THE UNIVERSITY *of* EDINBURGH

Customising Compilers for Customisable Processors

Alastair Colin Murray

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2011

Abstract

The automatic generation of instruction set extensions to provide application-specific acceleration for embedded processors has been a productive area of research in recent years. There have been incremental improvements in the quality of the algorithms that discover and select which instructions to add to a processor. The use of automatic algorithms, however, result in instructions which are radically different from those found in conventional, human-designed, RISC or CISC ISAs. This has resulted in a gap between the hardware's capabilities and the compiler's ability to exploit them.

This thesis proposes and investigates the use of a high-level compiler pass that uses graph-subgraph isomorphism checking to exploit these complex instructions. Operating in a separate pass permits techniques to be applied that are uniquely suited for mapping complex instructions, but unsuitable for conventional instruction selection. The existing, mature, compiler back-end can then handle the remainder of the compilation. With this method, the high-level pass was able to use 1965 different automatically produced instructions to obtain an initial average speed-up of 1.11x over 179 benchmarks evaluated on a hardware-verified cycle-accurate simulator.

This result was improved following an investigation of how the produced instructions were being used by the compiler. It was established that the models the automatic tools were using to develop instructions did not take account of how well the compiler could realistically use them. Adding additional parameters to the search heuristic to account for compiler issues increased the speed-up from 1.11x to 1.24x. An alternative approach using a re-designed hardware interface was also investigated and this achieved a speed-up of 1.26x while reducing hardware and compiler complexity.

A complementary, high-level, method of exploiting dual memory banks was created to increase memory bandwidth to accommodate the increased data-processing bandwidth provided by extension instructions. Finally, the compiler was considered for use in a non-conventional role where rather than generating code it is used to apply source-level transformations prior to the generation of extension instructions and thus affect the shape of the instructions that are generated.

Acknowledgements

I would like to thank my supervisor, Björn, for his help with guiding the direction of the research that eventually resulted in this thesis.

I would like to thank Nigel for helping to get me involved in the PASTA project.

Everyone involved in the PASTA project deserves a mention, but those who were involved with *ArcSim* especially – my research wouldn't have been possible without it.

Within PASTA Richard deserves a special mention, for all the parts of the pie-g toolchain he created, especially *ISEGen*– another tool that my research wouldn't have been possible without.

Finally, I would like thank my proof readers: Claire and Colin.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following papers:

- Richard V. Bennett, Alastair C. Murray, Björn Franke, and Nigel Topham. Combining source-to-source transformations and processor instruction set extensions for the automated design-space exploration of embedded systems. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '07)*, pages 83–92, June 2007.
- Alastair Murray and Björn Franke. Fast source-level data assignment to dual memory banks. In *Proceedings of the 11th International Workshop on Software and Compilers for Embedded Systems (SCOPES '08)*, pages 43–52, March 2008.
- Alastair C. Murray, Richard V. Bennett, Björn Franke, and Nigel Topham. Code transformation and instruction set extension. *ACM Transactions on Embedded Computing Systems (ACM TECS)*, 8(4):1–31, 2009.
- Oscar Almer, Richard Bennett, Igor Böhm, Alastair Murray, Xinhao Qu, Marcela Zuluaga, Björn Franke, and Nigel Topham. An end-to-end design flow for automated instruction set extension and complex instruction selection based on GCC. In *Proceedings of the First International Workshop on GCC Research Opportunities (GROW '09)*, pages 49–60, January 2009.
- Alastair Murray and Björn Franke. Using genetic programming for source-level data assignment to dual memory banks. In *Proceedings of the 3rd Workshop on Statistical and Machine Learning Approaches to Architectures and Compilation (SMART '09)*, pages 75–89, January 2009.
- Alastair Murray and Björn Franke. Compiling for automatically generated instruction set extensions. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '12)*, April 2012b.
- Alastair Murray and Björn Franke. Adaptive source-level data assignment to dual memory banks. *ACM Transactions on Embedded Computing Systems (ACM TECS)*, 11S(1), June 2012a.

(*Alastair Colin Murray*)

For Claire.

Table of Contents

Chapter 1

Introduction

"I have no data yet. It is a capital mistake to theorise before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts."
— Sherlock Holmes, *fictional character created by Sir Arthur Conan Doyle*, 1859–1930.

The designers of embedded computer systems are under pressure to produce high quality products which meet multiple conflicting constraints: low cost, low power, short time-to-market but still high performance. For embedded computing systems the performance requirement, at a high-level, is usually a fixed target (e.g. a Blu-ray player must be able to decode H.264 video streams and an accompanying audio stream at a specified peak bit-rate). The selection of processors, however, that may be designed, licensed or purchased to meet this performance requirement is huge, and there is frequently an existing processor that will perform sufficiently. For example, a top-of-the-line multi-core Intel processor would be able to perform the tasks of virtually any embedded processor from a performance perspective, but financial cost and power requirements make this uneconomical for an embedded system. Thus, designing an embedded computing system is about meeting the performance requirements for the lowest cost and power requirements, and to do this before your competitors release a similar product.

The advantages of developing a low cost product with a short time-to-market in the highly competitive embedded computing system market are obvious; if you are the first to meet a given performance/power requirement you can capture the market, and once multiple manufacturers have products the lowest-priced item will sell the most. The primary requirement for low power is that portable battery-operated devices must be able to function for significant periods without recharging; a smart-phone with only a few hours of battery life is of little use to anyone. The secondary low power requirement is the need for minimal heat output, a hand-held device must never be more than warm to the touch, must be cooled passively and is frequently sealed in a plastic enclosure. Static embedded systems must also have low heat output (and thus be low power, even when they are connected to mains electricity) as passive cooling is still highly

desirable, e.g. a wireless router is expected to operate silently, noise from fans is undesirable. Consequently, to meet these conflicting requirements the system designers either take standard components and combine them into a Multi-Processor System-on-Chip (MPSoC) and/or design custom pieces of hardware – in both cases they are creating a system specialised for a specific domain or application.

1.1 Specialised Processors

Specialising processors for a particular domain is an effective way of increasing the performance achievable for a given level of power consumption. The most obvious examples of this are the different processor families for different domain areas. Digital Signal Processors (DSPs) are based on VLIW or static superscalar designs (effective on highly data-parallel tasks), with scratchpad memories (effective on streaming data where standard caches are not) and specialised instructions (e.g. a multiply-accumulate, MAC, which are common in DSP tasks). Together these features allow effective digital signal processing (DSP) where highly data-parallel streaming data is processed. Other processor families include Network Processors, or general purpose embedded processors ranging from microcontrollers, used in everything from hard-disk drives to cars to washing machines, up to high performance processors used in portable media players, smartphones, netbooks, etc.

Within most families of embedded processors there are many more design-time configuration options that allow a processor to be tuned to a specific task. As an example, the ARM processor family [ARM, 2011] may be configured with: an optional scratchpad memory; different cache sizes; an optional Memory Management Unit (MMU); different bus interfaces; optional mixed 16/32-bit mode instructions; optional additional DSP instructions; optional floating-point calculation hardware; optional Single-Instruction Multiple-Data (SIMD) units; different process geometries (e.g. $90nm^2$ or $65nm^2$); different clock frequencies; and other more esoteric options. These options can either be thought of as additions to a baseline to improve performance, or as removing unnecessary components from a complete processor to reduce die-area and static power draw. In either case the engineer is tuning the processor to a specific task.

While selecting the correct processor from the correct family and then configuring it appropriately results in a processor well-matched to a task, designing custom hardware can dramatically reduce the power required for the targeted task [Ienne and Leupers, 2007]. Taking this idea to its full extent results in Application Specific Integrated Circuits (ASICs), which are very high performance and low power, but take time and effort to develop so they are neither low cost nor have a short time-to-market. Additionally, once they have been deployed their functionality is set, new features may not be implemented and bugs cannot be fixed. An in-

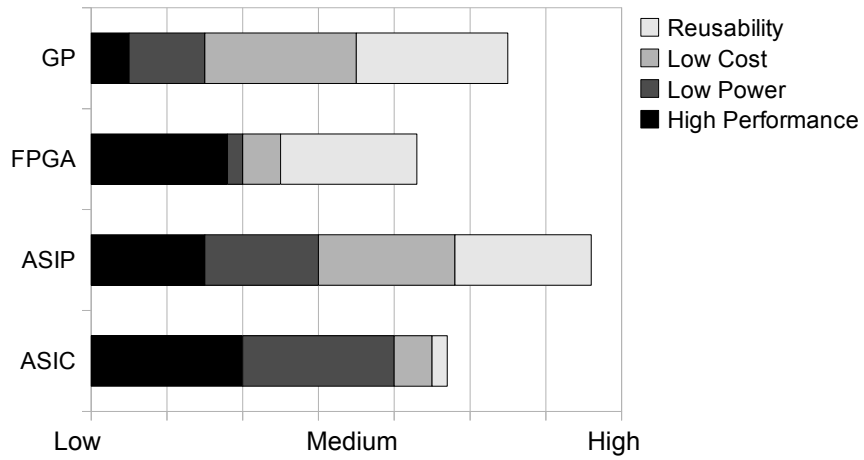


Figure 1.1: *A comparison of the merits of general purpose processors and different types of customisable embedded processors. Each type has some advantages, but ASIP is the only type that scores well in every criteria.*

creasingly popular compromise between ASICs and general purpose embedded processors are Application Specific Instruction-set Processors (ASIPs). These processors take a pre-verified baseline processor as a core and add extension instructions, thus standard tasks can use the baseline processor but critical kernels can be programmed to use the extension hardware. This strikes a balance between performance and time-to-market, and the pre-verified baseline avoids much of the risk involved in developing new hardware. See figure 1.1 for a visual comparison of the merits of ASICs, ASIPs and also general purpose processors and field programmable gate arrays (FPGAs).

Manually designed ASIPs can out-perform general purpose embedded processors, use less power and are quicker and cheaper to design than ASICs [Keutzer et al., 2002]. Using automated instruction set extension (AISE) to automatically design ASIPs, however, improves on this yet again. Manually designing an ASIP requires an engineer to spend a significant period of time analysing the target application, designing instructions and then implementing them in hardware. A set of automated tools, however, can do this by profiling the application to find "hot-spots" and analysing the data-flow at these spots to produce instruction definitions. These definitions can then be used to automatically create the hardware based on the data-flow graphs. This task would likely take an experienced engineer weeks or months to complete, but the automated tools can do the same thing in minutes or hours [Biswas et al., 2006a]. This approach reduces the cost of designing an ASIP and significantly reduces the time-to-market. There are many processors now on the market that allow the addition of customised extension instructions, allowing them to be used as the core of an ASIP. Examples of these baseline processors are the ARC 600 and 700 series, the Tensilica Xtensa, the ARM OptimoDE and the

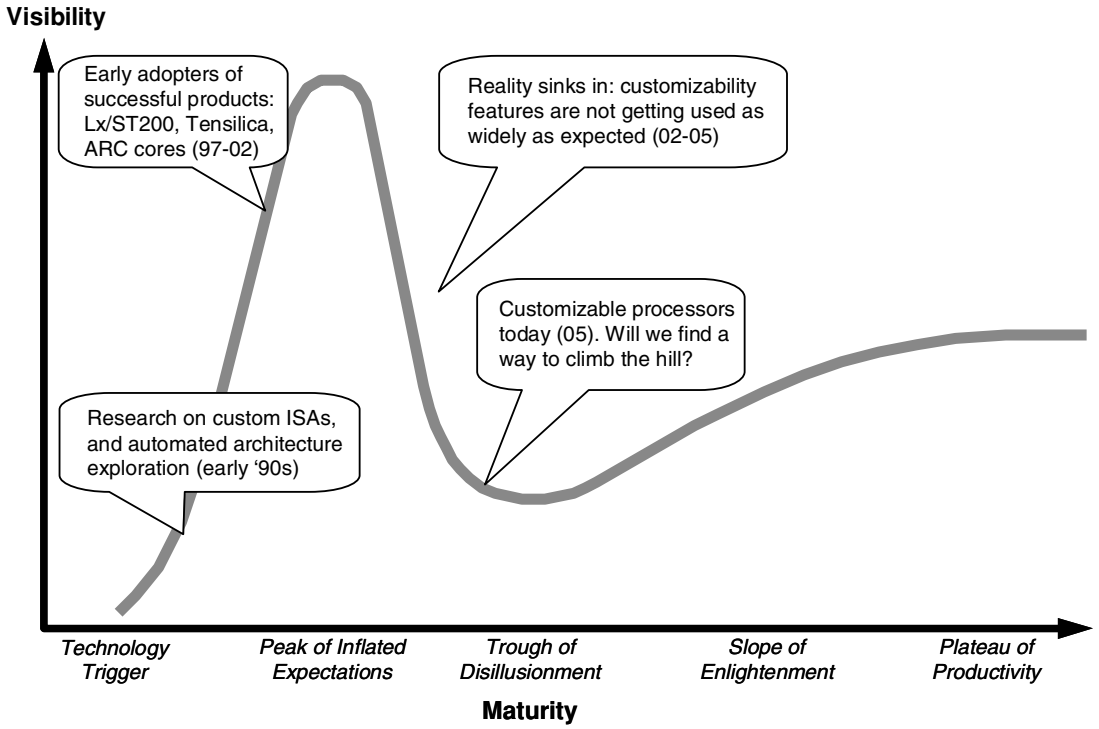


Figure 1.2: *Fisher, Faraboschi and Young's view of Gartner's "Hype Cycle" [Ienne and Leupers, 2007, Figure 3.4].*

MIPS Pro series.

1.2 The Problem

Fisher, Faraboschi and Young take an extremely cautious view on the future of customisable processors [Ienne and Leupers, 2007, Chapter 3]. Their reasons are related to customisable processors in general rather than focusing on extension instructions, but much of what they say also relates to AISE extended processors. Specifically, they present a view of Gartner's "Hype Cycle" [Linden and Fenn] which is shown here in figure 1.2. They state that in the 1990's there was a lot of hype regarding the potential of customised hardware. By 2005, however, it had been realised how hard it would be to bring customisability to anything like its full potential and the hype fell into a "Trough of Disillusionment". It is now 2011 and it does not seem as though the field has left the "trough" yet. It is the premise of this thesis that effective compiler technology is key to progressing to the "Slope of Enlightenment".

Others have already shown that using AISE to design ASIPs is an effective way to design hardware [Galuzzi and Bertels, 2008]. The problem, however, is that as with almost every hardware advance in the history of computing, the capabilities of compilers lag behind the features of the hardware. This thesis, therefore, investigates how the compiler can effectively

use an AISE produced processor.

The current standard methodology for using extension instructions within programs is for the AISE tool to note where it finds each extension. Generally, the tool will then modify the input code to eliminate the covered operations and replace them with either inline assembly code or *compiler known functions*. This is acceptable in the situation where a single program is being accelerated, but it creates an issue if the program needs to be changed after the processor has been fabricated. Re-running the AISE tool may generate different instructions requiring an engineer to manually map the old extensions to the new code, which is both time-consuming and error-prone [Ienne and Leupers, 2007, Section 6.5]. It would be more appropriate for the compiler to automatically perform the mapping for the engineer.

This, however, turns out to be a difficult task. The instruction set extensions (ISEs) produced by AISE are often far too complicated for conventional tree-based instruction selection (indeed, the instructions are often directed acyclic graphs, not trees), and they are often far too large for peephole-based instruction selection. Some form of graph-based instruction mapping is required instead; this has been investigated previously, but for much smaller instructions than those produced by AISE. Systems based on Architecture Description Languages (ADLs), for example, generally just produce compilers with standard tree-based back-ends. These are sometimes able to combine trees during matching to exploit small (e.g. two-node) instructions, but are unable to exploit larger instructions.

Not all extension instructions are automatically generated; many instruction set architectures (ISAs) have had additional instructions added that interact with the core processor via some extension interface. For example, the x86 ISA has the MMX and SSE extensions. These are used in the standard instruction stream, but interact with the baseline x86 instructions through additional vector registers. These instructions are designed to be used with highly regular, data-parallel code, such as that commonly found in multimedia applications. Successfully enabling a compiler to use them, however, is difficult. Techniques exist that "vectorise" loops to use these instructions [Allen and Kennedy, 2001], but they frequently fail to produce efficient code. In fact, code that can heavily exploit MMX and SSE instructions is one of the few areas in the x86 world where hand-written assembly is still considered worthwhile. For example, in the FFmpeg [Various, 2011a] and x264 [Various, 2011b] video codecs every critical kernel is hand-written in MMX or SSE assembly. MMX and SSE instructions are graph-shaped but repeat a very small, regular pattern two or four times, e.g. grouping four multiplies together. Extension instructions produced by AISE are also graph-shaped, but are not guaranteed to be regular. Thus, these instructions are a superset of the regular extension instructions and compiling for them will be at least as difficult, if not more so. This thesis, therefore, investigates how effectively the compiler can use these AISE produced extension instructions.

