2.Access(addr, size, accessType);

    // third level unified cache
    if ( ! ul2Hit) ul3.Access(addr, size, accessType);
}
```

The function first calls the `Access()` method of the Level 2 cache, which return `true` in the event of a cache hit and `false` otherwise. In the event of a cache miss, the Level 3 cache is then accessed. However, the Level 3 cache has been removed. Instead, the details of the memory access need to be recorded. The code has been modified as follows:

```
LOCALFUN VOID Ul2Access(ADDRINT addr, UINT32 size, CACHE_BASE::ACCESS_TYPE accessType)
{
    const BOOL ul2Hit = ul2.Access(addr, size, accessType);
    if ( ! ul2Hit)
      RecordMem(0x0, 'B', (VOID*)addr, size, false);
}
```

The first argument passed to the `RecordMem()` function is the address of the program counter - this is not important in this simulation so has simply been set to 0. The second argument is a `char` which determines the direction of the transfer. Normally this would be `R` or `W`. However, as this simulation is intended to determine only the values transferred across the bus without consideration for their direction, the argument is always set as `B`, so that the same value transferred in either direction is considered the same value transferred. The third argument is the address at which the transferred value resides. The fourth argument specifies the size in bytes of the transfer, and the fifth argument represents whether this memory access is a prefetch. This argument is always set to false. It appears that only the size and address arguments are necessary to be passed to the `RecordMem()` function. However, the `RecordMem()` function has been directly taken from the Pin Memory Access Value Profiling tool already developed. This function, and all other functions which were reused were not modified, so that they need not be re-tested in this implementation.

7.4 Testing

It is assumed that the cache simulation tools which have been modified function correctly. Therefore, no testing has been performed on them. Additionally, as all the Value Profiling classes were copied verbatim from the Pin Memory Access Value Profiling tool, these portions of this tool were not tested as they had already been part of testing in Sections 4.5.5 and 4.5.6.

7.5 Results

The implementation of the cache simulator alongside Value Profiling of Memory Accesses provides quite different results to that of Value Profiling alone. The results for each individual benchmark will be discussed.

Automotive-susan-c. It can be observed from the results that:

- On average, over 80% of all memory accesses involve the transfer of a single value.

- Examination of the Value Profile Data reveals that this value which is most frequently transferred is always zero across all datasets.
- As this single value is responsible for such a large fraction of memory accesses, other values are not considered as significant.

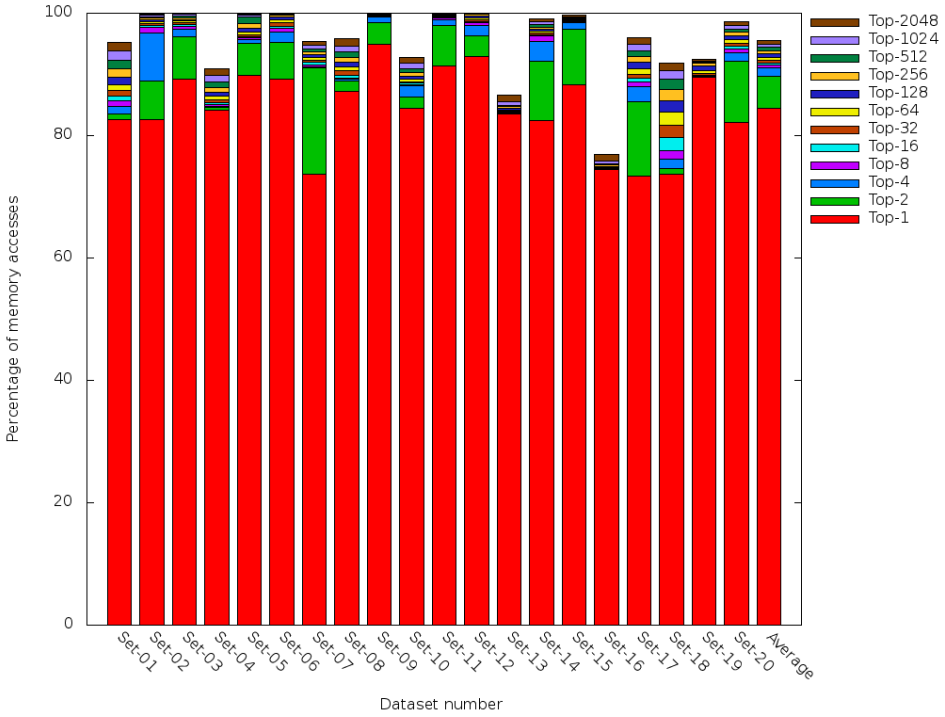

Figure 7.1: Automotive-susan-c. Percentage of all memory accesses accounted for by the top N most frequently transferred values when a cache is implemented.

Automotive-susan-e. It can be observed from the results that:

- Similarly to *automotive-susan-c*, there is a very high proportion of memory accesses which involve a single value.
- Again for all datasets, this value is zero across all datasets.
- Again, other values transferred do not represent a significant percentage of memory accesses.

Consumer-jpeg-c. It can be observed from the results that:

- On average over 30% of all memory accesses involve the transfer of a single value. This is far less than the Automotive-susan benchmarks, but still a significant proportion.
- Across all datasets, the most frequently transferred value is zero, with two exceptions. When using datasets 3 and 11, the most frequently transferred value is &FFFFFFFF. However, the second most frequently transferred value is zero.
- Therefore, the most frequently transferred value is generally zero for this benchmark.
- Again other values do not represent a significant percentage of memory accesses, in comparison to the single most frequently transferred value.

Consumer-jpeg-d. It can be observed from the results that:

- A large percentage (over 70%) of all memory accesses involve the transfer of a single value.
- Across all datasets, this value is always zero.

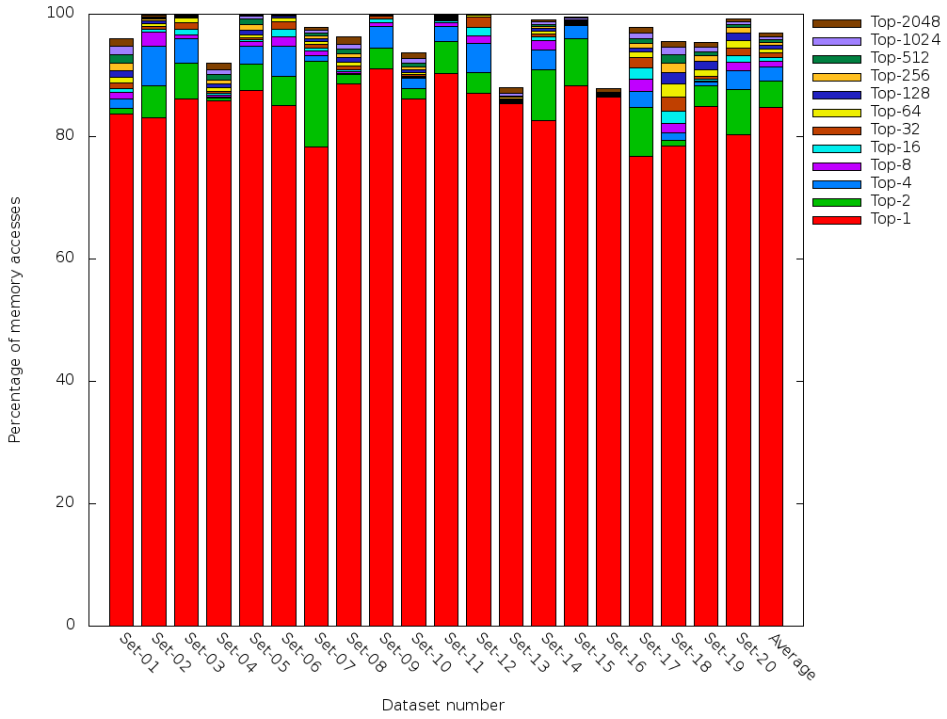

Figure 7.2: Automotive-susan-e. Percentage of all memory accesses accounted for by the top N most frequently transferred values when a cache is implemented.

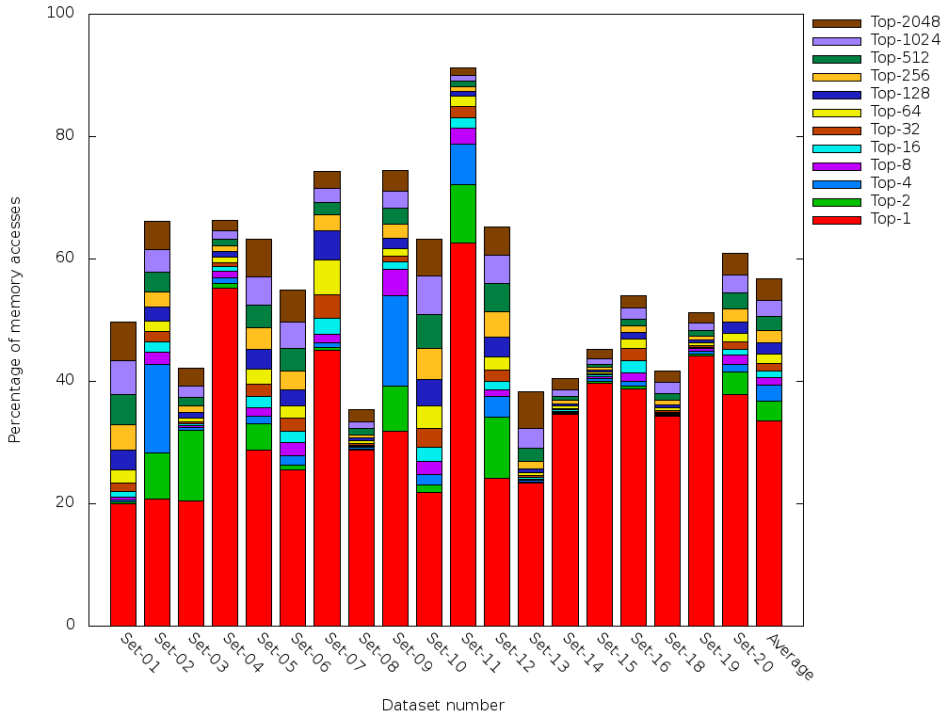

Figure 7.3: Consumer-Jpeg-C. Percentage of all memory accesses accounted for by the top N most frequently transferred values when a cache is implemented.

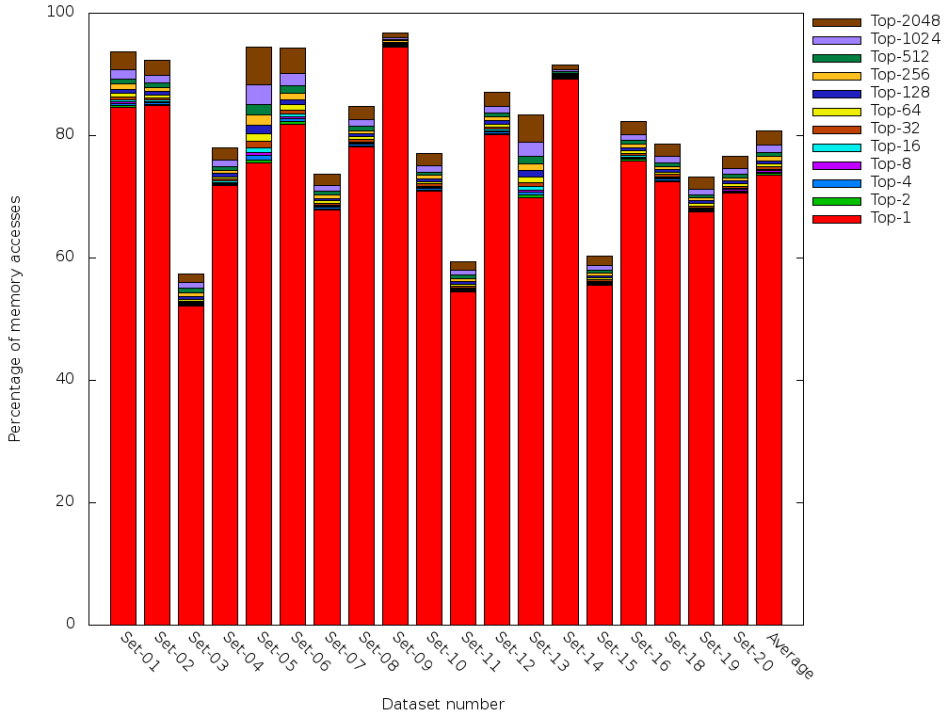

Figure 7.4: Consumer-Jpeg-D. Percentage of all memory accesses accounted for by the top N most frequently transferred values when a cache is implemented.

- Other values which are transferred do not represent a significant fraction of memory accesses.

Network-dijkstra. It can be observed from the results that:

- On average, over 40% of all memory accesses involve the transfer of a single value.
- The most frequently transferred value is always zero across all datasets.
- The trend across all the datasets is the opposite to the trend found for this benchmark for other Value Profiling results. As the size of the dataset increases, the percentage of memory accesses involving the transfer of the single most frequently transferred value increases.
- Further investigation may be required to determine the cause of this effect.

Office-stringsearch. It can be observed from the results that:

- A large percentage (over 60%) of all memory accesses involve the transfer of a single value.
- As with other benchmarks, the most frequently transferred value is zero across all datasets.
- Non-zero values again do not represent a significant fraction of memory accesses.

Security-rijndael-d. It can be observed from the results that:

- A modest percentage (approximately 20%) of all memory accesses involve the transfer of a single distinct value.
- Again the most frequently transferred value is zero across all datasets.
- The relatively low percentage of all memory accesses involving the transfer of this single value is not unexpected, as this benchmark has shown low levels of Value Reuse in other Value Profiling areas.
- No single non-zero value represents a significant percentage of all memory accesses.

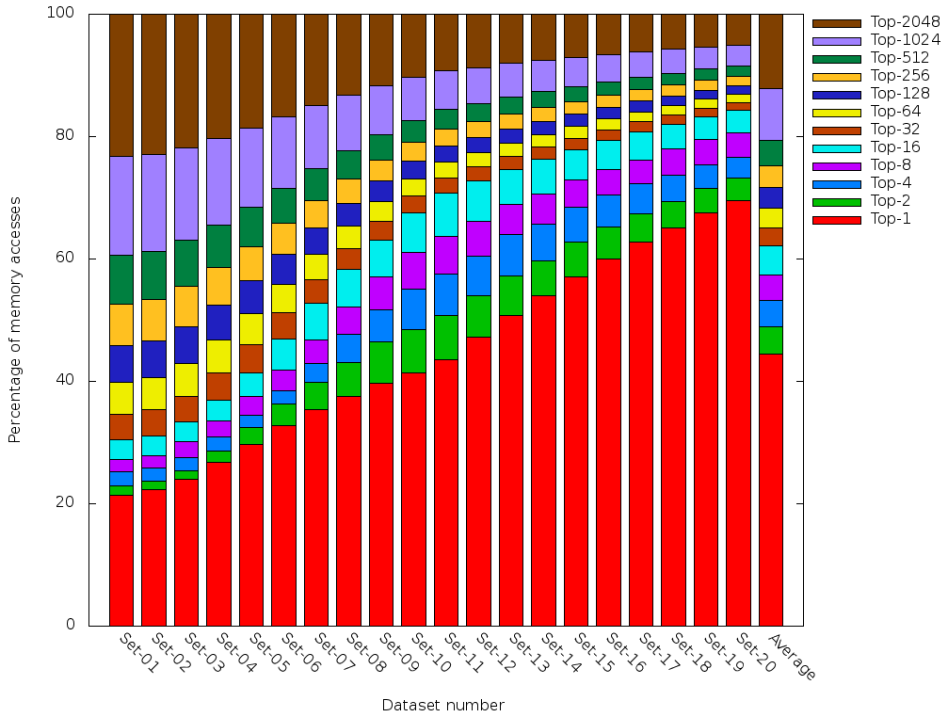

Figure 7.5: Network-Dijkstra. Percentage of all memory accesses accounted for by the top N most frequently transferred values when a cache is implemented.

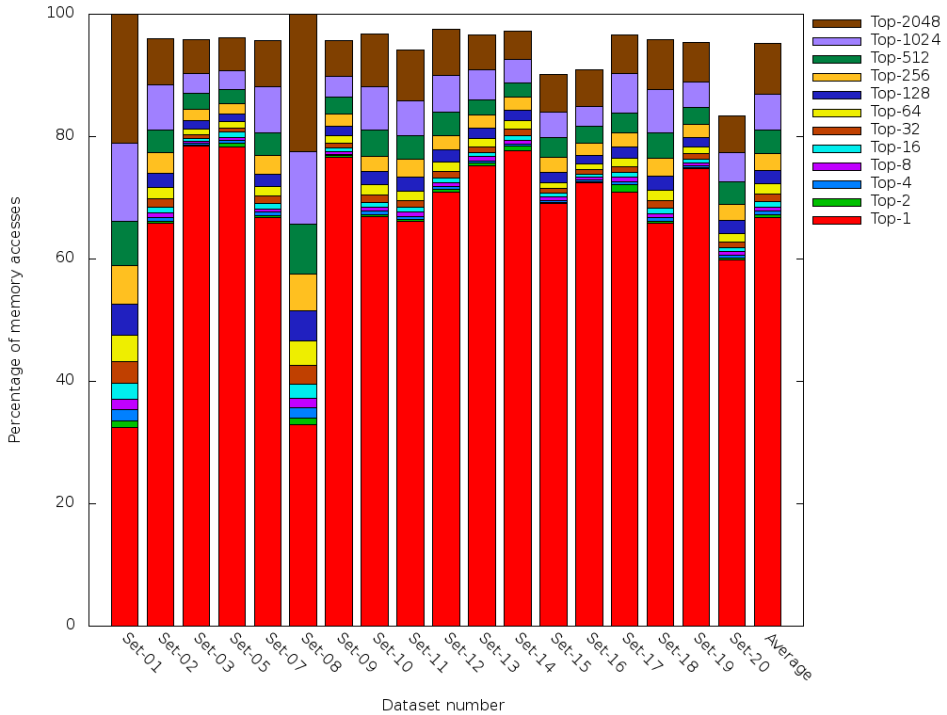

Figure 7.6: Office-Stringsearch. Percentage of all memory accesses accounted for by the top N most frequently transferred values when a cache is implemented.

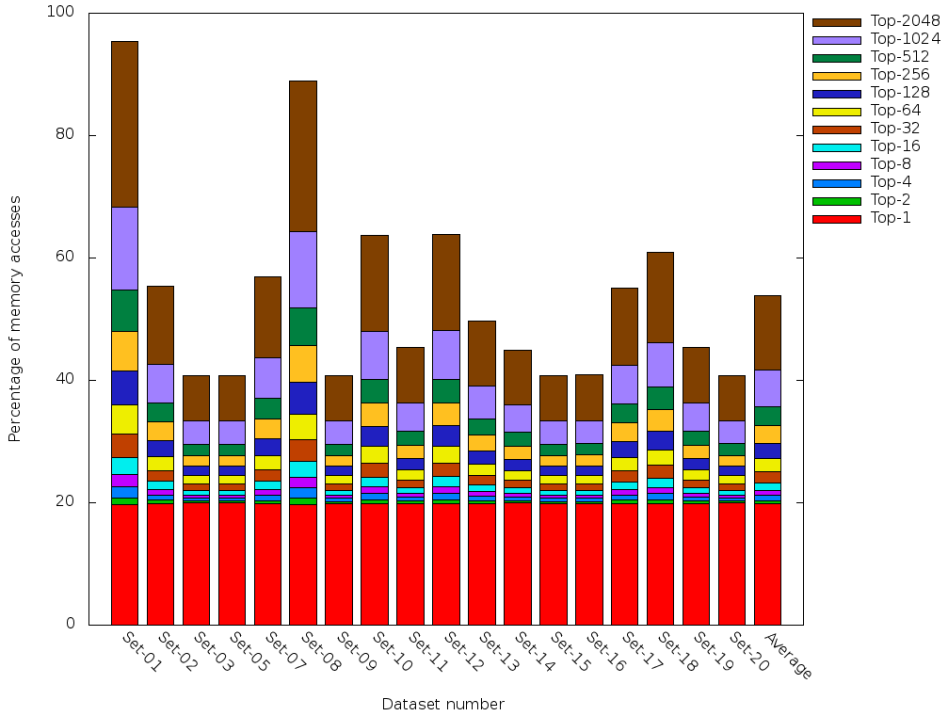

Figure 7.7: Security-Rijndael-D. Percentage of all memory accesses accounted for by the top N most frequently transferred values when a cache is implemented.

Security-sha. It can be observed from the results that:

- On average a relatively low (around 10%) of all memory accesses involve the transfer of a single distinct value. However, with two datasets, set 1 and set 8, the percentage is much higher at around 30%.
- This behaviour is not unexpected, as Security-sha has shown the lowest levels of Value Reuse when examined using other forms of Value Profiling.
- Across all datasets, the single most frequently transferred value is again zero.
- No single non-zero value represents a significant percentage of memory accesses.

Telecom-adpcm-c. It can be observed from the results that:

- A relatively low (around 20%) of all memory accesses involve the transfer of a single distinct value.
- Across all datasets, the most frequently transferred value is again zero.
- No non-zero value represents a significant percentage of all memory accesses.

Telecom-adpcm-d. It can be observed from the results that:

- As with *telecom-adpcm-c*, a relatively low (around 20%) percentage of all memory accesses involve the transfer of a single distinct value.
- The results for this benchmark are almost exactly the same as for *telecom-adpcm-c*. This is expected, as the two benchmarks have shown very similar results when examined previously using Value Profiling.
- Across all datasets, the most frequently transferred value is again zero.
- Again no single non-zero value represents a significant fraction of all memory accesses.

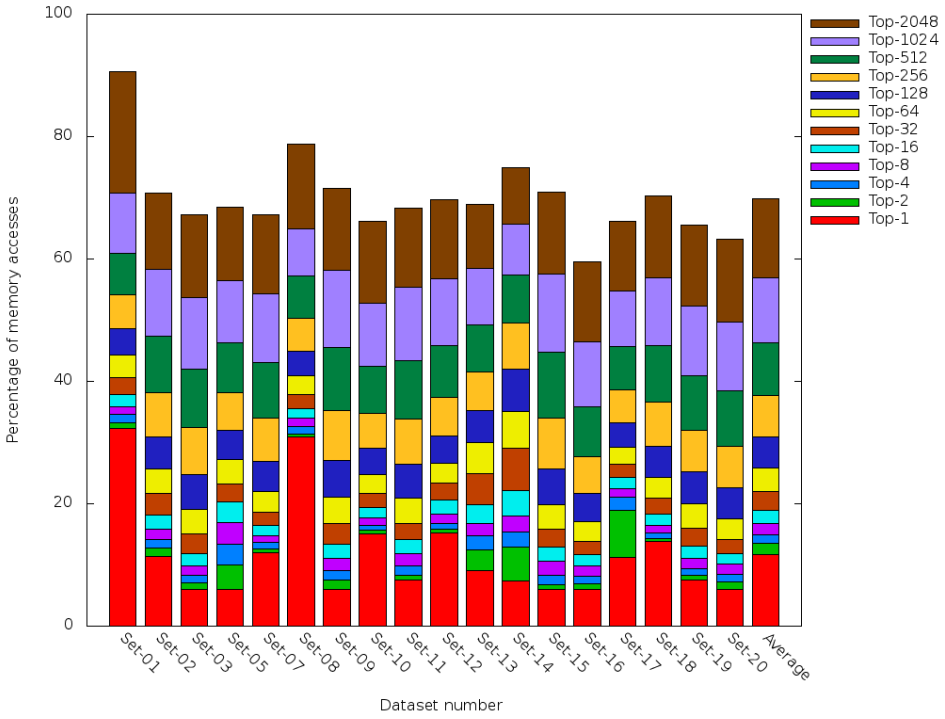

Figure 7.8: Security-Sha. Percentage of all memory accesses accounted for by the top N most frequently transferred values when a cache is implemented.

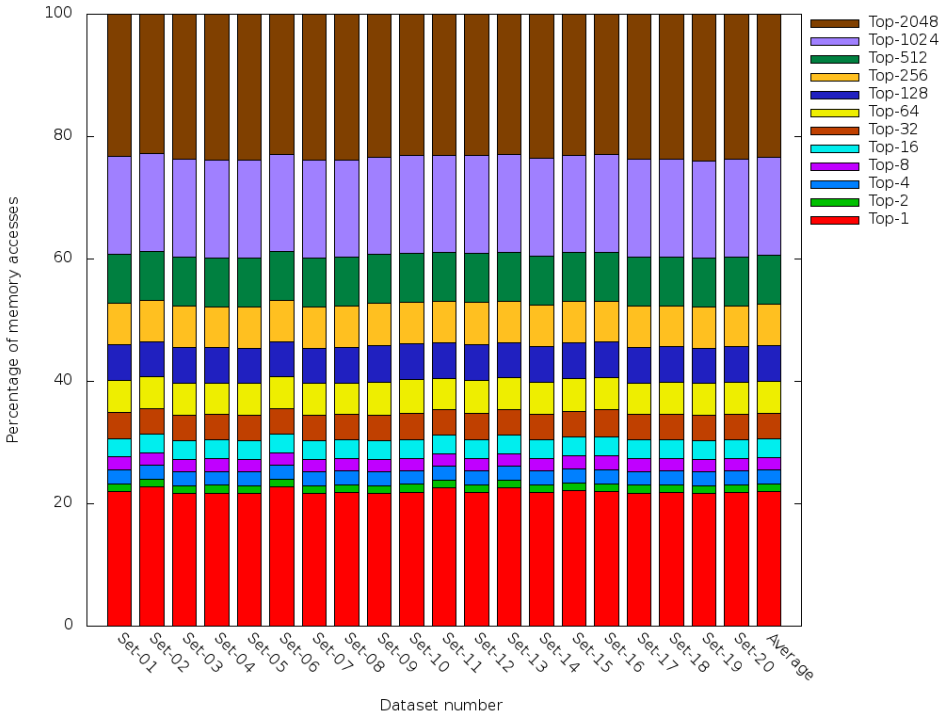

Figure 7.9: Telecom-Adpcm-C. Percentage of all memory accesses accounted for by the top N most frequently transferred values when a cache is implemented.

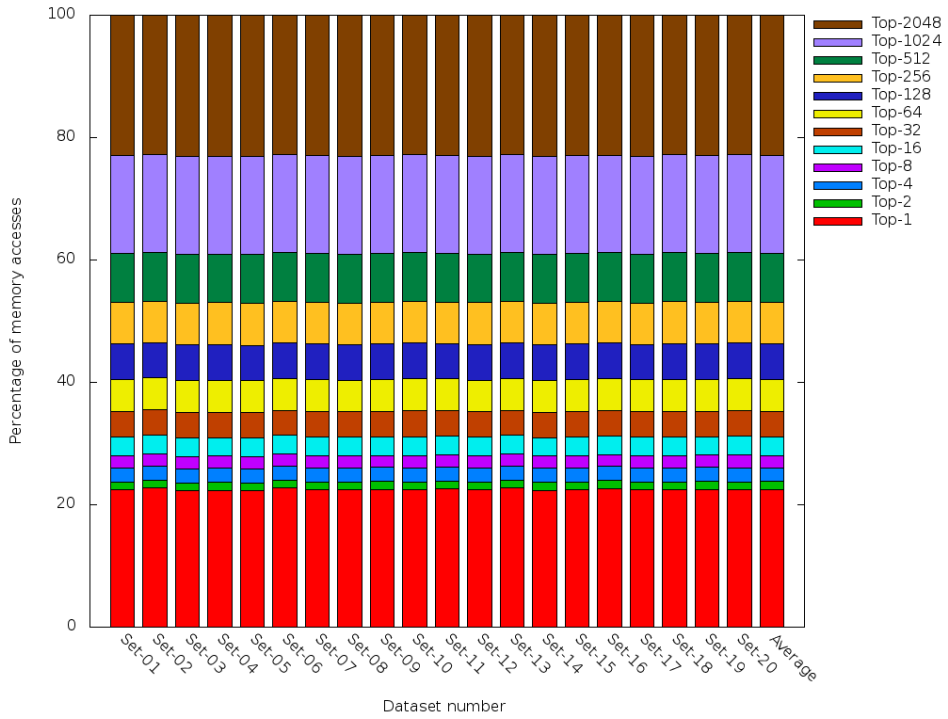

Figure 7.10: Telecom-Adpcm-D. Percentage of all memory accesses accounted for by the top N most frequently transferred values when a cache is implemented.

Telecom-crc32. It can be observed from the results that:

- Again a relatively low (around 20%) percentage of all memory accesses involve the transfer of a single value. However, this is a high percentage compared to the amount of reuse of a single value found in this benchmark for other forms of Value Profiling.
- Across all datasets, the most frequently transferred value is again zero.
- No single non-zero value represents a significant percentage of all memory accesses.

7.5.1 A Comparison Across all Benchmarks

It can be seen that on average, approximately 44% of all memory transfers involve the transfer of a single distinct value. Examination across all datasets of all the benchmarks has found that this value is consistently zero, with only two exceptions out of the 220 total runs of benchmarks and datasets. It can be concluded that where a cache is present, zero values are frequently transferred to and from the main memory. It has also been seen that the implementation of a cache greatly reduces the percentage of all memory accesses which do not transfer a zero value.

Additionally, the implementation of the cache increases the level of Value Reuse in Memory Accesses, even when it appears (using Value Profiling of Memory Accesses without a cache simulator) that there is little Value Reuse to be found in a benchmark. An example of this occurring is within Telecom-crc32, which previously did not appear to transfer any single value to or from memory repeatedly, yet when a cache is implemented, it is found that zero is transferred far more frequently than any other value.

Because zero values are transferred so frequently, any scheme which seeks to exploit the Value Reuse in Memory Accesses, whatever its purpose, should be designed with consideration for transferring zero values as efficiently as possible.

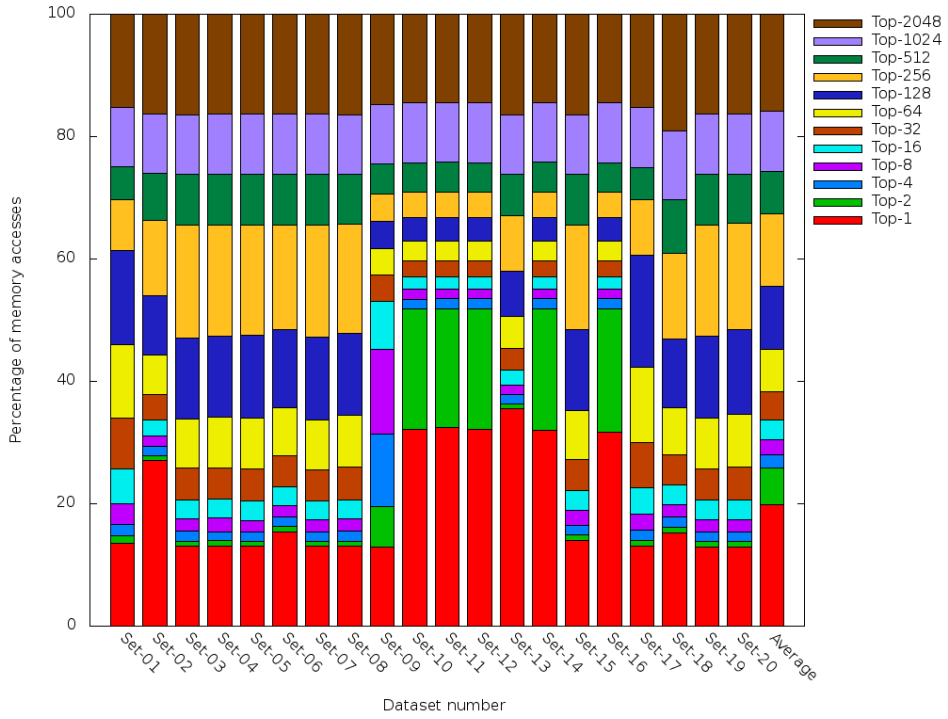

Figure 7.11: Telecom-Crc32. Percentage of all memory accesses accounted for by the top N most frequently transferred values when a cache is implemented.

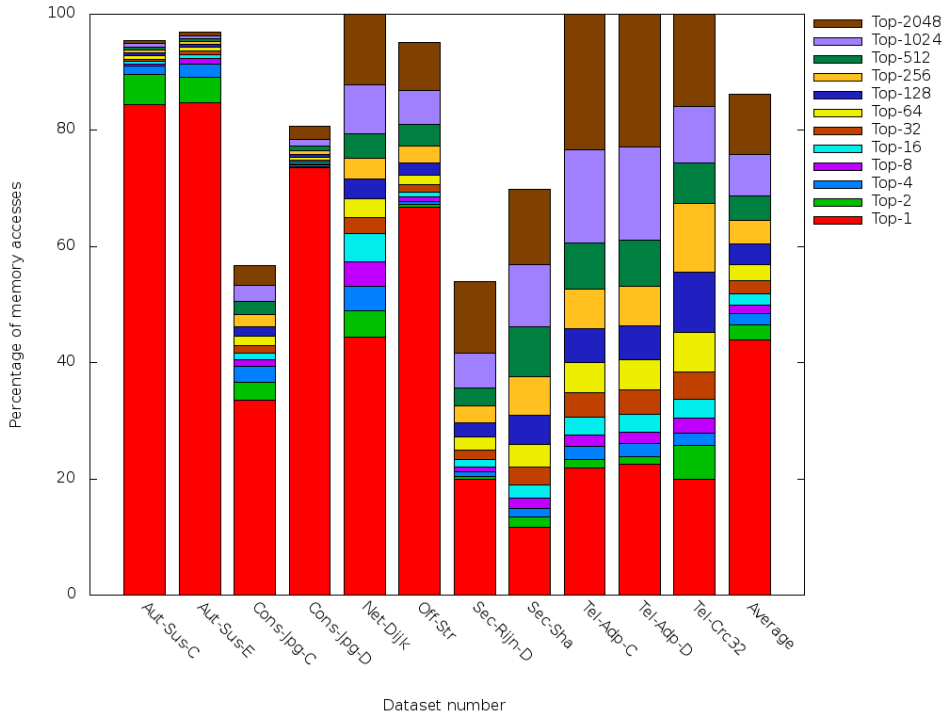

Figure 7.12: Comparison of the percentage of all memory accesses accounted for by the top N most frequently transferred values when a cache is implemented across all benchmarks.

7.6 Reducing Power Consumption by Exploiting the Effect of a Cache

(Yang & Gupta, 2002) performed Value Profiling of memory accesses, and used the information gathered to design an encoding for a low power data bus. However, the scheme which was developed did not place emphasis on the power-efficient transfer of a single value, but instead developing an encoding for the efficient transfer of a number of distinct frequently transferred values. A scheme to exploit the pattern of memory access found in the Value Profiling of Memory Accesses using a Cache Simulator will be presented here. However, the scheme focusses on the efficient transfer of the zero value, rather than the efficient transfer of a number of frequent zero or non-zero values, as the Value Profile data presented in this section implies that this is where the greatest exploitation may be possible.

It is stated in (Yang & Gupta, 2002) that the power consumption on a data bus is not a characteristic of how many data lines are switched on at a particular time, but instead that the *switching activity* on the bus is the main source of power consumption. Switching activity is the change in state of a particular line, either the turning on or turning off of current in a wire. Therefore, the scheme which is presented has been developed to try to minimise the amount of switching on the data bus involved in transferring a single value.

7.6.1 The Scheme

A data bus which already exists will have an existing line added to it. This line will be termed the *zero line*. The purpose of this line is to represent whether the value being transferred is zero or not. No other additions are made to the bus. An example of an 8-bit data bus between the CPU and memory with and without this modification is shown in Figure 7.13.

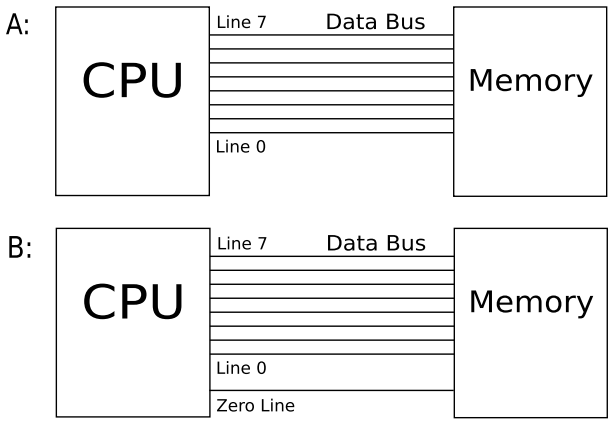

Figure 7.13: An 8-bit data bus. A: Without zero line. B: With zero line.

The operation of the zero line is as follows:

- When a zero value is transferred across the data bus, the zero line is switched on. All other data lines remain in their previous state.
- When a non-zero value is transferred across the bus, the zero line is switched off. The other lines of the data bus change state to represent the non-zero value, as if the zero line were not present.

In operation, one of the following four scenarios may occur each time a value is transferred across the bus:

1. A zero value is transferred, and the previous value transferred was also a zero. No switching activity is necessary in this scenario. All lines of the data bus remain in their previous state. The zero line remains switched on. If the majority of values transferred are zero, then this scenario is likely to occur frequently.

2. A zero value is transferred, and the previous value transferred was non-zero. The switching of a single line is necessary in this scenario. All of the lines of the data bus remain in their previous state. The zero line is switched on. If the majority of transferred values are zero, this scenario will occur less frequently than scenario 1.

3. A non-zero value is transferred, and the previous value transferred was a zero. Multiple lines may be switched in this scenario. The zero line is switched off, and the lines of the data bus are switched from the state representing the old value to the state representing the new value. This scenario is likely to occur with a similar (low) frequency to scenario 2.

4. A non-zero value is transferred, and the previous value transferred was non-zero. Multiple lines will be switched in this scenario. The zero line remain switched off. The scheme essentially has no effect in this scenario. If the majority of values transferred are zero, this scenario will occur less frequently than scenarios 2 and 3, and much less frequently than scenario 1.

7.6.2 Example Operation of the Scheme

Example transfer without the scheme. This example demonstrates the operation of the scheme when three values are transferred consecutively over the data bus. First &EC is transferred, then &0 is transferred, then &5F is transferred. Figure 7.14 shows the lines which are turned on when each value is transferred. At state A, lines 7, 6, 5, 3 and 2 are turned on, to represent the value &EC. To subsequently transfer &0 (state B), all of these lines have to be switched off. This is a total of 6 lines switched. When the value &5F is transferred, the lines 6, 4, 3, 2, 1 and 0 are switched on. This is a further 6 lines switched. The total number of lines switched in this case is 12.

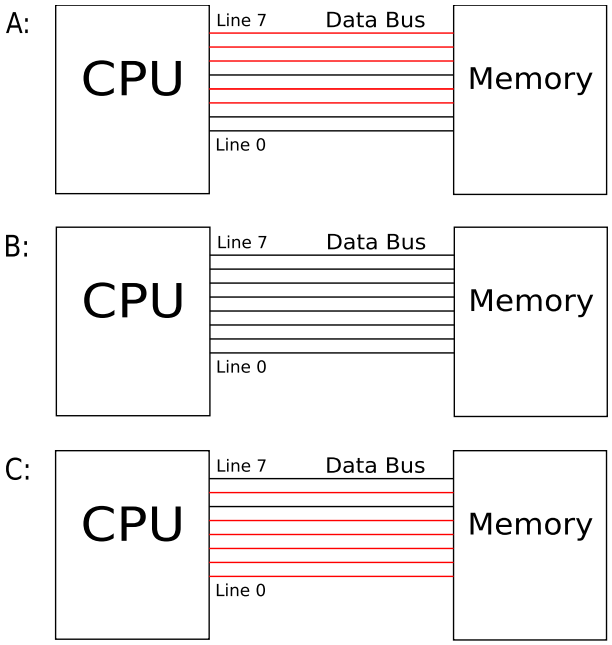

Figure 7.14: Three values transferred on an unmodified bus. Red represents a line switched on, black switched off. A: &EC transferred. B: &0 transferred. C: &5F transferred.

Example transfer with the scheme. Figure 7.15 shows which lines are turned on when each value is transferred on a data bus which does implement the scheme. As in the previous example, the line 7, 6, 5, 3 and 2 are turned on in state A. When the &0 value is to be transferred, these lines remain in the same state. The zero line is switched on. This state now represents the value zero being transferred, and has only required the switching of one line. When the third value is transferred, the zero line is turned off, and the lines 0-7 are set into the state where lines 6, 4, 3, 2, 1 and 0 are turned on. This requires a total of six lines to be switched. The total number of lines switched in

this example is 7. This is less than 60% of the switching required by a data bus which does not implement the scheme.

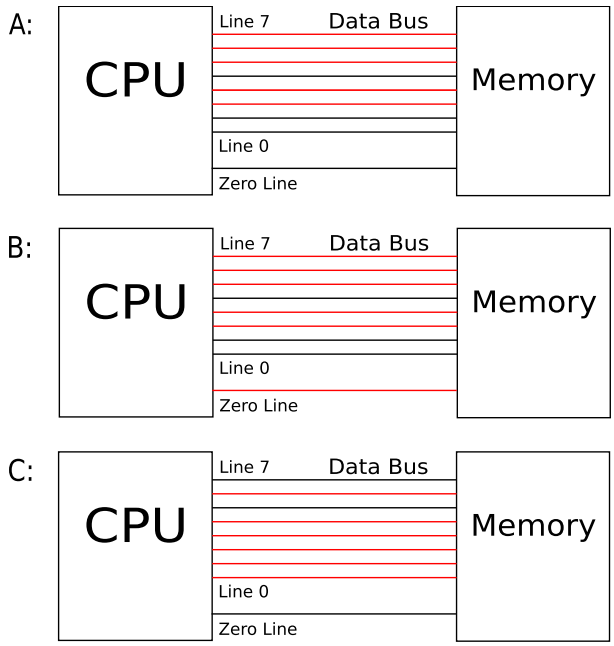

Figure 7.15: Three values transferred on a modified bus. Red represents a line switched on, black switched off. A: &EC transferred. B: &0 transferred. C: &5F transferred.

7.6.3 Testing of the Scheme

Testing of the scheme is beyond the scope of this project. However, two testing methods will briefly be discussed:

1. A software test of this scheme could be written as a Pin Tool. The software implementation of the scheme would potentially record the values transferred across the bus and the number of switching operations necessary to transfer these values both with and without the implementation of the scheme. The reduction in switching generated by the scheme could be calculated, and a reduction in power consumption could be estimated. The software test of the scheme would be subject to the same limitations that the profiling using Pin is - for example the cache simulator available is limited. It would be possible to implement a more sophisticated cache simulator, but this would require extra work to develop and validate. This method has the advantage that it would be very easy to modify the scheme and perform additional testing, in order to refine the scheme to further reduce power.

2. The scheme could be implemented in hardware. The scheme would be tested by executing test code and measuring the power consumption of the device on which the scheme is implemented. Another test would be performed on the same hardware, without an implementation of the scheme. Again the power consumption of the device would be measured running the same code. The power consumption of the two devices could be compared to determine the reduction in power consumption. This method of testing has the advantage that it provides a very accurate representation of the power saving incurred by the scheme. However, this method of testing would be very costly, and modifying the scheme incrementally would be difficult as this would require the manufacture of a new device with the modified implementation of the scheme.

7.7 Conclusion

In this chapter the technique for Value Profiling of Memory Accesses has been refined by using an implementation of a cache simulator to more accurately determine the values transferred across the data bus. This has shown that the true level of Value Reuse in Memory Accesses is far higher than is suggested by the Value Profile data which was recorded without the implementation of the cache.

Additionally, it has been shown that this Value Profile data can be used to guide the design of an encoding for a low-power data bus. There are no situations in which the encoding presented will increase the switching activity on the data bus. Therefore, it is very likely that the encoding will always reduce switching activity on the bus (as opposed to not changing the level of switching activity). This conclusion is in support of Hypothesis 6.

Chapter 8

Conclusions

8.1 Hypothesis 1

The Value Profile data presented in Chapter 5 showed consistently that Value Reuse is prevalent in Instruction Executions and Memory Accesses throughout the execution of the benchmarks tested. This was shown to be the case on both the LLVM and x86 architectures. Therefore, it is concluded that Hypothesis 1 is likely to be correct.

8.2 Hypothesis 2

The Value Reuse Cache presented in Chapter 6 was successfully able to exploit Value Reuse in Instruction Executions at the Global-level more than at the Local-level. This does not necessarily show that there is a greater level of Value Reuse at the Global-level than at the Local-level - it could be that this specific design of a Value Reuse Cache is more apt to exploit Global-level Value Reuse that Local-level Value Reuse. However, (Yi & Lilja, 2001) showed that there is more potential for Value Reuse at the Global level than at the Local level. Therefore, it it considered more likely that Hypothesis 2 is correct, as the results of using the Value Reuse Cache do not contradict Hypothesis 2.

8.3 Hypothesis 3

The Value Profile data presented in Chapter 5 showed consistently that there is a greater level of Value Reuse in Memory Accesses at the Global-level than at the Local-level. This was shown to be the case on both the LLVM and x86 architectures. Therefore it is concluded that Hypothesis 3 is likely to be correct.

8.4 Hypothesis 4

The Value Profile Data presented in Chapter 5 showed that the benchmarks with a high level of Value Reuse in Instruction executions were also the benchmarks with high levels of Value Reuse in Memory Accesses. This was observed on both LLVM and the x86 architecture. Therefore, it is considered that this hypothesis is likely to be correct.

8.5 Hypothesis 5