1.3 Contributions

The contributions of this thesis are many, as the research was undertaken using an iterative approach where each step directed the next, leading to some interesting results.

The first major contribution is a method for mapping arbitrary graph-shaped instructions (both disjoint and not) to arbitrary programs. The novel aspects of the mapper are that it performs instruction mapping in the middle-end instead of the back-end. This allows it to focus only on extension instructions, which the back-end cannot exploit. The mature and tuned back-end performs effective instruction selection for the code which is not mapped to extension instructions. Additionally, the instruction mapper is able to target a new extension interface based on vector instructions complete with vector permutation units [Almer et al., 2009].

An extensive evaluation is performed using 179 benchmarks from seven benchmark suites obtained from five sources. Results are generated using a hardware-verified cycle-accurate simulator. Additionally, the instruction mapper is evaluated using extension instructions generated for programs which are similar (but not identical to) the programs which they are mapped to. Changes include dropping profiling data, modifying programs or using completely different implementations of an algorithm.

The instruction mapper was able to find 84.5% as many extension instruction mappings as the AISE tools predicted. The more interesting results at this stage, however, are the negative results – as they lead to interesting conclusions about the role the compiler should play in an AISE based framework. The instruction mapper only uses 60% of the extension instructions that the AISE tool generates and only achieves an average speed-up of 1.11x. Additionally, 110 of 179 benchmarks actually run more slowly when using extension instructions. This is found to be due to the costs of mismatch between regular vectors and irregular extension instructions.

These results lead to modifications that arguable represent the most important contributions of this thesis. Several changes are made, but they are evaluated independently from each other. The first change allows the compiler to ignore extension instructions where the irregular-regular mismatch cost is likely to outweigh the instructions benefits. This increases the average speed-up to 1.20x and only 66 of 179 benchmarks slow-down. Hard-wiring some constants into extension instructions also increases the average speed-up to 1.20x. Using smaller extension instructions can increase the average speed-up up to 1.13x. Adding an additional parameter to the AISE model to represent the irregular-regular mismatch cost can increase the average speed-up to 1.24x with only 52 benchmarks slowing-down. The most effective change is to abandon the vector registers completely and thus discard all issues regarding irregular-regular data mismatches. Evaluated using the same terms as the rest of this paragraph this results in an average speed-up of 1.28x and zero benchmarks slowing down (though six do not experience a speed-up). This would have some unrealistic hardware requirements though, but reducing the hardware demands only lowers the speed-up to 1.26x (while still resulting in zero slow-downs).

Vector loads and stores are also found to be likely to provide a small benefit (increases average speed-up to 1.14), but to increase bandwidth in a system without vector registers dual memory banks are evaluated. A source-level memory partitioning scheme based on genetic programming was able to produce better results in far less time than an "optimal" integer linear programming based approach when operating at the source-level. The source-level approach itself is also a novel contribution which required the development of a C-to-DSP-C source-to-source compiler.

Finally, the compiler is used to modify the shape of programs before the AISE tool processes them. 59 transformations are used to create a large transformation space. An extensive random sample of this space is taken and is used to produce a reduced space which covers short sequences of the most important transformations. This reduced space is then exhaustively enumerated. This approach found that the combined speed-up of transformations followed by AISE could be greater than the product of the two separately. Additionally key transformations for AISE are identified.

1.4 Structure

The rest of the thesis is structured as follows.

Chapter 2 provides background information to aid in the understanding of the thesis. It primarily presents tools and techniques that later chapters use but do not improve upon.

Chapter 3 examines prior work attempting to solve the problems directly related to the issues explored by this thesis. The specific areas of previous work considered are: classical instruction selection, complex instruction mapping, compilation for dual memory banks and compiler transformations, as they apply to instruction set extension.

Chapters 4–7 present and evaluate the techniques developed for this thesis. **Chapter 4** specifically examines how to map extension instructions onto programs by use of a graph-subgraph isomorphism checker and how to perform register allocation for the vector registers used by the extension interface. These techniques are evaluated, both directly and by re-using extension instructions in programs similar, but not identical, to the programs for which they were generated. While this shows that the tool is effective at exploiting extension instructions, the results are sometimes marginal or introduce a reduction in performance.

Chapter 5, therefore, investigates various ways of increasing the effectiveness of extension instructions. Firstly, by changing the heuristic parameters for extensions generation. Secondly, two independent hardware modifications to ease the use of extension instructions are proposed; one using vector loads and stores with the vector registers and the other eliminating vector registers entirely.

Chapter 6 takes the two hardware modifications proposed in chapter 5 and unifies the

idea of increasing memory bandwidth (through vector loads and stores) with the clear benefits of eliminating the use of vector registers. This is achieved by proposing another hardware extension: dual on-chip scratchpad memories. Multiple techniques to split data between the two memory banks are proposed and evaluated.

The previous chapters presented techniques to enable the compiler to use extension instructions. **Chapter 7** uses the compiler to enhance extension generation. Source-to-source transformations are used to modified programs before they are presented to the AISE tool and certain key transformations were found to improve the quality of the extension instructions produced.

Chapter 8 concludes the thesis by summarising the results and contributions presented in earlier chapters and by providing a high-level critical evaluation that considers broad issues that were not specifically related to any single chapter.

Finally, **appendix A** contains a list of the source-to-source transformations used in chapter 7 and **appendix B** has the complete versions of the charts shown in previous chapters.

1.5 Summary

This chapter has introduced the thesis by motivating the use of ASIPs in embedded computer systems but also outlining the compiler issues that they introduce. The contributions of this thesis were summarised and the structure of the thesis was described by means of a brief walk-through which showed how each chapter is motivated by the results of previous chapters.

Chapter 2

Background and Infrastructure

"It would appear that we have reached the limits of what it is possible to achieve with computer technology, although one should be careful with such statements, as they tend to sound pretty silly in five years."
— John Von Neumann, *mathematician*, 1903–1957. Quote from circa 1949.

This chapter provides a short overview of the technologies and techniques relevant to the work of this thesis and is organised as follows: section 2.1 introduces embedded processors and highlights the various design options offered by these devices; section 2.2 covers the toolchain used in this thesis; section 2.3 outlines an automated instruction set exploration (AISE) algorithm; section 2.4 explains the concept of Design Space Exploration and section 2.5 describes Dual Memory Banks. Section 2.6 summarises Genetic programming relating to the work of this thesis, section 2.7 introduces Graph Theory, and finally and section 2.8 sets out the benchmark suites used for evaluation.

2.1 Embedded Processors

A complete description of embedded processors would be too large for this thesis. Therefore, a short summary is provided here, more complete descriptions have been written by Fisher et al. [2005, Chapter 1] or Ienne and Leupers [2007].

As described in chapter 1, embedded processors are required to be low cost, low power and small:

- **Low cost:** embedded processors must be cheap to produce. Despite the importance of the processor they usually only account for a small portion of the overall design budget.
- **Low power:** hand-held devices are battery operated and even in static devices low heat output is essential.
- **Small:** hand-held devices have limited space for electronics, the smaller the size of the packaged processor the better. This is helped by increasing transistor density, which

allows multiple processors to be included on a single die, i.e. System-on-Chip.

2.1.1 Embedded Processor Families

There are several different families of embedded processors:

- **General Purpose Embedded Processors** are designed to be usable in a wide-range of scenarios. Many started as scaled down versions of general purpose workstation processors. They can be used as stand-alone processors, but are often licensed as an IP-block for use within a system- on-chip.
- **Microcontrollers** are contained in most industrial electronics, from washing machines to cars. They operate as stand-alone processors controlling mechanical processes. Their design is often descended from older 8-bit and 16-bit architectures.
- **VLIW and DSP** are specialised for digital signal processing and multimedia functionality, they are effective for highly data-parallel processing. Their specialised design means they sometimes have to be paired with a general purpose processor.
- **Reconfigurable Processors** are processors that allow partial run-time reconfiguration. This is achieved with a microcode programmable micro-architecture, an FPGA is the full realisation of this. This fabric is not power efficient so reconfigurable processors rely on coarse-grain specialisation to exploit the reconfigurable fabric and save power.
- **Application Specific Integrated Circuits (ASICs)** are fully custom-designed pieces of hardware that can perform one task extremely efficiently. They have a very high design cost which unlike general purpose processors cannot be amortised across applications.
- **Application Specific Instruction-set Processors (ASIPs)** are normally general purpose embedded processors with additional application specific instructions, though any kind of processor can be used as a baseline. The use of a standard baseline means that their design-cost is much lower than that of an ASIC.

2.1.2 Application Specific Instruction-set Processors

Designers of embedded processors are increasingly using techniques that were previously only used in general purpose processors to trade-off processor size for performance. As processor fabrication capabilities improve embedded processors designers are able to use the extra die-space to design more complicated processors with deep pipelining and superscalar processing. For example, the ARM Cortex-A8 processor has a 13-stage pipeline and is dual-issue superscalar [ARM Ltd., 2010a]. Additionally, the ARM Cortex-A9 uses out-of-order execution [ARM Ltd., 2010b].

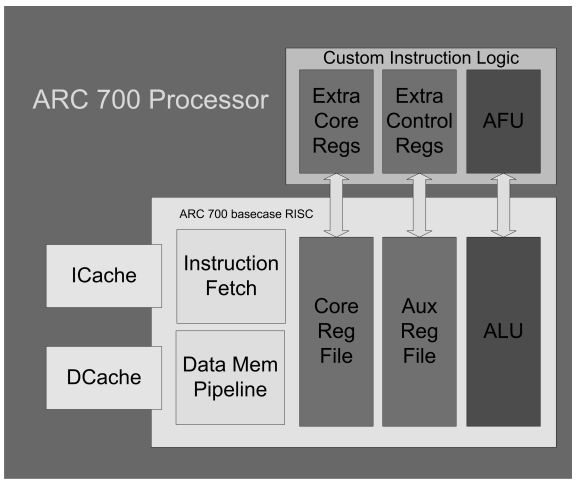


Figure 2.1: *A simplified system-level view of ARC700 family architecture, demonstrating the pre-verified baseline core and its connection to an instruction set extension through custom registers and arithmetic units.*

These general-purpose modifications will improve performance for almost every kind of program, either by allowing higher frequencies (deep pipelining), or by increasing the number of operations completed per-cycle (superscalar and out-of-order execution). Their generality, however, means they are power-hungry. In the situation were a specific set of applications are being targeted it is possible to use the information this provides to produce a specialised processor that is more power-efficient.

Historically the standard way of exploiting application specialism was to build an ASIC. As the tasks required of embedded computers have been increasing in complexity however, the cost and time required to design an ASIC has grown [Keutzer et al., 2002]. For most situations it is now preferable to build an application specific instruction-set processor (ASIP). These are cheaper and quicker to design because they are centred around a general-purpose baseline core which can be extended in an application-specific manner.

Extensible processors are based on the premise that processor speed, die area, and power consumption can be improved if the architecture of the processor is extended to include some features that are application-specific. This approach requires an ability to extend the architecture and its implementation, as well as the compiler and associated binary utilities, to support the application-specific extensions. It may seem counter-intuitive that adding to a processor will save die area, but application-specific extensions are more efficient than general purpose enhancements.

Processors may be extended *statically* or *dynamically*. A processor can be extended statically by augmenting the Verilog description of the processor prior to synthesis. Once the extensions have been incorporated, and the design has been fabricated, the extensions cannot be modified or further extended. A dynamically-extensible processor must be implemented,

either wholly or partly, in some form of field programmable logic fabric [Stretch Inc., 2007].

The simplest forms of extension, which are perhaps more properly considered to be forms of configuration, are the micro-architectural modifications that can be made to caches and their associated bus structures. Most embedded microprocessor cores provide the capability to adjust level-1 cache size, associativity and sometimes also block size and memory bus width. These all have a significant impact on performance, die area and power consumption. They are also relatively easy to exploit, as they require no changes to the instruction set architecture.

True architecture extensions begin with the capability to add custom instructions to a baseline instruction set. In their simplest form these may be predefined *packs* of add-on instructions, such as the ARM DSP-enhanced extensions included in the ARM9E [Francis, 2001], the various flavours of MIPS Application Specific Extensions [MIPS Technologies, 2007], or ARC's floating-point extensions to the ARCompact instruction set [ARC International, 2007].

These are domain-specific extensions, they can be used across many related tasks. Application-specific instruction set extensions are not predefined by the processor vendor but are instead identified by the system integrator through analysis of the application. To allow such instructions to be incorporated into a pre-existing processor pipeline, there must be a well-defined extension interface. From a high-level architecture perspective this interface will allow the extension to operate as a "black-box" functional unit at the execute (EX) stage of a standard RISC pipeline. This is an over simplification though, standard RISC instructions are two-input and one-output. Effective extension instructions require this constraint to be relaxed as extensions exploit the parallelism available in large instructions. This, therefore, generally requires an extended or additional register file, hence the need for an extension interface.

Practical extensible processors for the embedded computing market, such as those from ARC and Tensilica, normally have single-issue in-order pipelines of 5–7 stages. This permits operating frequencies in the range 400MHz to 700MHz at the 90nm technology node. Extension instructions may be constrained to fit within a single clock cycle, or may be pipelined to operate across multiple cycles. Current dynamically-extensible cores that use FPGA fabrics to implement the extension instructions cannot operate at these speeds. For example, extension instructions in the Stretch S5000 [Stretch Inc., 2007] operate at one-third of the clock rate of the processor.

The representation of instruction set extensions varies from one vendor to another, but essentially describes the encoding and semantics of each extension instruction in ways that can be understood by both a processor generator tool and all of the software tools (e.g. compilers, assemblers and simulators). There follows a process of translating the abstract representation of the extension instructions to structural form using a Hardware Description Language (HDL) such as Verilog or VHDL. This is then incorporated into the overall HDL definition of the processor, which is then synthesised to the target silicon technology or perhaps to an FPGA.

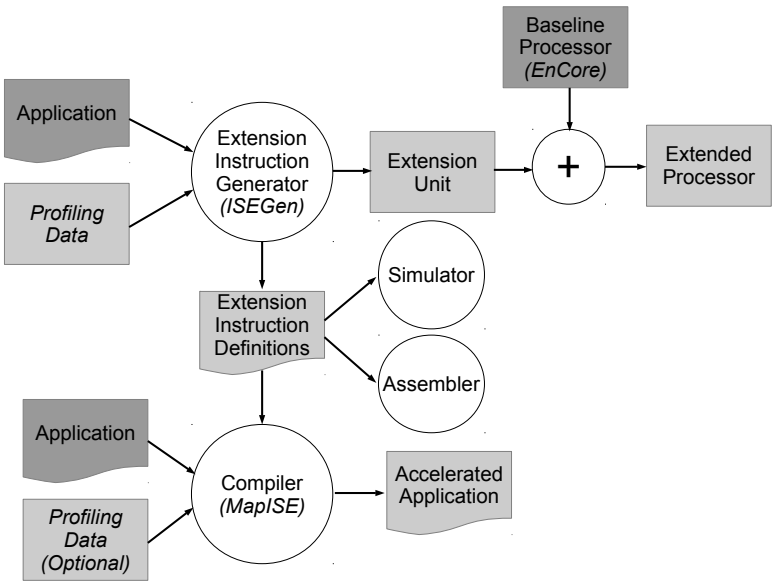


Figure 2.2: *How the different tools in PASTA interact.*

2.2 Infrastructure

The research presented in this thesis was undertaken within the PASTA project: Processor Automated Synthesis by iTerative Analysis. This was a large, multi-person project that aimed to automatically produce application specific processors and their associated tools. An application which is to be accelerated is provided and the PASTA toolchain produces a processor designed to accelerate that application as well as a compiler and a simulator that support the accelerated processor.

Figure 2.2 shows how the key components of the PASTA project interact. The target application and associated profiling data is passed to the extension instruction generator (*ISEGen*). This produces an XML specification of the generated extension instructions and a Verilog implementation of the extension unit. This Verilog extension unit can then be integrated with the *EnCore* baseline processor to produce an extended processor. The XML extension instruction specification is used to produce a new assembler and a simulator that supports the newly specified processor. Finally, this same XML specification is provided to the *MapISE* compiler along with an application (potentially the original target) and profiling data so as it can produce an accelerated application which will run on the extended processor.

Disclaimer. Of the components described in figure 2.2 only the compiler (*MapISE*) is presented as a contribution of this thesis. Specifically *ISEGen*, the simulator and *EnCore* were not developed by the author of this thesis but by other members of the PASTA project.

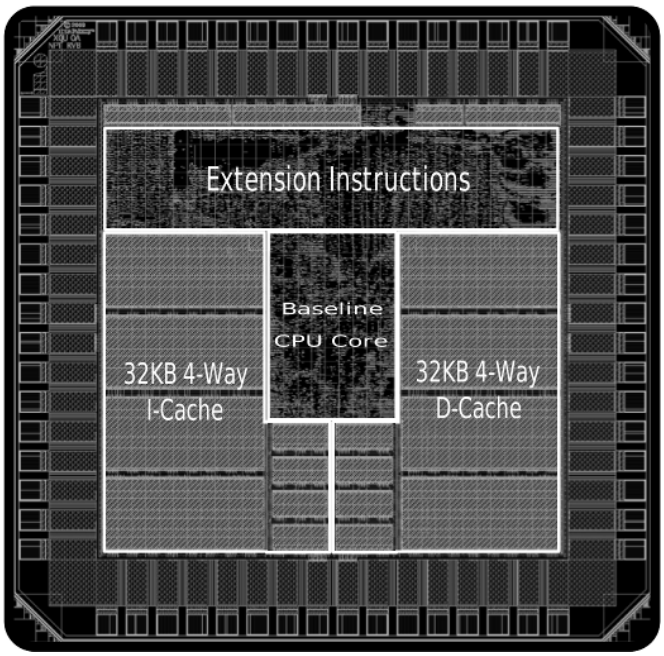


Figure 2.3: *A specific instantiation of an* EnCore *processor with an extension unit.*

2.2.1 *EnCore*

The extensible processor used in this thesis is the *EnCore*. This processor is an implementation of the ARCompact ISA. The ARCompact ISA is a 32-bit RISC ISA with 16/32-bit mixed mode instructions and optional extensions for digital signal processing, floating point operations and dual memory banks. The *EnCore* implementation is both low power and high performance: using a generic 130nm process it operates at a frequency of up to 375MHz, or up to 600MHz using a 90nm process. In terms of customisability: *EnCore* has configurable caches, it may be specified with 16, 32 or 64 general purpose registers and it has an extensive extension interface. Figure 2.3 shows the layout of a 90nm *EnCore* processor with a five stage pipeline, two 32Kb caches, 64 general purpose registers and additional extension instructions.

2.2.2 *EnCore* Extension Interface

EnCore's extension interface is an addition which is not present in the original ARCompact ISA. It supports up to 255 extension instructions which read and write data via vector registers (four elements each). Extension instructions are used via a `vextNNN` mnemonic, e.g. `vext001`, `vext002` etc. Each use of a `vext` instruction may use up to three input vectors and two output vectors, e.g.

```
vext001 vr04,vr05 = vr01,vr02,vr03
```

This means that an extension instruction may have up to twelve input values and eight output values. For instructions which do not require every vector the register `vr00` is used in

Vector	Scalar Registers			
vr01	r35	r34	r33	r32
vr02	r39	r38	r37	r36
vr03	r43	r42	r41	r40
vr04	r47	r46	r45	r44
vr05	r51	r50	r49	r48
vr06	r55	r54	r53	r52
vr07	r59	r58	r57	r56

Table 2.1: *The mapping of scalar registers to vector registers on* EnCore.

the mnemonic as this is a read-only register that is hard-wired to all zeroes, writing to it has no effect. If an instruction does not need to read or write an entire vector then elements may be ignored by unsetting a bit in the read or write mask that is hard-coded into the instruction's hardware implementation. This is especially important when writing to vectors as otherwise register contents will be unnecessarily "clobbered" (i.e. live-ranges are interrupted and the compiler will need to spill any live values stored in those registers). There is also a special `vmov` instruction (or `vext000`) which is the vector register equivalent of a RISC `mov` instruction.

The extension unit interface in *EnCore* is only connected to the vector registers, extension instructions are not capable of accessing individual scalar registers. *EnCore* has seven physical vector registers, each of which contains four 32-bit elements. Each of these 28 elements are mapped to the upper-half of the general purpose register file, as shown in table 2.1. In the ARCompact ISA registers `r00–r31` are already used for standard compilation and `r60–r63` are used for specific purposes. This leaves registers `r32–r59` for mapping to the vector register file. When these are arranged into four element vectors the limit of seven physical vector registers is created.

The baseline *EnCore* processor, however, can access any register `r00–r63` including the registers `r32–r59` which are mapped to vector registers. The baseline processor cannot access an entire vector at once, it can only access individual elements. Figure 2.4 shows the two separate register interfaces that are used to make this work. The baseline processor uses 32-bit ports to access single scalar registers and is connected to the entire register file with two read and two write ports. The extension interface uses 128-bit ports to access entire vectors at once and is connected to the extension register file with three read and two write ports. The reduced wiring complexity due to only having seven vector registers makes the third read port viable.

Figure 2.4 also shows that two of the vector register read ports have "permutation units". These allow the extension instructions to access the elements of vector registers in a different order than in which they are stored. Due to limited space in the instruction encoding only

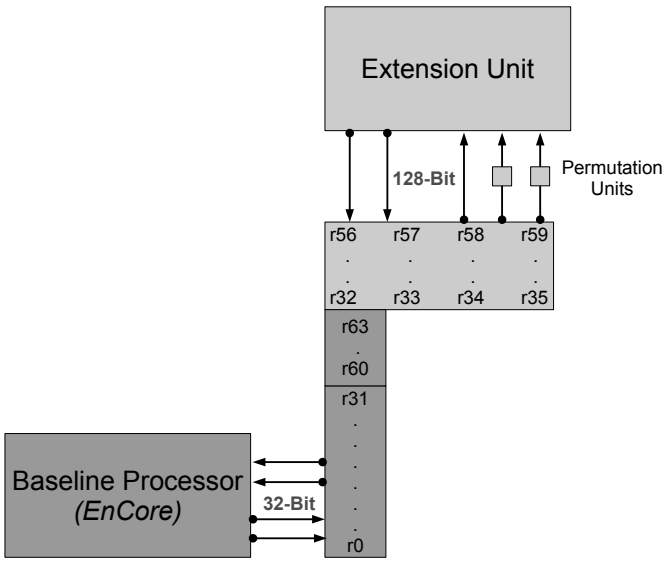


Figure 2.4: *Extension instructions interface with* EnCore *via vector registers.*

Vector	Permutation	Scalar Registers			
`vr01`	Identity	`r35`	`r34`	`r33`	`r32`
`vr08`	Swap Central 2	`r35`	`r33`	`r34`	`r32`
`vr09`	Swap 2 Ends	`r32`	`r34`	`r33`	`r35`
`vr10`	Swap Pairs	`r34`	`r35`	`r32`	`r33`
`vr11`	Swap Left Pair	`r34`	`r35`	`r33`	`r32`
`vr12`	Reverse	`r32`	`r33`	`r34`	`r35`
`vr13`	Rotate Left	`r34`	`r33`	`r32`	`r35`
`vr14`	Rotate Right	`r32`	`r35`	`r34`	`r33`
`vr15`	Rotate by 2	`r33`	`r32`	`r35`	`r34`

Table 2.2: *The permutations available for* `vr01` *on* EnCore. `vr16`–`vr63` *provide equivalent permutations for* `vr02`–`vr07`.

9 of the possible 24 permutations are available, these are shown in table 2.2. These nine permutations were chosen such that the other permutations are accessible by combining two permutations. This is done by using a `vmov` instruction to apply the first permutation and apply the second permutation via the `vext` instruction. The intention of the permutation units is to make it easier to use the output of one extension instruction as the input to another, especially in the context of loops.

2.2.3 *ISEGen* and *uArchGen*

The *ISEGen* tool generates a specification for a set of extension instructions to accelerate a provided target application by means of the ISEGEN algorithm [Biswas et al., 2006a]. The generation of the Verilog implementation of the extension unit is actually done by a second tightly coupled program called *uArchGen*.

uArchGen can optionally use resource sharing when creating the extension unit. This means that if two instruction implement the same operator the can share the same hardware (e.g. they both contain a multiply then the same hardware multiplier could be used for both). This can reduce performance, however, as the extension instructions are implemented over three of *EnCore's* five pipeline stages so as to avoid large instructions decreasing the maximum achievable clock frequency. This means that if two instructions want to access a shared resource at the same time a pipeline bubble will occur. *uArchGen* can also fit the extension unit into a target die area, but it does that by excluding some of the instructions that *ISEGen* finds. As this thesis is evaluating the compiler and not the hardware implementation, *uArchGen* is used in its default mode of not using resourcing sharing and allowing unlimited die size.

ISEGen, however, has many heuristic parameters. The core ISEGEN algorithm uses seven heuristic weights that affect the search process. These are all left at their default values for this thesis as small changes can drastically slow down the algorithm. Other parameters are safer to use, such as changing the maximum number of input and outputs that an instruction may have or changing the expected costs of implementing certain operations in hardware.

2.3 Automated Instruction Set Extension

An AISE algorithm used in this thesis is described below, but many others exist, Galuzzi and Bertels [2011] provide an overview of the topic with references to many other algorithms. There are actually two AISE algorithms implemented by the tools used in this thesis. The simpler algorithm is described below while also outlining AISE methodology. The more advanced algorithm is described by Biswas et al. [2006a].

2.3.1 Atasu AISE Algorithm

Atasu et al. [2005a] implemented an integer linear programming based automated instruction set extension tool to find optimal extensions. Each basic block was defined as an integer linear program, along with the input and output dependencies of the block. Additional constraints were then added to the model such as the maximum number of inputs or outputs an instruction may have and enforcing the convexity constraint. The convexity constraint states that if a custom instruction provides an input to some node, and uses the output of the same node then that node must be contained within the custom instruction or the instruction is invalid. Without

this the node may need to be scheduled both before and after the custom instruction, which is clearly impossible. Every node in the basic block implements some simple operation, in the model there are hardware and software costs for each operation. The hardware cost is the fraction of a cycle (maybe greater than 1 cycle) that it takes to perform the operation. The software cost is the number of cycles the standard instruction that performs this operation takes to complete. The integer linear program is then used to find the single instruction which results in the greatest reduction in schedule length from implementing that set of nodes in hardware. Once an instruction is found, all the nodes implemented by that instruction are collapsed into a single node and the algorithm is re-run to find another instruction (the collapsed node may not be included in another instruction). This means given the already picked instructions the next instruction picked will be genuinely optimal – however the overall set of instructions isn't necessarily optimal.

The ILP AISE algorithm generates data-flow-graph templates through the conversion of basic blocks to a set of constraints in an *Integer Linear Program (ILP)* and solution of that program. For the implementation used in chapter 7 a tool built into a CoSy compiler uses the *lp_solve* library [Berkelaar, 2008] to solve such problems and generate a set of candidate templates for an entire program. Constraints are declared for each basic block to generate a template such that:

1. The template is **convex** (i.e. there is no dataflow path between two operations in the template that includes an operation that isn't in the template), so that it may be scheduled.

2. Input and output **port constraints** are met (i.e. the number of register input and output ports are sufficient), so that it may be implemented.

In addition to the constraints, a goal function is also expressed. For this algorithm, the goal is the estimated serial time of execution in cycles of the instructions covered by the template, minus the estimated critical path of the template. The former is denoted the "software" execution time, and is indicative of the time the instruction would take to execute on the unextended architecture. The latter is denoted the "hardware" execution time, and is a real-valued factor of the cycle time taken to execute the template as a single instruction. A cycle time for each software and hardware operation is specified to the tool a-priori to ILP construction, to allow for the constraints to be generated. The per-template difference between software and hardware execution time is the per-execution gain in cycles to an architecture implementing that template. Following the generation of templates from basic blocks, the templates are checked for isomorphism with one another using the *NAUTY* [McKay, 2008] graph isomorphism library, then ranked using the product of their estimated usage and per-execution gain. The top four of these instructions are then recorded alongside their performance estimates for inclusion in results.

2.3.2 HW/SW Codesign

HW/SW Codesign was an active research area in the 1990's and has inspired subsequent work on *Electronic System Level Design*, e.g. [Keutzer et al., 2000; Balarin et al., 2003]. A comprehensive summary of research directions, approaches and tools has been written by Rozenblit and Buchenrieder [1995]. This work covers a broad scope of issues, typically ranging from the analysis of constraints and requirements down to system evaluation and design verification. In contrast, this thesis focuses on a more specific, individual problem, namely that of HW/SW partitioning in the context of extensible application-specific processors. The primary feature that distinguishes *HW/SW Codesign* from AISE HW/SW Partitioning is software synthesis. In *HW/SW Codesign* a problem definition is used to synthesise both hardware and software, in AISE HW/SW Partitioning portions of the software are implemented in hardware.

2.4 Design-Space Exploration

Design-space exploration is an optimisation process in the design flow of a System-on-Chip. Typically, the search has multiple constraints (performance, power, cost etc.), and targeting an often multi-dimensional and highly non-linear optimisation space. Multiple dependent levels (algorithm, SW, and HW design space exploration) of interaction make it difficult to employ isolated local search approaches, but require a combined effort crossing the traditional boundaries between design domains, providing feedback paths and integrating tools into larger frameworks.

Configurable and extensible processor cores such as the ARC 600 and 700 have a number of capabilities to allow their instruction set and micro-architecture to be optimised for a particular application. Design concerns guiding the exploration are often reduced to metrics such as execution speed, power usage, and die area; each of these metrics has an accompanying relevant design space in the configuration and extension domain.

Meeting the main execution speed requirement means that no further increase in speed is generally useful, other than to provide an overhead for development. Application deadlines will be met and the system built around the core will be able to communicate and process data without stalling due to system-level deadlines missed by the core.

Once execution speed requirements have been met the focus of designers may be switched to secondary axes of design concern, such as power usage. Efforts in addressing one axis of design concern may make use of excesses in other axes. For example if performance exceeds requirements the clock speed of the ASIP may be reduced, reducing the power consumption. These secondary concerns have additional design spaces of the configurable core available to be explored for satisfactory areas; for example clock gating, dynamic voltage scaling, and unit pruning. These are, however, outside the scope of instruction set extension and are not covered

here.

Unfortunately the "second order" effects of core extension are not always beneficial and are often hard to predict with any accuracy. Adding more logic to a core can increase the critical path and force a reduction in the overall clock speed. A complicated web of non-orthogonal trade-offs forms a space which can only be explored efficiently with iterative, automated help.

Instruction set extensions affect all three of the aforementioned axes of design concern. The guiding metric in deriving extensions is often still application execution speed; designers will add instructions that "cover" the hottest (most frequently executed) sections of their application code. The intention is that by partitioning the application code into areas covered by extension instructions, subsections of micro-architecture can be dedicated to the servicing of these new instructions. In this highly application-specific design space, several sources of micro-architectural optimisation are currently brought to bear on the hardware performance of the new instruction:

1. **Operation-level/Spatial parallelism**; Parallel instances of arithmetic hardware in order to perform multiple operations at-once, as allowed by dependencies.

2. **Reduced register-transfer overhead**, due to the increased locality of communication within the functional unit used to represent the new instruction.

3. **Aggregation of clock period surplus present in most arithmetic functions**. In particular, bit-wise functions have a hardware latency far below the clock period in most cases.

This growing catalogue of optimisation aims to ensure that the "hot-spot" represented by the new instruction achieves the maximum speed possible, by trading off die area for an increase in execution speed, a decrease in power usage, and a decrease in code-size. Often these extensions correlate to very frequently executed sections of code, and so the benefits for a relatively small increase in die-size can be very tempting to designers. The problem remains to find a way to accurately model both the existing architecture and the full range of potential extensions in such a way as to efficiently automate exploration.

It has been shown [Bonzini and Pozzi, 2006] that new search methods and heuristics can be developed to control the application of transformations, with respect to the new set of goals inherent in AISE as compared to code generation. Transformations once targeted at the back-end would attempt to limit increasing basic block size due to register pressure, e.g. [Gupta and Bodik, 2004]. Now in instruction set extension the drive is towards the largest possible basic block size for analysis.

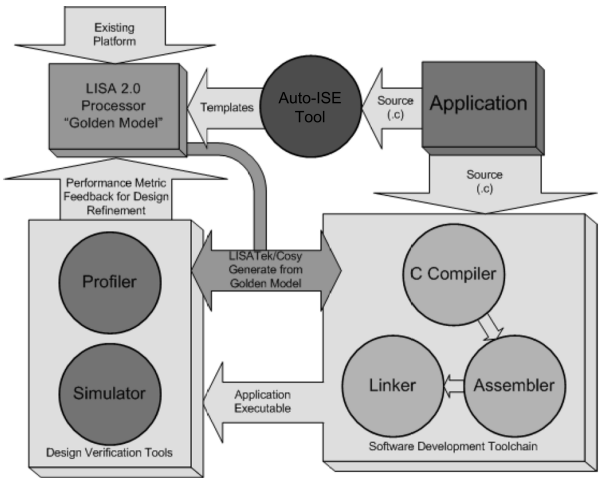


Figure 2.5: *The Compiler-in-loop methodology for ASIP design space exploration.*

2.4.1 Automated Instruction Set Extension

Automated instruction set exploration (AISE) has been actively studied in recent years, leading to a number of algorithms [Peymandoust et al., 2003; Biswas et al., 2006a; Atasu et al., 2005b; Pozzi et al., 2006] which derive the partitioning of hardware and software under micro-architectural constraints. Work is still underway in defining the full space of exploration even in purely arithmetic instruction set design [Verma and Ienne, 2006]. Work to include better models in tools has allowed for better decisions about the performance and feasibility of extensions [Pozzi and Ienne, 2005].