The Value Reuse Caches tested in Chapter 6 had a small, though reasonable hit rate for several of the benchmarks tested. It was shown in (Sodani & Sohi, 1997) that an Instruction Reuse Cache is able to improve performance by allowing the bypass of instruction executions. If the Value Reuse Caches tested were implemented in hardware, it is likely that it would also allow the bypass of some instruction executions when a cache hit occurs. Therefore, it is considered likely that this hypothesis is correct.

8.6 Hypothesis 6

It was shown in Chapter 7 that there is a very high level of Value Reuse in Memory Accesses when a cache is implemented. Based on this, it has been shown that it is possible to design a scheme which aims to reduce power consumption due to Value Reuse. It has been shown that this scheme can reduce power consumption in an example scenario. Through simple inspection of the scheme, it does not appear that there is any situation in which the scheme would increase power consumption. However, further testing would be needed to validate the effects of the scheme. Since other in other works it has been shown that it is possible to reduce power consumption by exploiting Value Reuse (Yang & Gupta, 2002) and (Yang *et al.*, 2004), it is considered likely that it is possible to develop a scheme which exploits Value Reuse to decrease power consumption, even if the previously proposed scheme were found to be ineffectual. Therefore it is considered likely that this hypothesis is correct.

8.7 Hypothesis 7

The Value Profile data presented in Chapter 5 showed that the Value Profile data gathered on LLVM is not representative of Value Profile data on the x86 architecture. It was shown for both Instruction Execution and Memory Access Value Profiles, the amount of Value Reuse present in LLVM may either be greater than or less than on the x86 architecture dependent on the benchmark. Therefore, it is not possible to predict the amount of Value Reuse on the x86 architecture using information about the amount of Value Reuse on LLVM.

It may be the case that because there are substantial differences between the two architectures, the Value Profile data of one of these architectures is not representative of the Value Profile data of the other. It is possible that Value Profile data from another architecture (e.g. ARM, or MIPS) may be better represented by the Value Profile data on LLVM. However, this has not been tested as it is outside the scope of this project.

It is necessary to conclude that Hypothesis 7 is incorrect, as only evidence which contradicts this hypothesis has been found. This hypothesis may be modified to state "As the LLVM IR is architecture independent, value profile data collected by executing a particular program using the LLVM interpreter is representative of its execution on certain specific architectures". Testing may find evidence in support of the modified hypothesis on certain architectures.

Chapter 9

Evaluation

9.1 Evaluation of Pin and LLVM as Platforms for Value Profiling

In this section, the use of LLVM and Pin as platforms to develop tools to investigate Value Profiling will be evaluated. There are several areas in which comparisons will be made.

Architecture Independence. LLVM IR code is architecture independent. It was originally hypothesised (Hypothesis 7) that the Value Profile data gathered on LLVM would be representative of all architectures. However, it has been found that this is not the case with the x86 architecture. Pin is partially architecture independent, in that it is supported on the IA-32, IA-64 and ARM architectures. However, Value Profile data gathered using Pin is not expected to be representative of all architectures, as the code which is instrumented will be executed on one of the three specific architectures which Pin supports. An additional argument against Pin being considered architecture independent is that there are many architecture-specific functions in the Pin API which may be necessary to perform certain tasks. The modified implementation of Pinatrace, and the cache simulator only use architecture-independent API functions and therefore could be used on any of Pin's three supported platforms. However, the tools for Value Profiling of Instruction Executions, and the implementation of the Value Reuse Cache use several functions specific to the IA-32 architecture, and therefore would need to be modified before they could be used on other architectures.

Performance. The LLVM interpreter has been instrumented to produce Value Profile data. Although LLVM has the capability to JIT compile code to a native ISA before execution to improve the performance of executed code, using this method would bypass the instrumentation code, and therefore suppress the recording of Value Profile data. An alternative to instrumenting the interpreter would be to write a pass which instruments specific sections of the code before it is JIT compiled. However, investigation of this method of Value Profiling has not been included in the scope of the project. The disadvantage of the instrumentation code being part of the interpreter is that the interpreter runs very slowly compared to natively compiled code. Execution of a single benchmark for all 20 datasets often requires up to 12 hours. By contrast, Pin dynamically instruments native code. As the code is running natively, its execution is much faster. Typically, a benchmark which would require 12 hours to complete execution on LLVM can be completed in under an hour when run natively and instrumented using Pin.

Scope of Instrumentation. Code in the standard libraries is not executed within the LLVM Interpreter. Instead, the interpreter calls the library functions itself via its wrapper functions (see Section 4.7). As a result, code which executes which is part of a library is not profiled. Pin is able to instrument all of the code which executes. When control is passed inside a library function, Pin continues to perform instrumentation in the same manner as when control is inside the main executable. Therefore, the Value Profile data gathered using Pin is more representative of the whole execution of the benchmark.

Ease of modification. There is no evidence that the LLVM Interpreter was designed with profiling of any type in mind. Although the interpreter is straightforward to understand and modify, it is sometimes difficult to identify a natural location to insert instrumentation code. Additionally, when writing Value Profiling code, consideration had to be made for the fact that there are several namespaces spread across several source files which make up the LLVM Interpreter binary, `lli`. Also, there are many portions of code throughout LLVM which must be understood in order to begin to write Value Profiling code - examples of these portions include understanding the GenericValue and APInt classes, which are different ways of representing values in memory. The Pin API appears daunting at first sight. However, once some experience is gained with using the API, it is often straightforward to modify any of the existing tools (of which there are many) or write a tool from scratch to meet specific goals. The Pin API is relatively consistent at all levels that instrumentation and inspection is supported at. Another benefit of Pin is that that different code to perform different functions can easily be isolated from each other, as a separate PinTool can be written to perform each function. This allows the code to perform profiling to be kept simple, and is less likely to become confusing. Writing additional profiling code for LLVM is relatively untidy compared to writing code to perform similar functions using Pin. All profiling code is mixed with the code which simulates execution in the LLVM Interpreter. Additionally, if several different profiling functions are to be added, these must all be added to the same source code and controlled by command line switches. This leads to source files becoming large and complicated, and there is no well-defined boundary between portions of code which perform different functions. In conclusion, after the initial steep learning curve, it is easier to use Pin to develop Value Profiling tools.