The current exploration approach of using a range of tools operating on a canonical system-level ADL, is described as “Compiler-in-Loop Design-Space Exploration” [Hohenauer et al., 2004]. It was originally motivated [Glökler et al., 2003] through the discovery that iterative and methodical exploration of ASIP design is very beneficial in decreasing time-to-market. The CoSy [ACE Associated Compiler Experts, 2011] and Processor Designer [CoWare, 2007] tools feature in many such frameworks; figure 2.5 illustrates such a combination. The compiler, binary utilities such as assembler and linker, profiler, and simulator are generated from a single “Golden Model” written in LISA 2.0. This toolchain may then be used to evaluate the performance of the architecture described by the model. The application for which the model is intended is compiled, assembled, linked, simulated and profiled for run-time and other statistics. The Automated Instruction Set Extension tool is run over the application to determine a set of effective instructions to add to the model. The designer selects a subset of these instructions to add, and is able to use the generation facilities of the compiler-in-loop framework to generate a new toolchain for further performance analysis of the program. The designer is then free to modify the constraints imposed on the AISE tool (such as register file input / output) and repeat the loop to further explore the design space of the application specific processor

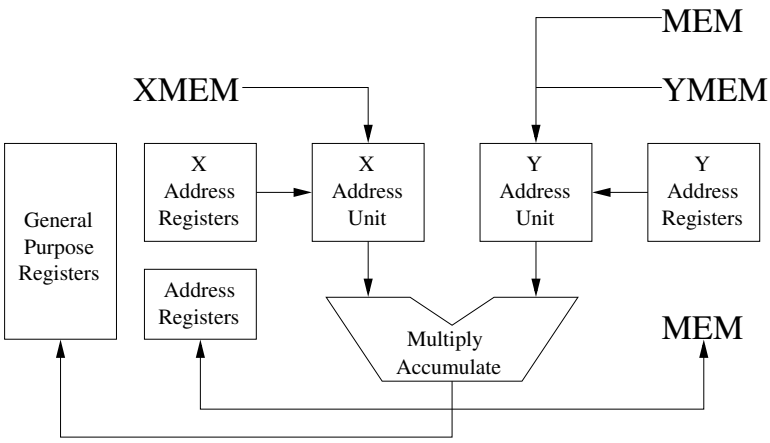


Figure 2.6: *Example DSP processor architecture with dual-input memory data path and MAC unit.*

described by the Golden Model. In this way, the designer is freed from many drawbacks inherent in design space exploration without such a framework. They may directly test the impact of a design decision without the need to manually re-implement a simulation and compilation toolchain. This top-down automated approach to design exploration and evaluation has been proven successful in case studies such as those cited in Hohenauer et al. [2004].

2.5 Dual Memory Banks

Digital signal processors are often required to process "infinite streams", e.g. decoding an incoming television signal or encoding a mobile telephone call. For these tasks the input data will keep on arriving for an unknown period of time, so the processor must be able to process the stream constantly in real-time.

A key performance bottleneck when doing this is memory bandwidth. To address this most digital signal processors have dual memory banks. For example, the TigerSHARC DSP platform that is used in chapter 6 is a hybrid VLIW/superscalar processor — this means it can issue multiple instructions per cycle. It also has dual memory banks (called "X/Y Memory"), this means it can issue two loads or two stores each cycle as long as the accesses are not to the same memory bank (i.e. it can access bank X once a cycle and bank Y once a cycle). This requires the compiler to split the program's data between the two banks in a way that maximises the possibilities for parallel accesses.

Figure 2.6 shows a generic DSP architecture with dual memory banks *X* and *Y*. These two banks are accessed via the *X* and *Y* addressing units, which may support DSP specific post-increment addressing modes. TigerSHARC's memory architecture fits this generic design.

2.5.1 DSP-C and Embedded C

Programmers can use dual memory banks directly with *DSP-C* [ACE Associated Compiler Experts, 1998], or its later extension *Embedded C* [JTC1 et al., 2004; Beemster et al., 2005]. These are sets of language extensions to the ISO C programming language that allow application programmers to describe the key features of DSPs that enable efficient source code compilation. As such, DSP-C includes C-level support for fixed point data types, circular arrays and pointers, and, in particular, divided or multiple memory spaces.

DSP-C uses address qualifiers to identify specific memory spaces in variable declarations. For example, a variable declaration like

```
int X a[32];
```

defines an integer array of size 32, which is located in the *X* memory. In a similar way, the address qualifier concept applies to pointers, but now up to two address qualifiers can be provided to specify where the pointer and the data it points to is stored. For example, the following pointer declaration

```
int X * Y p;
```

describes a pointer `p` that is stored in *Y* memory and points to integer data that is located in *X* memory. For unqualified variables a default rule will be applied (e.g. to place this data in *X* memory).

2.6 Genetic Programming in Compilers

There is a growing research field which makes use of machine learning in compilers. For example: early work was undertaken by Agakov et al. [2006], Wang and O'Boyle [2010] use machine learning for automatic parallelisation and Fursin et al. [2008] produced a complete machine learning compiler. For the purposes of this thesis, only one technique will be considered: genetic programming (GP) which is used in chapter 6.

Genetic programming is based on the same evolutionary principles as genetic algorithms. Instead of mutating and breeding strings, however, trees are used to represent functions [Cramer, 1985; Koza, 1992]. Trees are mutated and are allowed to survive by their "fitness", the higher their "fitness" the higher the probability that they will make it into the next generation. To breed two trees, one of each of their sub-trees are swapped and to mutate a tree, a sub-tree is replaced with a randomly generated sub-tree.

The algorithm is initialised with a population of entirely random trees. Each member of the population is then assigned a fitness by a provided fitness function. The next generation will be the same size as the current generation, $X\%$ of its members will be produced by breeding and $100 - X\%$ will be produced by mutation – 90 is a typical figure for X. Members are selected for breeding or mutation based on their fitness, there are many ways of doing this but

the experiments in this chapter use tournament selection. A set of size N is picked entirely at random with no bias towards fitter candidates, then the fittest of the N is selected. This means that the $N-1$ least fit candidates will never be chosen and the larger N is the stronger the bias towards fitter candidates becomes. Once the new generation has been formed the old one is discarded and the process starts-over with the new generation until some generational limit is reached, then the fittest candidate in the current generation is returned.

Stephenson et al. [2003b] used genetic programming in a compiler. They use genetic programming to generate heuristics for priority functions related to hyperblock formation, register allocation and data prefetching. Although many of the results they report are due to evaluating the heuristics on their own training data they also present results for separate test benchmarks. They were able to improve on the existing heuristic in a mature compiler by 9% on average for hyperblock formation across many SPEC benchmarks, demonstrating that genetic programming has potential within compilers.

Leather et al. [2009] take a different approach and use genetic programming to search a feature space as defined by a grammar. The work of Stephenson et al. [2003b] (and the work in chapter 6) both use GP to generate compiler heuristics, Leather et al. [2009] generate "features" which are then used by heuristics. This technique outperformed both the default *GCC* heuristic and a support vector machine (SVM) approach.

2.7 Graph Theory

Compiler writers have been using graphs extensively for a many years [Joshi et al., 2002]. They can be used to represent information, e.g. to represent control-flow. They can also be used to solve problems, e.g. finite automata to represent regular expressions during lexical analysis, or colouring graphs to perform register-allocation. Some of the techniques developed in this thesis are based on graphs: instruction mapping in chapters 4 and 5 and memory bank allocation in chapter 6. This section therefore summarises graph theory knowledge, leaving a more complete introductions to others [West, 2000; Diestel, 2010].

2.7.1 The Basics

A graph $G = (V,E)$ consists of a set of vertices V and a set of edges E. Each edge $e \in E$ joins two vertices $v_1, v_2 \in V$. This type of graph can be considered *undirected*, edges represent some form of association, but there is no direction of flow between vertices. In a *directed graph* each edge has a flow from $v_1 \rightarrow v_2$. If there are no loops ($\forall v_x \in V,\ \nexists\ a\ path\ v_x \rightarrow v_x$) in a *directed graph* then it is said to be a *directed acyclic graph*, or a *DAG*.

2.7.2 Specific Problems

Chapter 4 uses a graph-subgraph isomorphism checker. Graph-subgraph isomorphism is where the subgraph can be mapped onto some part of the larger graph. So each node from the sub-graph is mapped onto some node in the graph such that there are identical edges between the nodes on the graph as there are on the sub-graph. Every node and edge in the subgraph must be mapped to the graph, but nodes and edges in the graph do not need to be mapped to the subgraph. This thesis uses the VF2 algorithm for performing graph-subgraph isomorphism checking [Cordella et al., 2004].

Chapter 6 represents a problem as a soft-colouring problem. Hard-colouring is common in computing. In hard-colouring no two connected nodes may be the same colour, in soft-colouring it is merely desirable that this is so. A colouring should be found where the minimal number of neighbours are the same colour. This thesis uses an algorithm by Fitzpatrick and Meertens [2001] for soft colouring.

2.8 Benchmarks

This thesis has been evaluated using four benchmark suites that provide a representative overview of the domain of embedded applications. The EEMBC suite contains benchmarks for common tasks in the automotive, consumer (multimedia devices), networking, office machinery (e.g. fax-machines and printers) and telecommunication areas. The UTDSP [Lee, 1998] suite adds further digital signal processing applications and kernels. Each program is supplied in four versions, a version using pointer arithmetic, a version using arrays and software pipelined versions of each of these. There is a small amount of overlap with the EEMBC suite, specifically several of the kernels perform the same type of calculations as small parts of the applications that make up the telecommunication and consumer areas of the EEMBC suite. This overlap is actually an aid to evaluation and is directly used in chapters 4 and 5. The third suite used is another digital signal processing suite: DSPstone [Zivojnović et al., 1994], a collection of very small kernels. Finally, the SNURT [Seoul National University - Real-Time Research Group, 2008] suite is used, this is a set of embedded benchmarks representing common tasks in the real-time domain. Some of these are more digital signal processing kernels, but other tasks such as sorting and check-sum calculation are included. Additionally some more benchmarks are taken from an in-house collection of cryptographic implementations as these are commonly used for evaluating extension instruction generation techniques due to their high levels of data parallelism.

All the benchmarks used for evaluation are implemented in C, as this is the de facto standard in the embedded world, and none of them have been tuned for extension instruction generation or exploitation.

Chapter 3

Related Work

"O wad some Power the giftie gie us
To see oursels as ithers see us!
It wad frae monie a blunder free us,
An' foolish notion:
What airs in dress an' gait wad lea'e us,
An' ev'n devotion!"
— Robert Burns, *Scottish national poet*, 1759–1796. Final verse from "To a Louse" (approx. 1785).

This chapter examines prior work that is directly related to this thesis. Section 3.1 is directly related to chapter 4 and 5 and covers work that looks beyond the existing classical techniques to explore and exploit complex instructions. Section 3.2 is related to chapter 6 and considers existing solutions to the dual memory bank assignment problem. Finally, section 3.3 describes the early exploratory work that has been undertaken on the effect of compiler transformations on processor customisation.

3.1 Complex Instruction Mapping

An early piece of work in the area of complex instruction selection was undertaken by Leupers and Marwedel [1996]. The complex instructions targeted by this technique are instructions that may be represented as disjoint data-flow trees, i.e. their operations occur in parallel. For example: instead of targeting deep multiply accumulate instructions, they target wide multiply accumulates where the accumulation occurs via an architecturally visible temporary register and the result of the previous multiply is accumulated. The problem is encoded as the set of register transfer paths possible on the processor, and the set of transfers that each instruction implements. Standard optimal tree covering is performed on these register transfers and then an integer linear program is used to find instructions which may cover multiple register transfers. The type of hardware that this technique targets is less common now and small parallel instructions have mostly been replaced with short tree equivalents, reducing the usefulness of the technique since it was developed. The work was later revisited though [Leupers and

Bashford, 2000] to target a form of instruction that still consists entirely of parallel operations: SIMD, specifically sub-word SIMD. Sub-word SIMD tries to pack small operations into standard arithmetic instructions, e.g. packing four 8-bit additions into a 32-bit addition. As this vastly increases the space of possible instruction selections the standard optimal tree covering technique is extended so that instead of finding a single optimal covering it finds all optimal coverings. An integer linear program is then used to find a set of SIMD instructions which is capable of working within the register constraints that sub-word SIMD introduces. This is an interesting extension but is not equivalent to the instruction mapping requirements of this thesis: sub-word SIMD instructions are far simpler than AISE generated extension instructions.

Arnold and Corporaal [2001] extended the standard optimal dynamic tree programming algorithm to be able to handle instructions with multiple outputs. In principle, the cost function of the standard dynamic algorithm is modified so that when an output of the multiple-output instruction is used as an input to a node the cost is divided by the number of outputs. This is one of the few papers that specifically address mapping instructions generated by AISE, but as the paper concentrates on AISE itself the proposed mapping algorithm is not evaluated. This is an unfortunate short-coming as the changes to the dynamic tree covering algorithm breaks its ability to find an optimal covering. Without analysis it is difficult to establish the quality of the discovered mappings as optimally combining trees is a hard problem (Leupers and Bashford [2000] use an integer linear program to solve this part of the problem).

Scharwaechter et al. [2007] continues the development of complex instruction mappers by extending the idea of a code-generator generator to be able to handle parallel instructions. The described generator, Cburg, in principle operates like previous generators such as Iburg [Fraser et al., 1992] or Olive [Tjiang, 1993]. It extends the input grammar and replaces the matching algorithm, therefore both the code and the instructions can be represented as directed acyclic graphs (DAGs). Though unlike Iburg or Olive, Cburg is not guaranteed to find optimal results as it uses several heuristics to help reduce a worst-case exponential run-time down to an average case linear run-time. Evaluation was undertaken by integrating a Cburg produced back-end into the LCC compiler [Fraser and Hanson, 1991] for the MIPS ISA. As LCC uses Iburg produced back-ends this allows for simple integration of their Cburg produced back-end. LCC, however, has the disadvantage of not performing many middle-end optimisations. More complete compilers expend significant effort eliminating redundant or dead operations. As redundant code can generally be executed in parallel, and thus be ideal for execution by complex instructions, this paper could potentially be overstating its results. Local common sub-expression elimination and constant expression elimination are performed though, and these alone should eliminate some of the redundancy. The evaluation of the technique only describes the speed-ups provided by single instructions and some groups, presumably using Cburg as a baseline. The reader has to assume that the original Iburg produced back-end would have produced the same

baseline results. The back-end is evaluated on four benchmarks: two encryption benchmarks; an IPv6 stack; and an ADPCM application. The IPv6 stack's run-time, however, is dominated by an encryption layer, therefore the technique was evaluated on three encryption dominated benchmarks and one signal processing benchmark. Significant speed-ups were found for the encryption benchmarks but only a small speed-up was obtained for the ADPCM benchmark. This was entirely provided by parallel loads where there is already a significant amount of dedicated compiler work (see sections 2.5 and 3.2). Thus, the techniques presented have only been proved useful for encryption benchmarks, which are highly parallel and thus commonly provide excellent speed-ups in ASIP-related papers. Finally, the set of instructions evaluated were the fusion of only two unconnected simple operations. These are far smaller than the extension instructions considered in this thesis and could potentially mean this technique is inappropriate for AISE generated extension instructions.

Ebner et al. [2008] produced a technique for matching graph-based instructions based on a SSA representation. Interestingly, this technique operates on whole functions, not just basic blocks – though it can only match acyclic instructions. Instruction are specified via semantic rules, and a topological ordering amongst the rule dependencies must exist so that the convexity of matches is guaranteed (i.e. an instruction should never have to read its own output as it would then have to be scheduled before itself, which is clearly impossible). The algorithm then finds every possible (overlapping) place to use each instruction. This has a worst-case complexity of $O(\binom{n}{k})$ where n is the number of nodes in a basic block and k in the number of nodes in an instruction, however, in practice the worst-case is rarely reached. Each assignment then becomes a variable in a partitioned binary quadratic problem, and is either mapped to an integer linear program for solving optimally or is solved heuristically. The heuristics are able to find the optimal solution 99.83% of the time. This technique was evaluated on an ARM back-end in the LLVM [Lattner and Adve, 2004] compiler. It was consistently able to outperform the existing LLVM ARM back-end, but was not always able to outperform *GCC*. Consequently, the improvements found may have been opportunistically trivial due to poor optimisation choices earlier in LLVM. Finally, the ARM ISA does not contain any particularly complex instructions, therefore this technique is not shown to work with complex extension instructions. The worst-case complexity of $O(\binom{n}{k})$ makes it very likely the technique will not scale for large instructions, as when the number of nodes in an instruction (k) grows the complexity explodes.

Kobayashi et al. [2001] propose a method for automatically generating a compiler for an ASIP processor from a hardware architecture description. Their PEAS-III system allows a hardware designer to construct an ASIP by picking instructions from a provided database. This database then provides information on each instruction's functionality for code-generation purposes. Scheduling information is generated from an analysis of available resources on the

processor. This information is then used to produce a CoSy [ACE Associated Compiler Experts, 2011] back-end description. No evaluation is performed but as this is just a standard CoSy back-end there is little that is novel to evaluate. The CoSy back-end generator uses an optimal tree-based instruction selector which can also combine trees to achieve very limited support for complex instructions. This is the standard approach for compiler production in ADL systems [Brandner et al., 2007; Ceng et al., 2005; Hohenauer et al., 2004], thus although they can target customised processors they have very limited support for extension instructions.

An issue that is orthogonal to complex instruction mapping was considered by Yiannacouras et al. [2006]. They considered how processors may be specialised through elimination of hardware from the baseline processor, which may be used simultaneously with the addition of custom instructions. Initially, they investigated how much area could be saved by eliminating hardware support for instructions that are unused by specific applications, and found that area reductions of 25-60% were achievable. More interestingly, however, the same authors later considered how the compiler could be used to provide support for eliminating hardware features that *are* used by the target application [Labrecque et al., 2006]. The hardware changes made were the replacement of variable shifters with a small number of fixed shifters, the replacement of a 64-bit hi/lo multiplier with a 3-operand multiplier (requiring separate instructions for obtaining the hi and lo results, if they are both required), the elimination of load-delay and branch-delay slots, and finally a reduction in forwarding logic. The paper describes some other possibilities but does not fully evaluate them. This specific paper [Labrecque et al., 2006] is evaluated in the context of soft-processors designed for deployment to FPGAs. The same techniques, however, could be used with synthesised ASIPs to produce a smaller processor which may use less power, depending on the execution time cost of the changes. In the context of a soft-processor running on an FPGA the authors were able to obtain a 32% increase in efficiency by combining the techniques from both papers, where efficiency is described as the ratio of MIPS to the number of logic elements. The techniques described are only evaluated on a selection of kernel-sized benchmarks, therefore if the workload required of the processor is more complex then it is likely that the efficiency improvements seen will be reduced. This is because the techniques rely on eliminating features that have a small impact on a particular application, therefore the more complex an application the more likely it is to use each hardware feature.

3.2 Compilation for Dual Memory Banks

An early attempt to solve the dual memory bank assignment problem was undertaken by Saghir et al. [1996]. They produced a low-level solution that performs a greedy minimum-cost partitioning of the variables using the loop-nest depth of each interference as a priority heuristic.

The problem was formulated as an interference graph, where two nodes interfere if they represent a potentially parallel access in a basic block. The algorithm is overly simplistic, however the representation of the problem is very intuitive and has been used in many other solutions since.

Hiser and Davidson [2004] developed a highly portable memory bank assignment tool, EMBARC, that allows specification of a wide range of memory systems. A partition description language is used to specify cache hierarchies, the presence of scratchpad memories, and memory latencies and bandwidths. The assignment of variables to partitions is again handled in a greedy fashion. The most frequently accessed variables are considered first. The cost of assigning the current variable to each partition is considered and the variable is placed according to the lowest cost. The cost is calculated by considering the product of the estimated average access time to this partition and the number of references to this variable, and then factoring in expected conflicts with variables already assigned to this partition.

Gréwal et al. [2003] used a highly-directed genetic algorithm to provide a solution to dual memory bank assignment. They used a constraint satisfaction problem as a model, with hard constraints such as not being able to exceed memory capacity, and soft constraints such as not wanting interfering variables in the same memory. The genetic algorithm is then used to find the optimal result in terms of this model. This use of machine learning does not actually learn trends regarding the problem, but is more akin to solving the constraint satisfaction problem by brute force as it is re-run for every instance of the problem. Additionally, due to technical limitations this method was only evaluated on randomly generated synthetic benchmarks.

Fröhlich and Wess [2001] considered dual memory bank assignment within a genetic algorithm as part of a larger piece of work. The overall work looked at using a genetic algorithm to aid integrated code generation for a heterogeneous-register architecture with multiple memory banks (a Motorola DSP56k). The genetic algorithm is used to decide whether the result of each expression tree is stored in a register or memory, and if in memory then which bank. It is not possible to fully evaluate the effectiveness of the dual memory bank assignment used in this paper due to the highly integrated approach of the solution. Also, there is no explicit consideration of arrays or memory blocks, which are key to effective dual memory bank assignment.

Gréwal et al. [2006a] later revised their previous genetic algorithm [Gréwal et al., 2003] with a not-so-highly directed GA and a model that is more appropriate for solving by GAs. The technique was evaluated using the DSPstone benchmark suite [Zivojnović et al., 1994] where it was able to find the optimal solutions for all benchmarks in the suite. These are small benchmarks and therefore the large number of partitionings that a genetic algorithm would check means that for most of the benchmarks the compiler exhaustively checked every possible assignment. Thus, given the high computational cost of running a genetic algorithm it seems undesirable to include this approach into the run-time of the compiler. If exhaustive checking

is acceptable, for small benchmarks at least, there are more effective ways of achieving this.

Several authors have proposed integer linear programming solutions. Initially Leupers and Kotte [2001] described a method that modelled the interference graph between variables as an integer linear program and tries to minimise total interferences. This approach worked on the compiler IR after the back-end has been run once, allowing it access to very low-level scheduling and memory access information. Another approach by Ko and Bhattacharyya [2003] uses synchronous data flow specifications and the simple conflict graphs that accompany such programs. They used an integer linear program to find an assignment to memories, but for all benchmarks that the techniques were evaluated against there exists a two-colouring, so the technique is not demonstrated to work on hard problems. More recently Gréwal et al. [2006b] described a more accurate integer linear programming model for DSP memory assignment. The model described here is considerably more complex than the one previously presented by Leupers and Kotte [2001], but provides greater improvements. This model considers the size of arrays, whether operations are commutative and allows duplication (i.e. an array may be placed on both memory banks). The addition of these features means that this model is able to find the optimal solutions for the DSPstone benchmark suite.

Sipkovà [2003] describes a technique that operates at a higher-level than the previously described methods. It performs memory assignment on the high-level intermediate representation, thus allowing the assignment method to be used with each of the back-ends within the compiler. The problem is modelled as an independence graph and the weights between variables take account of both execution frequency and how close the two accesses are in the code. Several different solutions, based on a max-cut formulation, were proposed. Unfortunately, this paper does not address any of the issues created by assigning data to memories at a high-level. Therefore it is not clear that the technique is as portable as is claimed, nor that it is taking full advantage of the dual-memory capability.

The construction of compiler heuristics for dual memory bank assignment based on genetic algorithms is an application of well-known machine learning techniques in the field of computer systems. In recent years, various machine learning techniques have been studied in the context of compiler optimisation. This research has inspired novel optimisation approaches such as adaptive compilation [Cooper and Waterman, 2003] and iterative compilation [Fursin et al., 2002] where the focus is on deriving better phase orderings and tuning parameters for previously unseen programs. Cavazos and O'Boyle [2005] have investigated the use of a genetic algorithm for the generation of inlining heuristics for a Java just-in-time compiler.

Scratchpad allocation is a more general form of the dual memory bank assignment problem where the goal is to partition and assign data to small, fast, on-chip memory or larger, slower, off-chip memory. The body of work related to this problem is large but Verma and Marwedel [2007] provide a good overview of the problem and existing solutions. Panda et al. [2000]

also represents influential early work on the problem. Source-level approaches, similar to the work presented in chapter 6, have also been developed, e.g. in Falk and Verma [2004]. The two problems differ, however, as the "unpredictability" of back-end compiler optimisations is less critical for scratchpad allocation, e.g. temporary variables introduced by the compiler and register constraints do not interfere with the source-level data assignment. For the dual memory bank assignment problem considered in chapter 6 any such interference may be critical for the success or failure of the exploitation of simultaneous memory accesses.

3.3 Transformations Affecting AISE

Early efforts [Verma and Ienne, 2004] to combine code transformation and AISE have been targeted at Control Data-Flow Graph (CDFG) transformation to produce efficient arithmetic structures. This operates post-AISE, therefore does not directly contribute to the design space search, it just improves upon the result. This was the first paper to consider that extension instructions do not have to be used solely as the AISE intended – this idea was part of the motivation for the work in chapters 4 and 5 of this thesis.

The only prior work directly examining the effect of transformations on instruction set extension was written by Bonzini and Pozzi [2006]. They show that an exploration of the *if-conversion* and *loop-unrolling* transformations is successful in enabling better performing AISE. This provides good motivation for the more complete investigation undertaken in chapter 7 as it demonstrates the effectiveness of extension instruction targeted heuristics in transformation. Both the transformations considered increase basic block size which gives the AISE tool more options for extension instruction generation. The technique works by using a set of heuristics to decide where to use *if-conversion* and what *loop-unrolling* factor to apply. AISE is then used to generate instructions and several surround points in the transformation space are evaluated using these extensions instructions. This evaluation of a small transformation space is crucial for making a good choice of transformation, but it would be very difficult to extending the technique to include additional transformations. The work presented in chapter 7, however, considers a much larger transformation space (including *loop-unrolling*).

3.3.1 Source-to-Source Transformations for Embedded Systems

Due to their inherent portability and large scope, source-level transformations have been widely applied within embedded systems in areas such as code optimisations targeting I/O performance [Wang and Kaeli, 2003], energy efficiency [Chung et al., 2000; Kulkarni et al., 1998], formal verification [Winters and Hu, 2000], and, most notably, for single and multi-core performance optimisation of computationally intensive embedded applications (e.g. [Falk and Marwedel, 2004; Franke and O'Boyle, 2003a; Luz and Kandemir, 2004] and [Franke and O'Boyle,

2003b], respectively).

ROSE [Schordan and Quinlan, 2003] and Transformers [Borghi et al., 2006] are tools for building source-to-source transformation tools for the optimisation of C and C++ programs. They have been used for tasks such as serial loop optimisations and parallel message passing optimisations [Brown et al., 1999]. SUIF [Wilson et al., 1994] is a complete (source-to-assembly) compiler, but it is also capable of source-to-source transformation with a large suite of classical program optimisations available.

An empirical study of source-level transformations for digital signal processing applications is the subject of Franke and O'Boyle [2003a]. This work has also been extended [Franke et al., 2005; Agakov et al., 2006] to undertake more comprehensive studies in the context of machine-learning based adaptive compilation. The evaluation and development model used in these papers influenced the experiments in chapter 7, relating to investigating the effects of source-to-source compiler transformations. E.g. the model used by Franke et al. [2005] considers a large set of transformations on multiple target architectures and measures the probability that each transformation will be beneficial across all architectures and benchmarks.

Chapter 4

Code Generation for Complex Instructions

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."
— C.A.R. Hoare, *computer scientist, ACM Turing Award winner*, 1934–.

This chapter describes and evaluates a complex instruction mapper, *MapISE*, that has been implemented in *GCC*. This chapter treats the hardware as a fixed target based on the default processors that are generated by the AISE tools for each benchmark. Chapter 5, however, will take what has been learnt in this chapter and will use it improve the usefulness of the generated hardware.

This approach is necessary because most of the results in this chapter have significant drawbacks. For example, the gains presented may not be high enough to justify the hardware cost, the results may be too inconsistent to justify the effort of an industrial implementation or the steps taken to improve performance suggest there is a fundamental problem in the underlying system that should be addressed rather than avoided. This chapter, therefore, is used to motivate the changes presented in chapter 5.

There are a significant number of experiments presented within this chapter – this is because *MapISE* was extended with many different modes to aid in the investigation of its effectiveness. Some of these experiments show that certain modes of operation provide little change from the default mode, or make things worse. They are still presented, however, because it is appropriate to show that there are problems involved in mapping that are not simple to solve, hence the need for the changes presented in chapter 5.

Section 4.1 summarises why the techniques presented in this chapter are required, and sections 4.2–4.4 explain the implementation of the instruction mapper. Section 4.5 describes a method that the instruction mapper can use to avoid performance issues with the extended processor and this is evaluated in Section 4.6. Section 4.7 shows the results of a direct evaluation of

the instruction mapper, whereas section 4.8 evaluates the performance of the instruction mapper when presented with a processor already specialised for a different benchmark. Section 4.9 critically evaluates the instruction mapper performance and presents possible future work that is not described by later chapters.

4.1 Motivation

Most existing techniques for complex instruction mapping are designed for finding good, i.e. near optimal, solutions when using small graph-shaped instructions (see section 3.1). They do not, however, generally scale well to very large instructions – most papers perform evaluations using instructions with only two operations in them. E.g. the approach taken by Ebner et al. [2008] finds every possible overlapping use of each instruction before selecting which ones it wants to use. This only works for small instructions, e.g. to map a single instruction (of two, three or eight nodes) in a single basic block of one hundred nodes has a worst-case complexity of $\binom{100}{2} = 4950$, or $\binom{100}{3} = 161,700$ but $\binom{100}{8} = 186,087,894,300$. Although in reality it is unlikely that the worst case will occur, large instructions are still clearly not practical with this class of technique.

It is apparent that any instruction selector supporting very large instructions will encounter significant problems. The tool described and evaluated in this chapter, *MapISE*, therefore focuses solely on the part that full instruction selectors cannot handle: large instructions. *MapISE* operates at a high-level and identifies places to use extension instructions. The existing backend then takes care of code-generation for all parts of the program that are not covered by extension instructions.

4.2 Mapping by Graph-Subgraph Isomorphism Checking

4.2.1 Overview

To map extension instructions to a given input program the problem is defined in terms of graph-subgraph isomorphism checking such that each basic block is a graph and each extension instruction is a subgraph. If a subgraph is isomorphic with a graph then the corresponding extension instruction may be used in the corresponding basic block.

The graphs for the basic blocks are produced from the compiler IR and the graphs for the extension instructions are produced from a provided XML specification. The operations performed (e.g. an addition, or a bitwise XOR) are represented as vertices and the dataflow between operations as edges. As an example, the simple C expression "`a += b * c`" would result in the graph shown in figure 4.1. The variable names are not directly preserved but implicitly exist as dataflow dependencies.

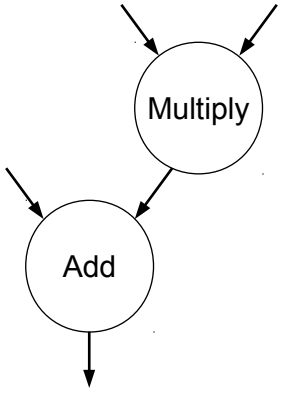


Figure 4.1: *The expression "`a += b * c`" represented as a dataflow graph.*

For each basic block every extension instruction is tested to determine whether it may be mapped or not. The extension instructions with the largest expected benefit are tested first to try and maximise the overall benefit, but as this a greedy approach the mappings found might be sub-optimal. The test determines that an instruction may be mapped to a basic block if the instruction's subgraph is isomorphic with the basic block's graph.

A subgraph is isomorphic with a graph if both the vertex types and the shape of the subgraph may be mapped to the graph. A vertex in a subgraph can be mapped to a vertex in a graph if they implement the same functionality, e.g. they both implement an integer addition function. The shape of the graph is determined by the edges between vertices. Traditionally edge order does not matter in graph-subgraph isomorphism checking, but the order of inputs to arithmetic and logical operations does matter (except for commutative operations). Therefore an additional check is added to ensure that edge order is preserved for inputs to non-commutative operations.

Once all mappings have been found a simple register allocation pass assigns the input and output vectors of each extension instruction to specific vector registers. Whereas the graph-subgraph isomorphism checking operated on a per-basic block level, this pass operates on a per-function level. This allocation pass attempts to minimise the amount of data movement required, primarily by minimising the frequency of live-range interruptions. Finally, each vector register access is assigned a permutation to load the register with. The permutation which minimises the amount of data movement is picked (this may be the identity permutation).