Reliability. The LLVM Interpreter is only able to execute programs which make use of library functions for which it already has wrapper functions implemented (Section 4.7). Additionally, it is not uncommon for the (uninstrumented) interpreter to terminate with failed assertions repeatedly whilst executing specific executables. This includes several benchmarks which could not be profiled because of this problem, (e.g. the *security-blowfish* benchmarks, and *security-rijndael-e*). Throughout the execution of all of the Value Profiling tools using Pin, no crashes or failures have been encountered, except where a bug existed in the PinTool.

In conclusion, using Pin to develop Value Profiling tools has the advantages that it is faster, can produce a more representative Value Profile, is easy to develop for, and is more reliable than the LLVM Interpreter. The argument that LLVM can produce an architecture independent Value Profile is unlikely to be correct. Therefore, it is clear that Pin is the more preferable platform to develop Value Profiling tools.

9.2 Evaluation of the Execution of the Project

Throughout the course of the project, several problems have been encountered. Additionally, changes were made to the aims and objectives listed in the Terms of Reference.

9.2.1 Learning C++

At the start of the project, I did not know C++, and was unaware of the Standard Template Library (STL). When I originally allocated time for each objective of the project, I assumed that I could pick up C++ instantly, and would have to spend very little time learning the features of the language. This assumption was based on my previous knowledge of C and Java, and I assumed C++ would be very much like a combination of the two. However, I had to spend time learning concepts which are not present in either language, and understanding the STL in order to use it effectively.

At the time that the interim report was written, I had gained some proficiency in using C++. The majority of this experience had come from making modifications to the LLVM Interpreter. Since the interim report was written, I have further developed this ability. I have become more familiar with using the C++ STL, and am comfortable in using the various commonly-used data structures it provides, e.g. Sets, Maps, and Vectors etc. This extra experience has come from using the Pin API to develop PinTools

from scratch. Writing tools from scratch has allowed me to consider the design of a program in C++, and additionally has reinforced concepts familiar to me from C, such as function pointers.

9.2.2 A Lack of Support for External Library Functions in the LLVM Interpreter

This characteristic of LLVM (described in detail in Section 4.7), was unforeseen at the start of this project. This caused the project to be delayed by approximately two weeks. Part of this time was used in determining why none of the benchmarks would run under the interpreter, and understanding the structure of the wrapper functions. Once these two tasks had been completed, the problem of how to determine which functions to implemented had to be tackled. Once the correct functions were determined, the new wrapper functions were implemented and tested. A lesson learned from this episode was that time should always be put aside to work around unforeseen problems. In this case, there was enough slack space in the project schedule to make time to solve this problem.

9.2.3 Removal of Basic-Block Value Profiling from Aims and Objectives

Some work has already been done in this area by other researchers (Huang & Lilja, 2003). At the start of the project, replicating the work done in this paper appeared to be achieveable. However, it was soon found that the amount of work to be completed in developing and applying Value Profiling at the instruction level would not leave enough time to investigate Value Profiling of basic blocks. However, Value Profiling of Memory Accesses was investigated instead, which provided a manageable extra workload, and generated results which could be more easily related to the results of Instruction level Value Profiling.

9.2.4 Introduction of Pin as a Platform for Value Profiling

At the start of the project, it was intended that all Value Profiling would be done using LLVM. However, in January it was proposed by the project supervisor that investigation into Value Profiling using Pin may provide a useful alternative with which to compare and contrast LLVM. Because of the experience gained in C++ from initial work with LLVM, it was possible to quickly develop Pin Value Profiling tools. Additionally, the execution of benchmarks was much faster using Pin, so results could be gathered more quickly. It could be said that the development and application of the Value Profiling tools using Pin was more successful than development on LLVM, in terms of the time taken to produce results.

9.3 Conclusion

The project has provided useful output in the form of the tools to perform Value Profiling, and the Value Profile data. The Value Profile data has been useful as it has been shown that it can be used to guide the design of a low-power bus. Additionally it has been shown that because Value Reuse is prevalent throughout execution of programs, schemes to exploit Value Reuse (such as a Value Reuse Cache) may be successfully developed. However, these developments in Value Profiling and its applications, although an end in themselves, are not an end result; there is much further work to be done based on these initial investigations into Value Profiling.

Chapter 10

Further work

10.1 Investigation into Precomputation Tables

Precomputation tables were investigated in (Yi *et al.*, 2002). However, the training was only done on one data set, and the testing was done on another. An implementation should be tested with many data sets - e.g. the 20 data sets for each MiBench benchmark. It is expected that training for some data sets will perform very poorly with other data sets. This hypothesis should first be tested. If this is confirmed, then it is suggested that an algorithm to train from multiple data sets should be used to create a precomputation table which is more general to all the data sets rather than being well-fitted for a particular set and ill-fitted to all the others. An algorithm which combines the data sets has been devised, which is as follows:

```
N = Desired size of precomputation table
i = 1
PCTSize = 0
PC = Empty Precomputation Table
T = space to hold operations from all data sets

while (PCTSize<N)
  Load top i operations from all data sets into T
  Finished = false
  while (!Finished)
    F =  most frequently occurring unmarked operation in T
    Mark all F in T
    if(F is present in >25% datasets in T or i > 2*N)
      if(F does not exist in PC Table)
        insert F into PC Table
        PCTSize++
      endif
    else
      Finished = true
    endif
  endwhile
endwhile
```

Further work in this area would involve testing and modifying this algorithm. Potential further developments in this area could lead to schemes which allow the bypass of the execution of frequent computations without the complexity of managing entries present in the Value Reuse Cache.

10.2 Examination of the Distribution of Frequent Values in Memory

The work examining the difference in Value Reuse in Memory Accesses at the global and local levels suggests that most programs store frequently transferred values in many different locations throughout the memory. Further investigation in this area may involve experiments to determine the distribution of these values in memory. The outputs of this work may be useful for guiding the design of memory chips with optimisations which exploit the distribution of these values, perhaps to decrease power consumption or increase memory density.

10.3 Testing the Memory Bus Power Reduction Scheme

The scheme presented in Chapter 7 is designed to reduce power consumption by reducing switching activity on the bus. However, the scheme is not tested in this project. Further investigation in this area may involve developing a software simulation of the scheme to estimate the reduction in power consumption. Other work may involve performing Value Profiling of Memory Accesses on other architectures (e.g. ARM, MIPS etc) to determine if the scheme would appear to substantially reduce switching activity on the memory bus of architectures other than the x86. This further work could lead to refinement of the scheme to reduce power consumption.

10.4 Refinement of the Value Reuse Cache

The Value Reuse Cache presented in Chapter 6 was able to exploit Value Reuse well within the cacheable instructions, but did not provide high levels of cache hits over all instruction executions. Further work on the Value Reuse Cache may involve porting the simulation to another architecture, where the overall hit rate may be higher. Alternatively, the number of instructions supported by the Value Reuse Cache on the x86 architecture could be increased to try to increase the overall hit rate of the cache. This further work could lead to a design for a Value Reuse Cache being developed which is able to increase performance substantially by allowing the bypass of a significant amount of instruction executions.

Appendices

Appendix A

Terms of Reference

A.1 Project Background

Throughout the execution of a program, it can be demonstrated that a small set of values can occur in a large proportion of its memory. Additionally, a large proportion of instructions are repeatedly executed with the same operands, computing the same result multiple times. (Yang & Gupta, 2002) showed that only eight values occupy up to 48% of the memory locations throughout the execution of particular benchmarks from the SPEC95 suite. (Yi & Lilja, 2001) showed that throughout the execution of particular SPEC95 and SPEC2000 benchmarks, less than 10% of input sets (made up of an instructions opcode, operands and program counter) make up more than 65% of all instructions executed.

This repetition of computations and loads/stores to memory locations can be exploited to increase performance. (Kumar, 2003) describes three schemes in which the repeated computation of the same results can be exploited. Two of these use value profile information gathered by executing an instrumented program, and recording the most frequent values. Additional code is then inserted into the program source, which provides special cases for the values which occur frequently.

The Low Level Virtual Machine (LLVM) (Lattner & Adve, 2004a) compiler infrastructure provides an ideal framework for experimenting with value profiling, as it provides an interpreter for the LLVM bytecode, which can easily be instrumented to profile the execution of instructions.

A suitable set of benchmarks are required to execute on the LLVM interpreter for development and testing. All of the components of the MiBench suite of benchmarks are available as C source code, and provide a wide variety of different workloads (Guthaus *et al.*, 2001). This makes it suitable for developing and testing value profiling within LLVM. MiDataSets is an additional set of input data for the MiBench suite (Fursin *et al.*, 2007), which will be used to provide additional datasets.

This project was chosen due to an interest in computer organisation and compilers.

A.2 Aims

This project will involve the modification of the LLVM interpreter to investigate value re-use in arithmetic and logical operations at the instruction level.

Should the project succeed in meeting the first aim, the potential for value re-use at the basic block level will be investigated.

A.3 Objectives

1. To review and critically evaluate different approaches to value profiling at the instruction and basic block level via a literature survey. The knowledge gained from the literature survey will be applied later during the development of value profiling code.

2. To develop and enhance an instrumented version of the LLVM interpreter to provide value profile information for the execution of arithmetic and logical operations. This will be done by inserting

code into the functions which perform these operations. The inserted code will store the operands in a Multiset from the C++ Standard Template Library (STL) ¡ref¿.

3. To evaluate the potential for value reuse optimisation across the MiBench suite of applications using MiDataSets.

4. To develop a compiler-controlled scheme for managing the contents of a value reuse cache in the the LLVM interpreter. The design of the scheme will be guided using the information from objectives 2 and 3.

5. A cache to store frequently used values will then be implemented in the LLVM interpreter, the parameters of which (number of entries, how the entries are managed etc.) will be chosen for the optimum amount of value reuse, guided using the value profile information from earlier.

6. The effect of this cache will be evaluated by running the same benchmarks through the modified interpreter, and analysing the amount of value reuse. This information could be used to speculate what the speed increase would be, if the value reuse cache were implemented in a real processor.

7. If the value reuse at the instruction level is successful, The LLVM interpreter will then be instrumented to profile values at the basic block level. The data will be recorded in a similar way to the values at the instruction level, with appropriate modifications.

8. A value reuse cache will be developed for basic blocks. This cache will also be evaluated to determine its potential to increase the speed of execution of the benchmarks.

A.4 Deliverables

- A modified version of the LLVM compiler infrastructure which has been developed to transform code using value profiling information.

- An interim report and a project report. The interim report will state progress of the project, and if necessary, any changes that are required to be made to the plan of the project. The project report will include the details of the research and development of the value reuse caches. An evaluation of the software product, and an evaluation of the project itself will be included.

A.5 Resources

- A PC or laptop with Linux and standard development tools (gcc etc.).

- Access to Journals.

Appendix B

Test Cases

Test cases which have been developed to ensure the correct operation of the Value Profiling code developed are as follows:

Test case 1. This is a minimal test case which implements a single Add instruction, which adds the integers 1 and 2. Minimal code for this case is as follows:

```
int main() {
        int a = 1, b = 2, c = a + b;
}
```

Test case 2. This implements two Add instructions, one which adds 1 and 2, and another which adds 2 and 1. This is intended to test that the code written recognises the commutativity of the operands of the Add instruction. Minimal code for this case is:

```
int main() {
        int a = 1, b = 2;
        int c = a + b;
        c = b + a;
}
```

Test case 3. This implements two Sub instructions, one which subtracts 1 from 2, and another which subtracts 2 from 1. This is intended to test that the code written recognises that the operands of the Sub instruction are not commutative. Minimal code for this case is:

```
int main() {
        int a = 1, b = 2;
        int c = a - b;
        c = b - a;
}
```

Test case 4. This implements a loop. The loop counter will be incremented once per iteration of the loop. No other operations are performed in this test. Minimal code for this case is:

```
int main() {
        int i;
        for(i=0; i<10; i++);
}
```

Test case 5. This implements two Add instructions, one with integer operands, and the other with floating-point operands. This is designed to test that the code does distinguish between instructions with operands with the same numeric value but with different types. Minimal code for this case is:

```
int main() {
```

```
        int a=1, b=2, i = a+b;
        float c=1.0f, d=2.0f, j = c+d;
}
```

Test case 6. This implements one Add instruction, with operands of type double. This is to test that the code written recognises the operands as being of the double type. Minimal code for this case is as follows:

```
int main() {
        double a=1.0, b=2.0, c=a+b;
}
```

Test case 7. This implements one Mul instruction, with integer operands. Minimal code for this case is:

```
int main() {
        int a=1, b=2, c=a*b;
}
```

Test case 8. This implements one Mul instruction, with operands of type double. Minimal code for this case is:

```
int main() {
        double a=1.0, b=2.0, c=a*b;
}
```

Test case 9. This implements one Sub instruction, with operands of type double. Minimal code for this case is:

```
int main() {
        double a=1.0, b=2.0, c=a-b;
}
```

Test case 10. This implements one floating-point division operation, with operands of type double. Minimal code for this case is:

```
int main() {
        double a=1.0, b=2.0, c=a/b;
}
```

Test case 11. This implements an addition repeatedly from the same location. The purpose of this is to test local level instruction profiling. The repeated additions of 1 and 2 should all have the same program counter/reference value in the profile output. The loop counter increment will also appear in the profile, and these operations should have a different program counter/reference value to the other add instructions. Minimal code for this case is:

```
int main() {
        int a = 1;
        int b = 2;
        int c, d;
        for(d=1; d<3; d++)
                c = b + a;
}
```

Test case 12. This implements two memory stores. The first memory store of the value 1 will have an address 20 (5×4) bytes before the second store of the value 39. This is due to the integer operands being 4 bytes long, and the second store is 5 locations later in memory than the first one. Within LLVM, the GETELEMENTPTR instruction will be used to calculate the address of the location to store the values. Minimal code for this case is as follows:

```
#include <stdlib.h>
int main() {
        int *i;
        i = malloc(sizeof(int[10]));
        i[0] = 1;
        i[5] = 39;
}
```

Test case 13. This is as the previous test case. However, loads back from memory are also implemented. The purpose of this is to test that the memory profiling code correctly distinguishes between a load and a store. Minimal code for this case is:

```
#include <stdlib.h>
int main() {
        int *i, a, b;
        i = malloc(sizeof(int[10]));
        i[0] = 1;
        i[5] = 39;
        a = i[0];
        b = i[5];
}
```

Test case 14. This implements a nested loop. The outer loop executes many times. The counter for the inner loop will repeatedly perform the same additions. This case is to test the Value Reuse Cache over a long number of instructions. Minimal code for this case is:

```
int main() {
        int a=5, b=7, c, i, j;
        for(i=0; i<10000; i++)
                for(j=0; j<2; j++)
                        c = a & i;
}
```

Test case 15. This implements a loop which performs three operations. The purpose of this is to test the eviction policy of the Value Reuse Cache. Very small value reuse caches (< 3 entries) will not be able to store all of the instructions which are repeated, so will show a lower hit rate than caches of size greater than 3. Minimal code for this case is as follows:

```
int main() {
        int a=5, b=7, c, i;
        for(i=0; i<5; i++) {
                c = a & b;
                c = a | b;
                c = a ^ b;
        }
}
```

Test case 16. This is similar to the previous case. However, an inner loop is included. The effect of this is that for each iteration of the outer loop, more entries will be required to cache all the different

computations which take place. This provides a further test for the eviction policy, as slightly larger cache sizes than 3 should may also have to evict a potentially reusable value. Minimal code for this case is:

```
int main() {
        int a=5, b=7, c, i, j;
        for(i=0; i<10; i++)
                for(j=0; j<2; j++){
                        c = a & i;
                        c = a | i;
                        c = a ^ i;
                }
}
```

Test case 17. This test case implements two add instructions which perform the same operation, at different locations. This is to test the local level instruction profiler and value reuse caches. At global level, there should be one cache hit, and two of the same instruction recorded. At local level, there should be no cache hits and a single execution of two different (by the program counter) instructions. The operands will be the same for both of these instructions. Minimal code for this case is as follows:

```
int main() {
        int a=2, b=3, c, d;
        c = a + b;
        d = a + b;
}
```

Test case 18. This test case implements a store to a single pointer value. The purpose of this is to test the implementation of the GETELEMENTPTR profiling. Minimal code for this case is:

```
#include <stdlib.h>
int main() {
        int *a = malloc(sizeof(int));
        *a=1;
}
```

Test case 19. This test case implements a store into an array. The purpose of this is to test the implementation of the GETELEMENTPTR profiling. Minimal code for this case is:

```
#include <stdlib.h>
int main() {
        int *a = malloc(sizeof(int[4]));
        a[2]=1;
}
```

Test case 20. This test case implements two stores into different elements of a struct. This is to test the implementation of the GETELEMENTPTR profiling. Minimal code for this case is:

```
struct lev1 {
        int a;
        int b;
};

int main() {
        struct lev1 d;
        d.a = 2;
        d.b = 5;
}
```

Test case 21. This test case implements a store into a location into a struct, and into the same place accessing the struct as if it were an array. Although this is not a technique which would really be used in a program, it is done to test that the GETELEMENTPTR profiling correctly distinguishes between struct access and array access. The compiler generates a warning about this code, which is expected. Minimal code for this case is as follows:

```
struct lev1 {
        int a;
        int b;
};

int main() {
        struct lev1 d;
        int *c = &d; // c points to d
        d.b = 5; // Access struct through element b
        c[1] = 5; // Access the same memory location through array index access
}
```

Appendix C

CD Contents

C.1 Report and JISC Originality Report

In the `report` folder, an electronic version of this report in PDF format and the JISC Originality report can be found.

C.2 Value Profiling Tools

In the `tools` folder, the Value Profiling tools which were used to record Value Profile data may be found.

C.2.1 LLVM

In the `llvm` subfolder, an archive containing the full source to the modified LLVM interpreter is found (`llvm-2.1.tar.gz`). This can be built by untaring the archive and running `./configure` and `make` in the `llvm-2.1` directory. Additionally, each of the source files which were modified are also in this subfolder, so that it is easy to see which files were modified.

The LLVM interpreter is called `lli` and will be built in the `Release/bin` folder. The interpreter can be used for Value Profiling by executing:

```
./lli --force-interpreter [-inst-profile|-mem-profile] [-prof-buf-size=<size>]
      -prof-level=[local|global] -vpcsv=<file> <program name>
```

The `-force-interpreter` option is mandatory for Value Profiling. One of the `-inst-profile` or `-mem-profile` options may be used. These enable Value Profiling of Instruction Executions or Value Profiling of Memory Accesses respectively. The `-prof-buf-size` option is used to specify the size of the Value Profile Buffer. This defaults to 100000 and may be omitted - the default size has been found to be a reasonable value and used when recording all the Value Profile data presented in this report. The `-prof-level` option is to specify whether Local-level Value Profiling or Global-level Value Profiling is performed. The `-vpcsv` option is for specifying the output Value Profile data file name, which is `profile.csv` by default.

C.2.2 Pin

In the `pin` subfolder, an archive containing the standard distribution of Pin, and the PinTools for Value Profiling is found (`pin-2.3-16358-gcc.4.0.0-ia32e-linux.tar.gz`). This can be built by untaring the archive and running `make` in the `pin-2.3-16358-gcc.4.0.0-ia32e-linux` directory. Additionally the source files to each of the modified and new PinTools is also found in this folder.

Value Profiling of Memory Accesses

To use Pin to perform Value Profiling of Memory Accesses, change to the `Bin` folder and use the following command:

```
./pin -t ../SimpleExamples/pinatrace [-o <file>] [-localprof] -- <binary name>
```

The `-o` option specifies the output file name, which defaults to `pinatrace.out`. The `-localprof` option turns on Local-level Value Profiling of Memory Accesses. By default Global-level Value Profiling of Memory Accesses is performed. The binary name should be a full or relative path to a binary, e.g. `/bin/ls`.

Value Profiling of Instruction Executions

To use Pin to perform Value Profiling of Instruction Executions, change to the `Bin` folder and use the following command:

```
./pin -t ../SimpleExamples/vrprofile [-o <file>] [-bufsize <size>]
      [-localvr] -- <binary name>
```

The `-o` option specifies the output file name, which defaults to `profile.csv`. The `-localvr` option turns on Local-level Value Profiling of Instruction Executions. By default Global-level Value Profiling of Instruction Executions is performed. The `-bufsize` option is used to specify the size of the Value Profile Buffer. This defaults to 100000 and may be omitted - the default size has been found to be a reasonable value and used when recording all the Value Profile data presented in this report. The binary name should be a full or relative path to a binary, e.g. `/bin/ls`.

Value Reuse Cache

To use Pin to simulate a Value Reuse Cache, change to the `Bin` folder and use the following command:

```
./pin -t ../SimpleExamples/vrcache [-vrcachesize <size>] [-localvr] -- <binary name>
```

The `-localvr` option forces simulation of a Local-level Value Reuse Cache. By default a Global-level Value Reuse Cache is simulated. The `-vrcachesize` option is used to specify the size of the Value Reuse Cache. This defaults to 32 entries, and must be greater than 1. The output of the simulator is written to `stderr`, as only a small amount of output is produced. The binary name should be a full or relative path to a binary, e.g. `/bin/ls`.

Value Profiling of Memory Accesses with Cache Simulator

To use Pin to perform Value Profiling of Memory Accesses in conjunction with the instruction/data cache simulator included with Pin, change to the `Bin` folder and use the following command:

```
./pin -t ../Memory/cacheprof [-o <file>] -- <binary name>
```

The `-o` option specifies the output file name, which defaults to `profile.csv`. The binary name should be a full or relative path to a binary, e.g. `/bin/ls`.

C.3 Test cases

Source code and binaries compiled to LLVM and the x86 architecture of all the test cases are found in the `tests` folder. The outputs of the LLVM interpreter when each of the test cases are executed are found in the `llvm-results` subfolder. The outputs of the Pin Value Profiling tools when executing each of the test cases are found in the `pin-results` subfolder. The outputs of the Value Reuse Cache Simulator when executing the test cases are found in the `vrc-results` subfolder.

C.4 Value Profile Data

All the Value Profile data which was recorded throughout the course of the project is included in the `profiledata` folder. The full data is not present, as this is around 100GB. Instead, each Value Profile data file has been truncated so that only the top 32 entries remain. This requires a fraction of the original space, but still allows the reader to see the most frequent values.

C.4.1 LLVM Value Profile Data

The Value Profile data recorded using LLVM is in the `llvm` subfolder. A further three subfolders are contained within this folder:

- `inst-global` - Global-level Instruction Value Profile data.
- `mem-global` - Global-level Memory Access Value Profile data.
- `mem-local` - Local-level Memory Access Value Profile data.

C.4.2 Pin Value Profile Data

The Value Profile data recorded using Pin is in the `pin` subfolder. A further six subfolders are contained within this folder:

- `cacheprof` - Value Profiling of Memory Accesses with Cache Simulator data.
- `inst-global` -Global-level Instruction Value Profile data.
- `mem-global` - Global-level Memory Access Value Profile data.
- `mem-local` - Local-level Memory Access Value Profile data.
- `vrc-global` - Global-level Value Reuse Cache data.
- `vrc-local` - Local-level Value Reuse Cache data.

C.5 MiBench Benchmarks and Midatasets Datasets

In the folder `benchmarks`, the MiBench benchmarks of which Value Profile data was recorded is included. Inside the `src` subfolder of each benchmark, the binaries compiled to the x86 and LLVM are to be found. Additionally in each `src` folder, the following scripts will be found:

- `run-llvm-inst-global` - Runs Global-level Value Profiling of Instruction Executions on LLVM for all 20 datasets for the benchmark.
- `run-llvm-mem-global` - Runs Global-level Value Profiling of Memory Accesses on LLVM for all 20 datasets for the benchmark.
- `run-llvm-mem-local` - Runs Local-level Value Profiling of Memory Accesses on LLVM for all 20 datasets for the benchmark.
- `run-pin-inst-global` - Runs Global-level Value Profiling of Instruction Executions on the x86 for all 20 datasets for the benchmark.
- `run-pin-mem-global` - Runs Global-level Value Profiling of Memory Accesses on the x86 for all 20 datasets for the benchmark.
- `run-pin-mem-local` - Runs Local-level Value Profiling of Memory Accesses on the x86 for all 20 datasets for the benchmark.
- `run-pin-vrc-global` - Runs a simulation of the Global-level Value Reuse Cache on the x86 for all 20 datasets for the benchmark.
- `run-pin-vrc-local` - Runs a simulation of the Local-level Value Reuse Cache on the x86 for all 20 datasets for the benchmark.
- `run-pin-mem-global` - Runs Value Profiling of Memory Accesses in conjunction with the cache simulator on the x86 for all 20 datasets for the benchmark.

These scripts all assume that `pin` and `lli` are in the current path.

References

Allen, Todd. 2008. *CPUID - A Linux tool to dump x86 CPUID information about the CPU(s).* http://www.etallen.com/cpuid.html.

Burger, Doug, & Austin, Todd M. 1997. *The SimpleScalar Tool Set, Version 2.0.*

Calder, Brad, Feller, Peter, & Eustace, Alan. 1997. Value Profiling. *micro*, **00**, 259.

Chen, Guangyu, Kandemir, Mahmut, & Irwin, Mary J. 2005. Exploiting frequent field values in java objects for reducing heap memory requirements. *Pages 68–78 of: VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments.* New York, NY, USA: ACM.

Criswell, John, Lattner, Chris, Brukman, Misha, Adve, Vikram, & Shi, Guochun. 2008. *Getting Started with the LLVM System.* http://www.llvm.org/docs/GettingStarted.html.

Feller, Peter. 1998. *Value profiling for instructions and memory locations.*

Fursin, Grigori, Cavazos, John, O'Boyle, Michael, & Temam, Olivier. 2007 (January). MiDataSets: Creating The Conditions For A More Realistic Evaluation of Iterative Optimization. *In: Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2007).*

Gabbay, Freddy, & Mendelson, Avi. 1997. Can program profiling support value prediction? *Pages 270–280 of: MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture.* Washington, DC, USA: IEEE Computer Society.

Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., Mudge, T., & Brown, T. 2001 (December). MiBench: A free, commercially representative embedded benchmark suite. *Pages 3–14 of: 4th IEEE International Workshop on Workload Characteristics.*

Handy, Jim. 1998. *The Cache Memory Book.* Academic Press Ltd.

Hewlett-Packard, Company. 1994. *Standard Template Library Programmer's Guide.* http://www.sgi.com/tech/stl/.

Huang, J. 2000. *Improving Processor Performance through Compiler Assisted Block Reuse.*

Huang, Jian, & Lilja, David J. 2003. Balancing Reuse Opportunities and Performance Gains with Subblock Value Reuse. *IEEE Transactions on Computers*, **52**(8), 1032–1050.

Intel, Corporation. 2007. *IA-32 Intel Architecture Software Developer's Manual.*

Johnson, Neil E. 2004. *Code size optimization for embedded processors.* Ph.D. thesis, University of Cambridge.

Knaggs, P., & Welsh, S. 2004. *ARM Assembly Language Programming.*

Kumar, K. V. Seshu. 2003. Value reuse optimization: reuse of evaluated math library function calls through compiler generated cache. *SIGPLAN Not.*, **38**(8), 60–66.

Lattner, Chris, & Adve, Vikram. 2004a (Mar). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. *In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04).*

Lattner, Chris, & Adve, Vikram. 2004b (Sep). The LLVM Compiler Framework and Infrastructure Tutorial. *In: LCPC'04 Mini Workshop on Compiler Research Infrastructures.*

Lattner, Chris, & Adve, Vikram. 2008. *LLVM Language Reference Manual.* http://www.llvm.org/docs/LangRef.html.

Lattner, Chris, Dhurjati, Dinakar, Stanley, Joel, & Spencer, Reid. 2008. *LLVm Programmer's Manual.* http://www.llvm.org/docs/ProgrammersManual.html.

Lipasti, Mikko H., Wilkerson, Christopher B., & Shen, John Paul. 1996. Value locality and load value prediction. *Pages 138–147 of: ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems.* New York, NY, USA: ACM.

Luk, Chi-Keung, Cohn, Robert, Muth, Robert, Patil, Harish, Klauser, Artur, Lowney, Geoff, Wallace, Steven, Reddi, Vijay Janapa, & Hazelwood, Kim. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Pages 190–200 of: PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation.* New York, NY, USA: ACM.

Luk, Chi-Keung, Cohn, Robert, Muth, Robert, Patil, Harish, Klauser, Artur, Lowney, Geoff, Wallace, Steven, Reddi, Vijay Janapa, & Hazelwood, Kim. 2008. *The Pin FAQ.* http://rogue.colorado.edu/Wikipin/index.php/FAQ.

Parsons, Thomas W. 1992. *Introduction to Compiler Construction.* Computer Science Press.

Rotenberg, Eric, Bennett, Steve, & Smith, James E. 1996. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. *Pages 24–35 of: International Symposium on Microarchitecture.*

Sodani, Avinash, & Sohi, Gurindar S. 1997. Dynamic Instruction Reuse. *Pages 194–205 of: ISCA.*

Spencer, R. 2008. *The Often Misunderstood GEP Instruction.* http://www.llvm.org/docs/GetElementPtr.html.

Whiteley, David. 2004. *Introduction to Information Systems.* Palgrave Macmillan.

Yang, Jun, & Gupta, Rajiv. 2002. Frequent value locality and its applications. *ACM Trans. on Embedded Computing Sys.*, **1**(1), 79–105.

Yang, Jun, Gupta, Rajiv, & Zhang, Chuanjun. 2004. Frequent value encoding for low power data buses. *ACM Trans. Des. Autom. Electron. Syst.*, **9**(3), 354–384.

Yi, Joshua, & Lilja, David. 2001. An Analysis of the Potential for Global Level Value Reuse in the SPEC95 and SPEC2000 Benchmarks. *Laboratory for Advanced Research in Computing Technology and Compilers Technical Report No. ARCTiC*, **1**(1).

Yi, Joshua J., Sendag, Resit, & Lilja, David J. 2002. Increasing Instruction-Level Parallelism with Instruction Precomputation (Research Note). *Pages 481–485 of: Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing.* London, UK: Springer-Verlag.