At this point there will remain many basic block operations that have not been mapped to an extension instruction. This is the intended behaviour as there is no guarantee that a complete mapping of basic block operations to extension instructions will even exist. Once the above pass, *MapISE*, has completed the existing compiler back-end is used to map all remaining basic block operations to standard RISC instructions on the baseline core. The back-end also performs register allocation around the existing vector register allocations and schedules both

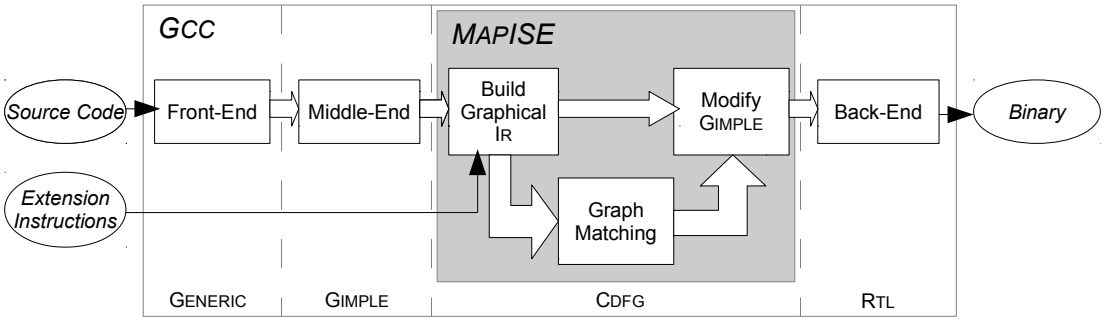


Figure 4.2: *The structure of passes within* GCC *and* MapISE*.*

the extension and baseline RISC instructions.

4.2.2 Integration into *GCC*

MapISE is implemented in *GCC* 4.2. The primary reason for this is that the *EnCore* processor that is used as the baseline processor in the AISE framework, implements the ARCompact ISA. The most complete compiler which supports ARCompact is the ARC version of *GCC* 4.2. Unfortunately it has not been ported to newer versions of *GCC*.

GCC operates in four main stages: a front-end, a high-level middle-end, a low-level middle-end and a back-end. The pass presented in this chapter, *MapISE*, runs at the end of the high-level middle-end while the IR is still in SSA form, as shown in figure 4.2.

GCC uses several IRs. The front-ends translate the input source code into GENERIC, a high-level IR. This is then lowered into GIMPLE, a medium-level IR used by the high-level middle-end. GIMPLE can be used in both an SSA and a non-SSA form, but most high-level optimisation passes operate on the SSA form, including *MapISE*. The GIMPLE form is then lowered again into a low-level IR: RTL (register transfer language). Low-level optimisation and target specific passes operate on RTL. Finally the back-end performs instruction selection on the RTL form ensuring all operations have a one-to-one mapping with assembly. These operations are also annotated with register and scheduling constraints which are used by the register allocator and the scheduler respectively before the RTL is finally converted into assembly.

The instruction mapper presented in this chapter, *MapISE*, exploits *GCC's* support for extended inline assembly. This is required to work around a lack of support for several specific features that an extension instruction mapping pass requires. The most unusual requirement is the need to support an arbitrary number of extension units with a single compiler binary. If any changes are made to a *GCC* back-end then *GCC* must be recompiled. This only takes about a minute for the second recompile onward (i.e. `make` only recompiles what has changed), and therefore is acceptable for the purposes of running the experiments in this thesis. If this technique is to be usable in a real-world context, however, then expecting the compiler to be

recompiled not acceptable. A processor designer or application developer may be evaluating several hundred extension configurations as a design space exploration exercise. Even if the process is automated this would likely still hinder productivity. The implementation described in this chapter avoids this by taking a description of the extension unit as part of its input. This way, the requirement of deploying a single compiler install can be satisfied, while still supporting an arbitrary number of extension units. The second unusual requirement is the need to fit irregular data into vector registers, and then exploit permutation units attached to them.

Both of these requirements can be satisfied by inserting ASM statement operations into the IR. These are usually generated by the front-end when a program contains inline assembly. No other passes in *GCC* insert ASM operands into the IR but the alternative approaches would require *GCC* to be recompiled with each change. The use of ASM nodes also allows vector registers and permutation units to be used, with only a minimal amount of information about them encoded in the back-end. The seven vector registers mentioned in section 2.2 are mapped to the scalar registers `r32` through to `r59`. The *EnCore* back-end in *GCC* has been modified to have 28 additional singleton register classes `a32` through to `a59`. Each input and output to an ASM operand must be assigned to a specific register class. Usually this would mean "the class of all general-purpose registers" or similar, but *MapISE* takes advantage of the singleton classes to force *GCC* to place data in specific registers. This then allows the inserted assembly string (inside the ASM operand) to operate on vector registers complete with permutation units, even though the back-end does not know they exist.

If *MapISE* mapped any extension instructions onto a function, then once the mapping pass is complete the pass manager is configured to re-run *loop-invariant code-motion* as the mapping pass introduces various temporaries which may benefit from being hoisted.

MapISE contains approximately 12,000 lines of C code, or 11,000 lines without assertions and other debug code. This suggests a level of complexity similar to the auto-vectoriser in *GCC* 4.2, which has approximately 9000 lines of code. The code which builds the graphical IR (CDFG) from GIMPLE, and then produces an XML file for *ISEGen* to process, is a further 2500 lines.

4.2.3 Construction of Graphical Intermediate Representation

As *MapISE* is based on graph-subgraph isomorphism checking, it must operate on a graphical IR. For this purpose an IR called CDFG (Control Data-Flow Graph) was specified by the author of *ISEGen*. As this IR was not going to be used for compilation it did not need to be complete. For a given program the CDFG representation is a list of basic blocks with no control-flow information describing how they link together. This is ideal for the purposes of AISE tools since they only look for dataflow graphs within basic blocks, control flow information is extraneous. The 'C' in the name CDFG is there because support for control flow was intended to be added,

but ended up being unnecessary.

Each basic block is a list of nodes and edges and each possible node-type has a one-to-one mapping to some type of GIMPLE node. The converse is not true, however, GIMPLE has many types of nodes not supported by CDFG.

For the purposes of AISE, a pass was created in *GCC* that iterates over the GIMPLE representation of a program and produces a CDFG representation. The representation is then serialised into XML and the resultant information transferred to the AISE tools.

MapISE also needs to operate on CDFG because the extension instruction definition is provided to it in XML CDFG format. *MapISE* therefore processes GIMPLE to create a list of CDFG basic blocks and parses the provided XML to create a list of CDFG instructions. The manner in which these two tasks occur is completely mechanical and therefore not discussed here. At this stage however *MapISE* has to do some additional post-processing of the CDFG.

Firstly, cast nodes need to be removed as the XML CDFG provided will have already had this done on the basis that the hardware implementation of the extension instructions operate on 32-bit arithmetic. Secondly, pointer aliasing information is used to add virtual dependencies between CDFG nodes if *GCC's* virtual use and defines state that two nodes are dependent. This means that if operation X writes to memory, and operation Y reads from what may be the same memory location then Y has a virtual dependence on X. This means that Y must occur after X. See section 4.9.1 for a discussion on the effects of *ISEGen* missing this information.

Finally, the VFlib implementation of the VF2 algorithm [Cordella et al., 2004] that is used to perform graph-subgraph isomorphism checking requires graphs to be in its own format, so for every basic block and every extension instruction definition an additional representation is built.

4.2.4 Matching Subgraphs

To find where extension instructions may be mapped, a greedy search strategy is used. The extension instructions are sorted by their expected gains. Then, for each basic block every instruction is iterated over. The basic blocks are simply iterated over from start to end, the instructions, however, are ordered according to their expected benefit – the best instructions are considered first. Each basic block and extension instruction pair is passed to VFlib and if the instruction is a sub-graph of the basic block then a match is recorded. Every CDFG node that was just mapped to an extension instruction gets marked as such to avoid a node being mapped to multiple instructions. The same pattern that was just mapped is retried, in case the same pattern may be reused. This process repeats until every extension instruction has been checked.

When VFlib finds a place to map an extension instruction, it is necessary to check the mapping is viable prior to its application. As a consequence of *MapISE* supporting disjoint exten-

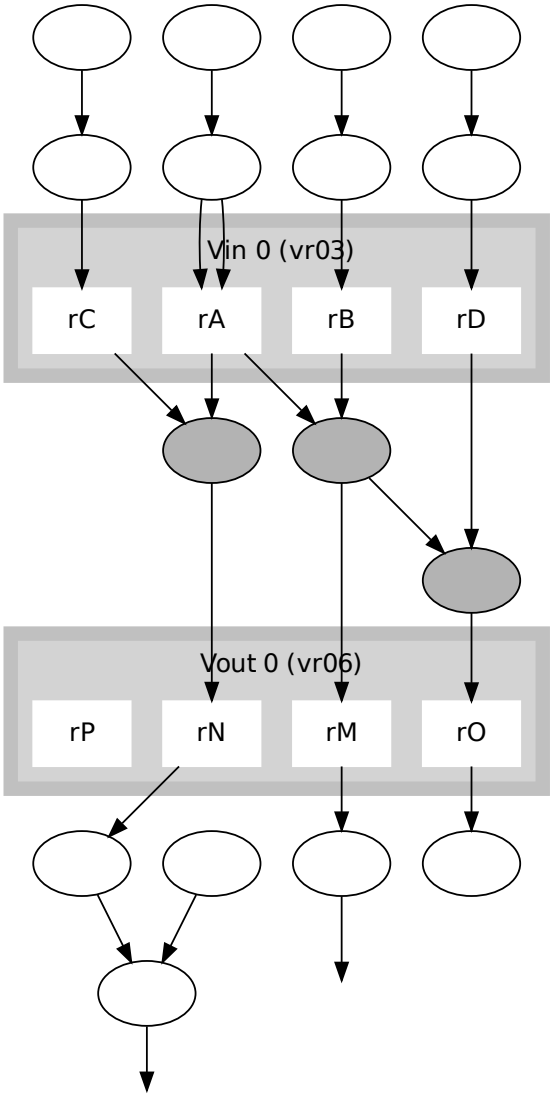


Figure 4.3: *A small extension instruction mapped to DSPstone* matrix2.

sion instructions it is possible for VFlib to find mappings which violate convexity constraints. If convexity constraints are violated then the key offending node is marked as unusable and the extension instruction is tried again. This is allowed to occur a maximum of 10,000 times per basic block/extension instruction pair. Convexity constraints are quite common: across the set of 179 benchmarks it was observed that matching was re-run 279,851 times due to convexity violations and the 10,000 iteration limit was reached 97 times. Convexity violation can occur when one of the two (or more) disjoint parts of an instruction become indirectly dependent on the other part of the instruction. The result is that the extension instruction must be scheduled both before and after the intermediary node(s): a clear impossibility.

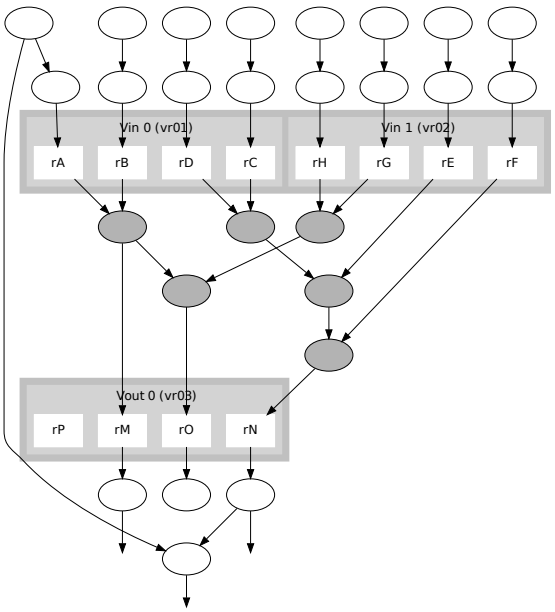


Figure 4.4: *A medium-sized extension instruction mapped to SNURT* jfdctint.

4.2.5 Determining if Two Nodes are Equivalent

A function is provided to VFlib which when given one node from the graph (basic block) and one node from the subgraph (extension instruction definition) it determines whether or not they are equivalent. As this function is performance critical (it accounts for 58% of *MapISE's* run-time, see table 4.1) it attempts to establish non-equivalence as quickly as possible.

If the nodes do not perform the same operation, have different data types or the basic block node takes a 64-bit value as input, then the nodes are not equivalent. If these nodes are constants then they must have the same value, or they are not equivalent.

If multiple extension instruction nodes read from a single input register, such as `rA` in figure 4.3, then an additional node must be inserted into the extension instruction definition to merge the two edges. Without this the four-input instruction in figure 4.3 may attempt to fit five values into four slots.

If the basic block node has already been mapped to another extension instruction then it cannot be mapped to this one. If the SSA name that the basic block node writes to is used outside of the current basic block then the extension instruction node must write its value to a vector register or they are not equivalent.

The two nodes must have the same number of inputs and the same number of outputs. If a node has two inputs then the predecessors of both nodes must be similar (same operator type, same data type). This is necessary because the VF2 algorithm does not consider edge order, it would consider $A - B$ to be equivalent to $B - A$. If the current node is commutative and the predecessors do not match then they may be swapped and rechecked.

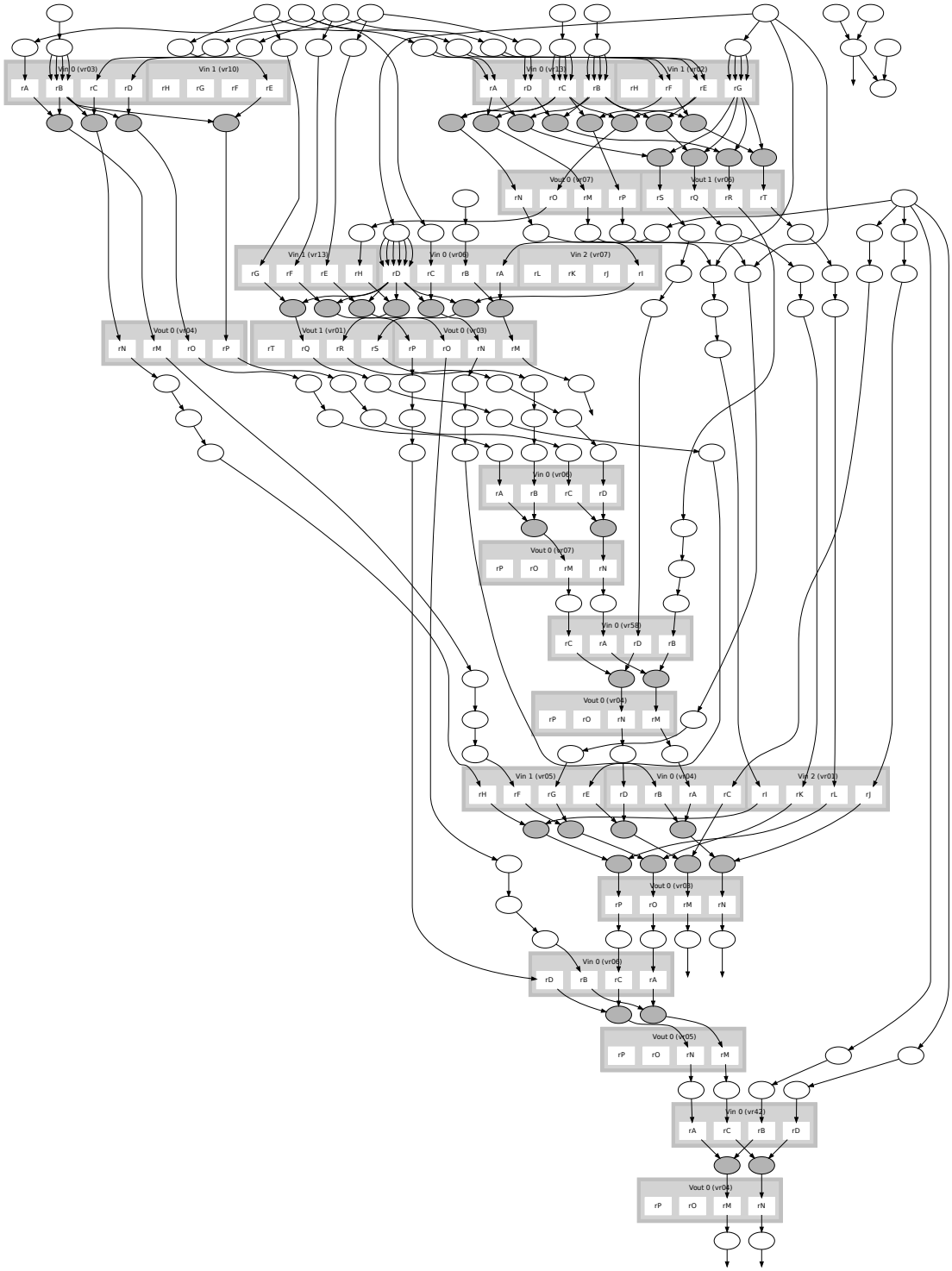


Figure 4.5: *Multiple extension instructions mapped to Crypto* aes.

4.2.6 Exploiting Matches

CDFG is an incomplete IR and therefore it is not possible to convert it back into GIMPLE. Since every CDFG node is linked with a GIMPLE node, the mappings annotated on the CDFG can be used to determine which parts of the GIMPLE to remove instead.

The vector register assignment and the chosen permutation is then used to determine which scalar registers the compiler must place the inputs in, and which scalar registers the outputs will be found in. Although the assembly will handle this via vector registers, *GCC* must use the scalar registers to interact with the extension unit. The inline assembly that is being inserted will describe the mapping between SSA names and registers.

A side-effect of performing mapping on a graphical form, and then inserting the extension instruction into the linear IR is that after the inline ASM has been inserted the IR may no longer be in a valid schedule. It is, however, guaranteed that a schedule exists, therefore a simple scheduling pass is used to reorder the GIMPLE into a valid schedule. This is not associated with the instruction scheduling performed by the back-end since there is no concept of latency in GIMPLE.

4.3 Allocation of Vector-Registers

The use of vector registers on *EnCore* is defined in section 2.2.2.

Whereas instruction mapping operates per-basic-block in *MapISE*, vector register allocation is performed per-function.

Vector register allocation is done in a greedy manner in two phases. In the first phase all input vectors are assigned a register, and in the second phase output vectors are assigned.

For each phase a list of every unassigned vector is kept. A score is calculated for assigning every vector to each of the seven vector registers. The assignment with the highest score is chosen, that vector is removed from the unassigned list and the process is repeated until the list is empty.

To calculate the score several features are combined. Firstly, the profiling execution count is used to strongly bias the score to performing assignments to vectors in "hot" code first. If profiling data is unavailable this has no effect. This feature only varies per-vector, the assignment being tested has no effect (i.e. the execution count is identical for all seven tested assignments). If this is the output vector phase and the current vector's data is read by other extension instructions, which are allocated to the vector register assignment currently being tested, then provide a score boost. If the other vector is within the same basic block this is a large boost, if it is in a different basic block or the same basic block but for the next loop iteration then this is a small boost.

Finally a load-balancing cost is introduced, each time an input vector is assigned, for each

constant or scalar input it reads, a penalty is stored next to that vector assignment. Every time that vector assignment is assessed that penalty is subtracted from its score. If that assignment is picked anyway, the penalty is increased according to how many constants or scalar inputs the new assignment reads. This eliminates the algorithm's bias towards over-using the lower registers and also helps to avoid having loop-invariant hoisted variables being overwritten on every iteration of the loop.

4.4 Permutation of Vector-Register Elements

While vector register assignment uses a greedy approach, permutation choice is handled exhaustively as there are only 24 ways to permute four elements. The *EnCore* hardware only actually provides 9 permutations (including the identity permutation). The remaining 15 permutations are handled in a two-step process since they can each be reached by performing two permutations in sequence: once on a vector move, and then the second on the extension instruction. This tool would be able to trivially target all 24 permutations directly if they were accessible.

Applying a permutation on a vector move does, however, introduce a complication. When a permutation is applied to the input of an extension instruction the reordering is not saved. The results of the vector move, however, are written to the register file. This would mean that SSA variables are now in different registers than before the extension instruction was run. This effect can be expressed in the extended ASM operand, but doing so causes *GCC* to insert scalar move instructions to move all the data back after the extension instruction has run. This is necessary because register assignments cannot rotate through loop iterations, the assembly code executed is the same for every iteration, so the data needs to be in the same registers.

To work around this `vr06` and `vr07` become dedicated scratch registers, if two-step permutations are enabled. The vector move instruction writes to `vr06` or `vr07` and the extension instruction then reads from that register instead of the vector move's source. The extended ASM operand states that the registers `r51` through `r59` (the scalar mapping or `vr06` and `vr07`) get overwritten (or "clobbered" in *GCC* parlance) – this stops *GCC* from using these registers. The unfortunate effect of this is that there are only five vector registers remaining, which is enough for a three-input two-output extension instruction, but leaves little room for the register allocator. Because of the default register allocator only uses one-step permutations.

Permutations are chosen based on which one will take fewest cycles. The source of each of the vector's four input variables is traced. If any of the sources are a use of the same vector register then its "channel" is noted (i.e. which of the four lanes in the vector it is writing to). Writes to other vector registers are not considered, as a move would be required regardless of the permutation. Scalar registers or constants are not considered because the compiler can place

them in the correct register directly. If two "channels" do not link then a scalar move instruction (`mov`) will need to be inserted The aim is to minimise the number of moves necessary, so each of the eight permutations are considered. If two-step permutations are enabled then the remaining sixteen permutation possibilities are also considered, with a one-cycle penalty to represent the vector move. The permutation that takes the fewest cycles to load the input data is chosen.

4.5 Eliminating Poor Mappings

A simple heuristic is optionally added to *MapISE*. When the list of extension instructions is provided each instruction has two additional pieces of data: the number of cycles it takes to execute the instruction on the extension unit, and the number of cycles that *ISEGen* estimated it would take to run the equivalent code using baseline processor instructions. These are referred to as the "hardware cost" and the "software cost" respectively. Subtracting the hardware cost from the software cost provides the predicted benefit of each extension instruction. The *ISEGen* model, however, does not consider register allocation overhead.

This heuristic assumes each input variable will take, on average, 0.5 cycles to configure, and each output variable will on average result in 0.25 cycles of work. Additionally, if the extension instruction provides the result of a comparison as an output, then an additional cycle is required to compare that value against zero before it can affect control-flow. This additional cost is subtracted from the instructions predicted benefit and if the result is zero or less the instruction is discarded.

4.6 Evaluation Methodology

MapISE is evaluated using the toolchain described in section 2.2. Each of the 179 benchmarks had cycle counter annotations added to them so as I/O or book-keeping code could be excluded from the performance evaluation. Each benchmark was compiled with vanilla *GCC* and with *MapISE*. The speed-ups presented are the performance of the *MapISE* produced code relative to the *GCC* code.

Profiling information is used by *MapISE*, but mostly only for the purpose of skipping basic blocks or functions which are never executed. It is also used as part of the register allocators priority heuristic (see section 4.3). The primary purpose of including profiling information is because *ISEGen* requires this information. This has two effects, firstly *ISEGen* only processes basic blocks which are executed, so if *MapISE* does not do the same then *MapISE* mapping statistics will be needlessly inflated even though it would offer no additional speed-up. Secondly, both tools are influenced by *GCC's* optimisation decisions. When profiling data is available *GCC* will treat hot and cold blocks differently, applying different compiler transforms. If

MapISE is to see the same view of the code that *ISEGen* does then it must allow *GCC* to use profiling data (even if it were to not use it itself).

The experiments were run on a Linux system with two dual-core 3.0GHz Intel Xeon processors and 4GB of memory. For experiments where run-times are reported only a single core was used to ensure timing consistency.

4.6.1 Presentation of Results

Many of the charts presented in this section are subsets of more complete charts found in appendix B. The subset included in the smaller versions is consistent throughout the thesis and was chosen such that in figure 5.9 the average speed-up of the subset and the average speed-up of the complete set are similar. This property does not hold for every chart, however. Each chart which is a subset has this stated in the caption. When average speed-ups are discussed in the text the average for all 179 benchmarks will be used, which is called "FULL AVERAGE" in the charts. The charts also include the average for the subset ("AVERAGE") because in many charts the "AVERAGE" and the "FULL AVERAGE" are different. Some charts also include the geometric-mean to provide a second perspective.

The arithmetic mean is referred to as "AVERAGE" in charts, and the geometric mean is referred to as "GEO-MEAN". If a chart does not contain a "GEO-MEAN" entry then that indicates that the data-set contains the value zero. For example, in this chapter "Speed-Up" graphs are displaying ratios and thus are never zero, graphs counting the number of extension instruction mappings, however, may contain zero values and thus geometric means are not displayed for those graphs.

4.6.2 Consideration of Floating Point Hardware

All results presented in this chapter (with the exception of figure 4.6) and chapter 5 will assume the presence of a hardware floating point unit in both the baseline and the extended processor. Although the hardware described in section 2.2 does not contain a hardware floating point unit it is necessary to assume one exists to allow accurate evaluation. Figure 4.6 is used to justify this.

Figure 4.6 shows the speed-up obtainable from either adding a hardware floating point unit or extension instructions (which may or may not contain floating point operations). Although the hardware floating point unit has no effect on integer benchmarks, it can have a significant effect on floating point benchmarks (e.g. SNURT *qurt* or UTDSP *compress*). Extension instructions can also obtain large speed-ups for many floating point benchmarks, e.g. SNURT *qurt* performs better with instruction extensions than with a floating point unit. Although these are genuine speed-ups that may be obtained by automatically extending the baseline processor, it is difficult to use these results to evaluate the usefulness of extension instructions.

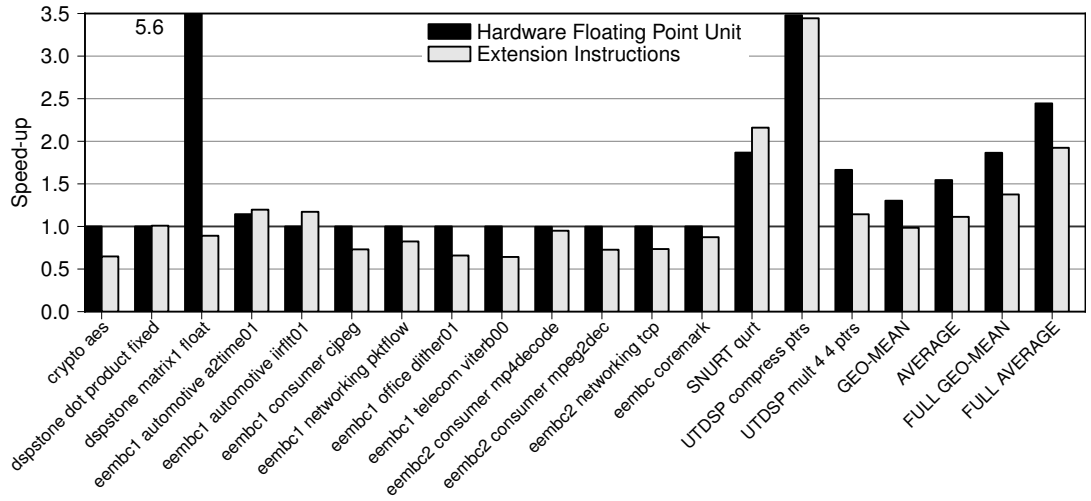


Figure 4.6: *A comparison of a hardware floating point unit with extension instructions. In this chart the baseline processor, that speed-ups are calculated from, has no floating point hardware. The left bar of each benchmark is the speed-up obtained by adding a floating point unit to the baseline processor, the right bars are the speed-ups obtained by adding extension instructions.* Note: the full version of this chart is figure B.1 on page 156.

For example: UTDSP *compress_ptrs* is improved by a factor of 3.44x, which may seem like an excellent result, but the majority of the speed-up comes from implementing floating point operations in hardware (as shown by the floating point unit providing a similar speed-up of 3.48x). The fact that implementing floating point in hardware provides large speed-ups is well understood and although automatically providing a customised unit of hardware is somewhat novel this does not outweigh the evaluation problems that are introduced.

There are three problems, the first has just been mentioned: the benefits and costs of implementing floating point in hardware are already fully understood.

The second problem is all floating point benchmarks are highly amenable for acceleration by extension instructions when compared against integer benchmarks, so their importance in the presented results would be inflated compared to their real-world significance.

The third problem is that a standard hardware floating point unit actually outperforms extension instructions. This can be seen in figure 4.6 where the average speed-up from adding a floating point unit exceeds that obtained by adding extension instructions (average speed-ups of 2.44x and 1.92x respectively). This is because it is always faster to map floating point operations to hardware and with a floating point unit all such operations can be mapped. The instruction set extension generator, however, will only construct complex instructions which combine several operations. This means that many floating point operations will not be mapped

to extension instructions (e.g. a single floating point operation in a small basic block). These floating point operations will therefore need to emulated in software (i.e. a long sequence of integer operations), which is significantly slower than hardware.

In addition to these evaluation issues a hand-designed floating point unit is likely to take less processor die space than floating point extensions and will be highly reusable in the event of changes to the target applications. Thus, for floating applications a floating point unit is usually a better choice than extension instructions alone, though combining the two may be desirable.

For evaluation purposes all experiments in this chapter, chapter 5 and section 6.1 therefore assume the presence of a floating point unit and add extension instructions based on this premise. Although the *EnCore* processor that is used as a baseline does not actually have a floating point unit, the ARC ISA that it implements does contain floating point instructions. These are fully supported by the cycle-accurate simulator, therefore the existence of a floating point unit is a reasonable assumption to make. It is worth noting that *ISEGen* makes the same assumption for similar reasons, therefore the instructions it generates will not be biased towards floating point operations.

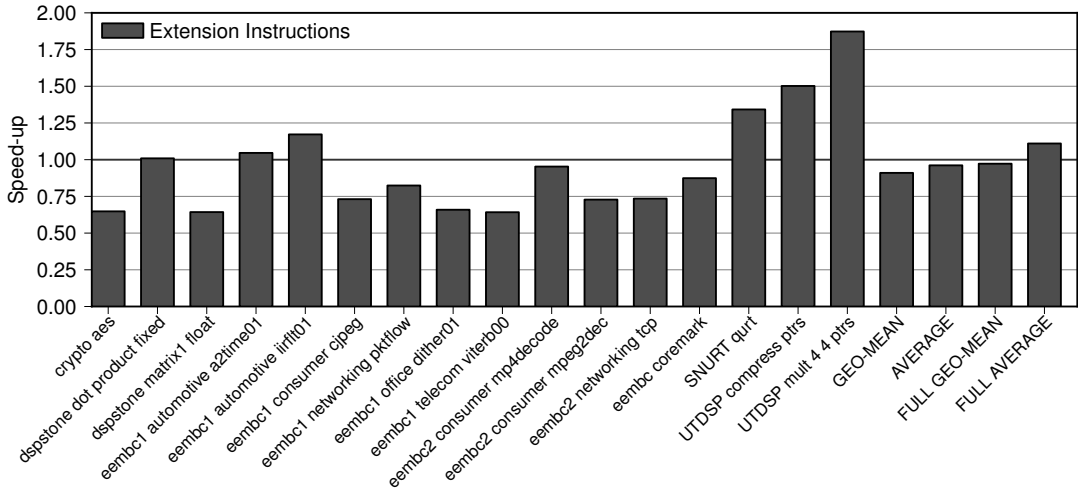
4.7 Results

4.7.1 Default Mapping

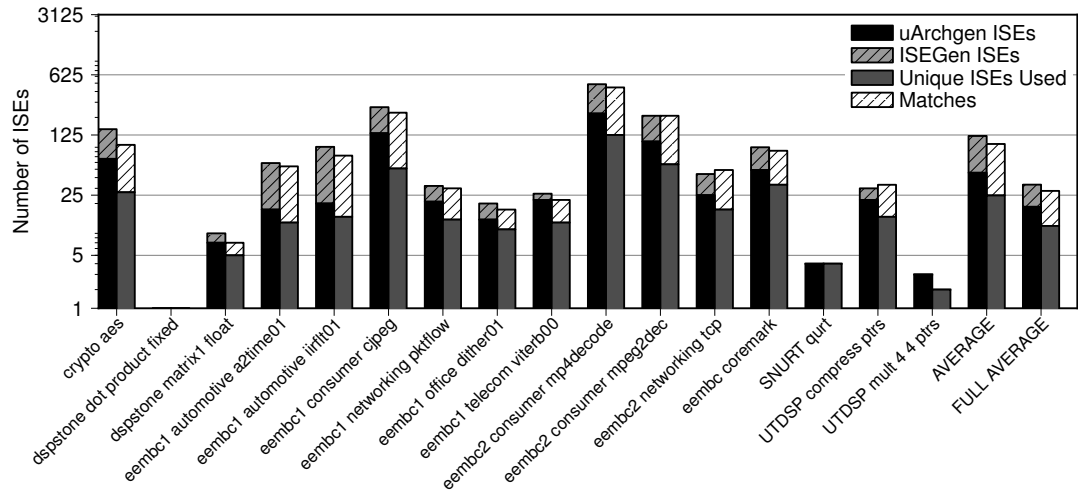
Figure 4.7 shows the default performance of *MapISE*. An overall average speed-up of 1.11x is obtained and an average of 28.01 extension instructions are used per benchmark or 10.98 unique extension instructions used on average.

Figure 4.7(a) shows the speed-ups obtained for each benchmark. The most striking feature of the chart is that more benchmarks slow-down (have a speed-up below 1.0) than speed-up. The reasons for this are covered in section 4.9.4. Attempts to rectify this issue are covered in section 4.7.3 and chapter 5.

Figure 4.7(b) compares the quantity of extension instructions that *MapISE* was able to use against the number that *ISEGen* and *uArchGen* produced. *ISEGen* finds instructions, *uArchGen* eliminates duplicate instructions within the set that *ISEGen* found – so the *uArchGen* results are the number of unique extension instructions, the *ISEGen* numbers are the total number of extension instruction sites found. *ISEGen* finds, on average, 18.30 unique extension instructions per-benchmark that may be used at 33.12 sites. *MapISE* uses 10.98 unique extension instructions per-benchmark at 28.01 sites. The reasons for *MapISE* not using 40% of the extension instructions that *ISEGen* produces are covered in sections 4.9.1 and 4.9.3.

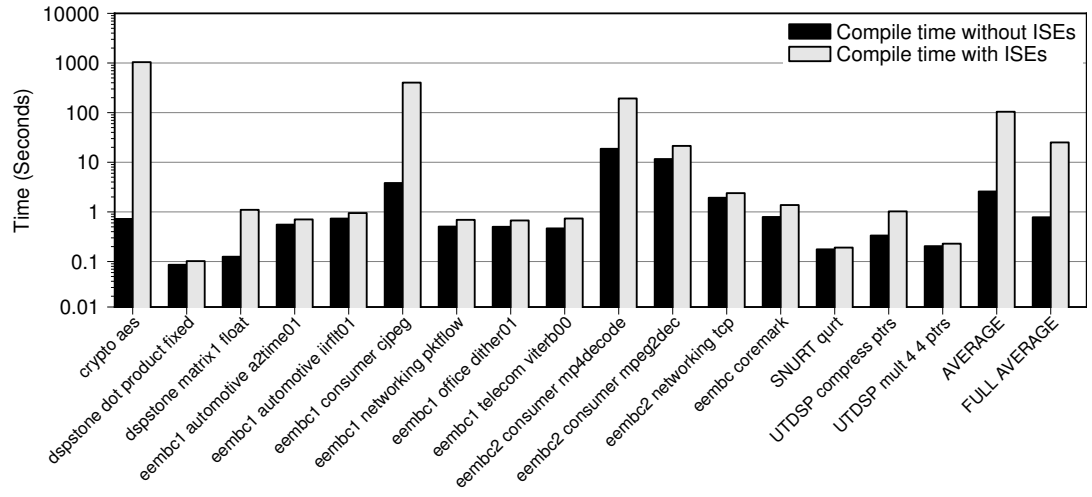


(a) The speed-ups provided by adding extension instructions specialised to each benchmark. Note that unlike figure 4.6 the baseline processor used to calculate speed-up from in this (and subsequent) graphs *does* have a floating point unit. *Note: the full version of this chart is figure B.2 on page 158.*

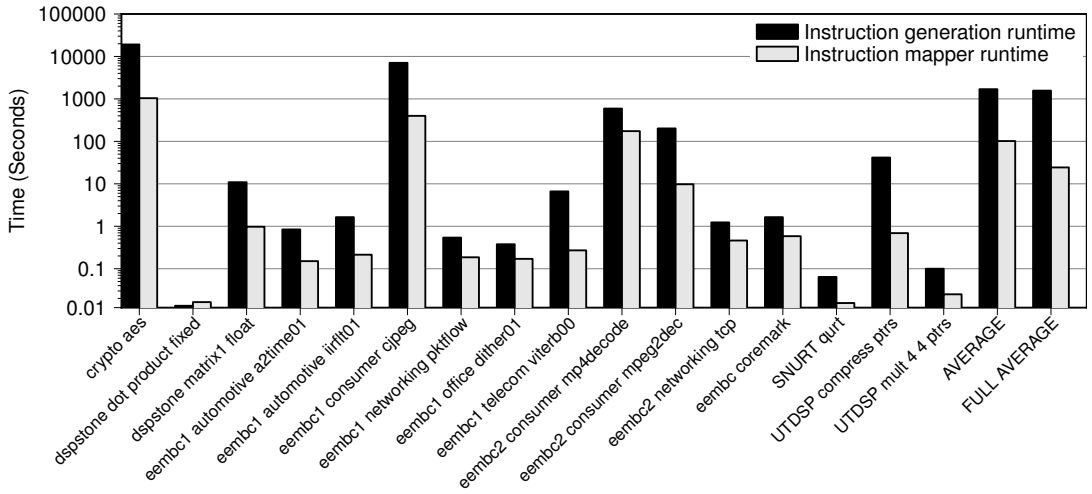


(b) Number of extension instructions found and used. Note that this chart has a logarithmic scale. The stacked bars on the left are the number of extension instructions found by the AISE tools. The lower bar is the number of unique extension instructions, the upper bar is the number of non-unique extension instructions found. The stacked bars on the right describe the equivalent for *MapISE*. The lower bar is the number of unique extension instructions exploited, the upper bar is the total number of sites where extension instructions were used. *Note: the full version of this chart is figure B.3 on page 160.*

Figure 4.7: *The performance of* MapISE *with default settings. Each benchmark is using its own extension unit specialised for it by the AISE tools.*



(a) The time taken to compile each benchmark with the standard compiler (left-hand bar) or with *MapISE* (right-hand bar). *Note: the full version of this chart is figure B.4 on page 162.*



(b) The left bar for each benchmark is the time taken to generate extension instructions (the run time of *ISEGen*, time to construct the hardware of extension unit is not included). The right bars are the length of time that *MapISE* adds to the compile time (over *GCC* alone). *Note: the full version of this chart is figure B.5 on page 164.*

Figure 4.8: *Run-time of* MapISE *compared to other tools. Note that both charts have a logarithmic scale.*

4.7.2 Timings

Figure 4.8 provides information about how long *MapISE* takes to run on each benchmark, compared against vanilla *GCC* and *ISEGen*. On average *MapISE* (*GCC* plus the mapping pass) takes 25.1 seconds to run per-benchmark, whereas *GCC* alone has an average run-time of 0.78 seconds. *MapISE's* run-time varies from a fraction of a second up to 1040 seconds (about 17 minutes). *GCC* runs slowest on the largest benchmarks but *MapISE* runs slowest on the

benchmarks with the largest basic blocks, e.g. *GCC* compiles Crypto *aes* quickly, but it is one of the hardest benchmarks for *MapISE* to compile.

Figure 4.8(b) compares the time it takes *ISEGen* to identify instructions for a benchmark against the time it takes the mapping pass inside *MapISE* to exploit them. The run-time for the mapping pass is roughly equivalent to: $RunTime(MapISE) - RunTime(GCC)$. It can be seen that *MapISE* is over an order of magnitude faster than *ISEGen* with average run-times of 24.32 seconds and 1565 seconds respectively. *ISEGen* has a maximum run-time of 98,244 seconds on Crypto *des* (approximately 27 hours), thus clearly *MapISE* has a reasonable worst-case run-time for the complexity of the problem.

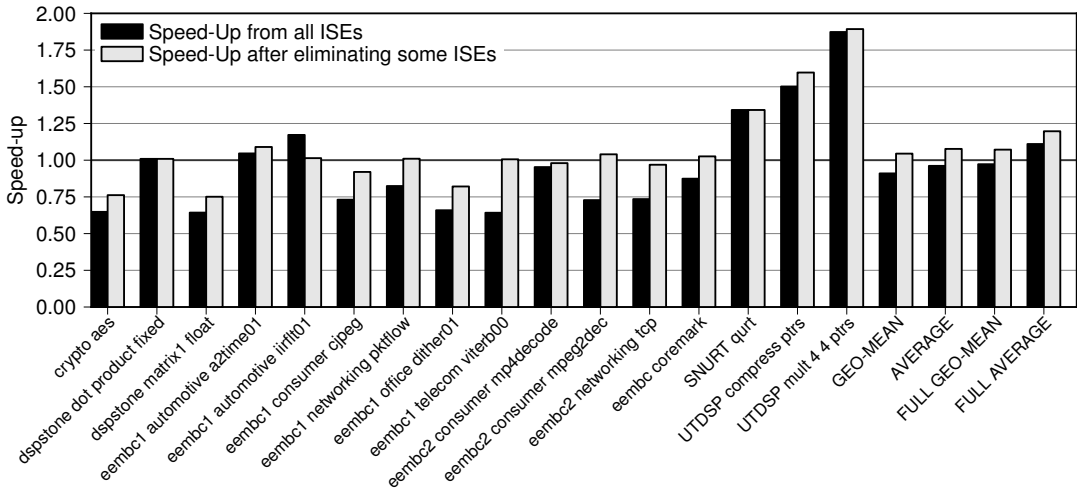
Task	**Time (s)**	**Time %**	**Sub-Task**	**Time (s)**	**Time %**
Build IRs	114.43	0.8%	Parse XML ISEs	96.92	0.7%
			Build CDFG	5.71	0.1%
			Build VF2 graphs	3.74	0.0%
Mapping	13,733	97.5%	VF2	5307.3	37.6%
			Node Comparison	8164.0	58.0%
			Viability Checking	244.83	1.7%
Register Allocation	27.50	0.2%			
GIMPLE Modification	208.84	1.5%	Scheduling	208.04	1.5%

Table 4.1: *The total time spent in each sub-pass of* MapISE*. The timings are from the summation of all 179 benchmarks.*

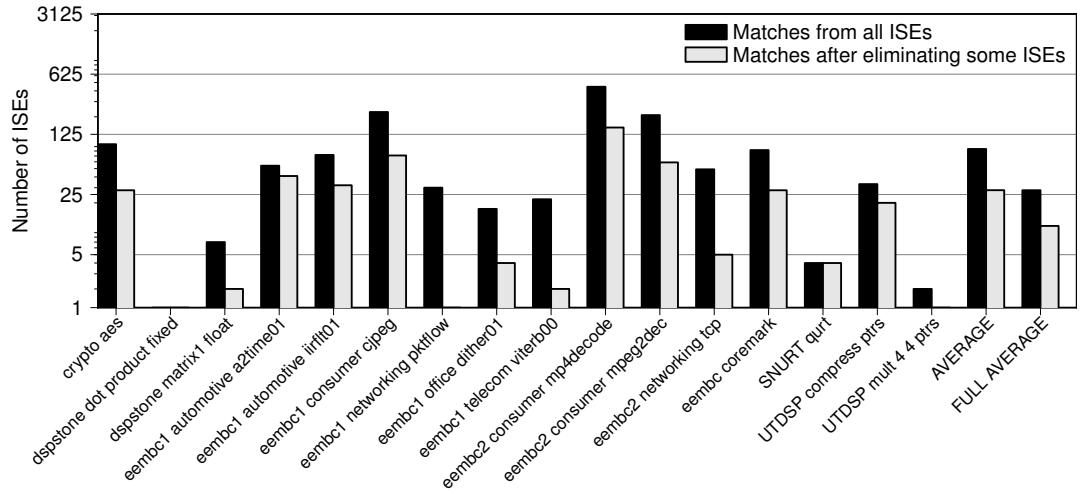
Table 4.1 breaks-down where the mapping pass within *MapISE* spends its time. The numbers presented are for all 179 benchmarks summed, which took 14,083 seconds (approximately 4 hours). It can be seen that the vast majority (97.5%) of the passes run-time is spent on graph-subgraph isomorphism ("mapping"). Within the "mapping" sub-pass 38.6% of the run-time is spent in the VF2 library [Cordella et al., 2004] and 59.5% of the run-time is spent in the node comparison function that *MapISE* provides to VF2. The remaining 1.9% of the time is spent ensuring that the mappings that VF2 finds are viable instruction candidates.

4.7.3 Eliminating Poor Mappings

Figure 4.9 shows the results of implementing the technique described in section 4.5: eliminate extension instructions that are likely to slow the code down. This works as was hypothesised: fewer instructions are used, but a greater speed-up is achieved. The number of extension instructions used falls dramatically, from an average of 28.02 per-benchmark with the default settings to 10.79 when eliminating ineffectual extension instructions. The speed-ups achieved,



(a) Speed-up obtainable with default mapping (left-hand bar) or when eliminating poor mappings (right-hand bar). *Note: the full version of this chart is figure B.6 on page 166.*

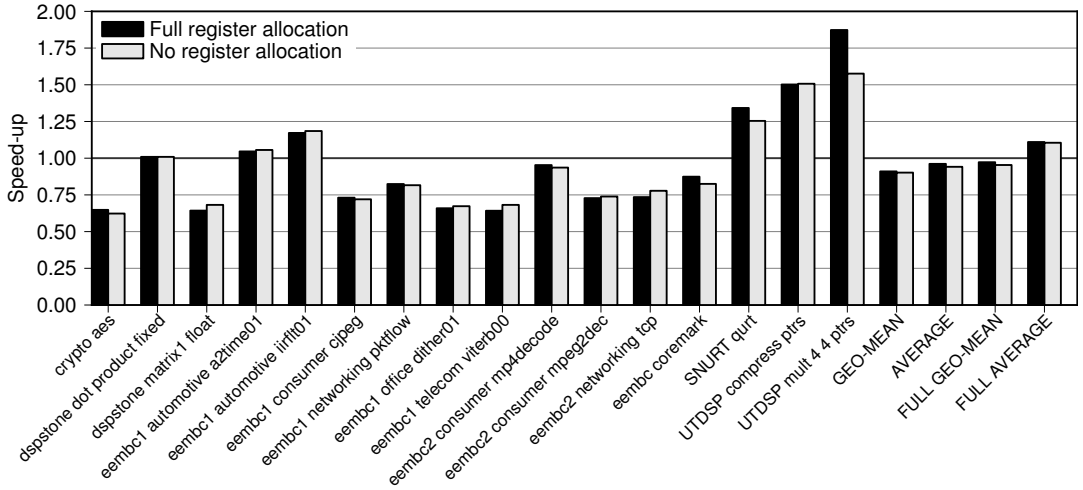


(b) The number of sites where extension instructions are found for default mapping (left-hand bar) or when eliminating poor mappings (right-hand bar). *Note: the full version of this chart is figure B.7 on page 168.*

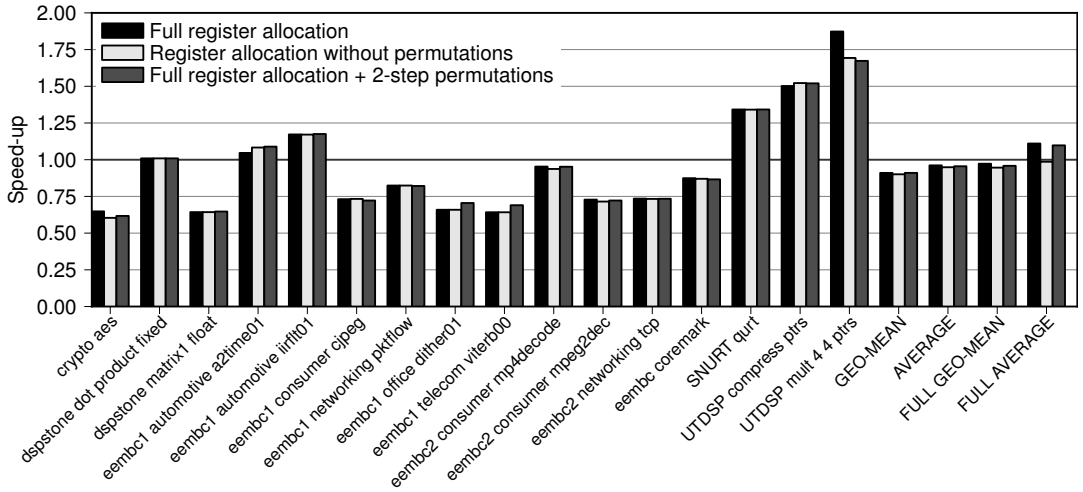
Figure 4.9: *An evaluation of eliminating poor mappings in* MapISE*.*

however, rise from an average of 1.11x to an average of 1.20x; clearly this is an effective technique.

The future work section of this chapter suggests a way of potentially improving this further. This is quite a dissatisfying result though, *MapISE* achieves better results by occasionally opting out from taking any action. AISE should to produce instructions which provide acceleration, therefore in section 5.1.3 the cost function described in section 4.5 is added to *ISEGen* to ensure it will provide extension instructions that *MapISE* can always use.



(a) The speed-ups obtained when using the default register allocation settings (left-hand bar) and when a single static allocation is used for every extension instruction (right-hand bar). *Note: the full version of this chart is figure B.8 on page 170.*



(b) This chart compares the speed-ups possible when using the permutation units. The default setting is to use one-step permutations (left bar), this is compared with never using the permutation unit (middle bar) or using two-step permutations (right-hand bar). *Note: the full version of this chart is figure B.9 on page 172.*

Figure 4.10: *A comparison of different register allocation modes.*

4.7.4 Register Allocation Variations

Section 4.3 did not describe a "No register allocation" technique because this is not an actual technique. Registers still have to be allocated to registers, but in this mode there is no decision process involved. For each and every extension instruction that is mapped the first input vector is assigned to `vr05`, the second input (if it is used) to `vr04`, the third input (if it is used) to `vr03`, the first output is assigned to `vr01` and the second output (if it is used) is assigned to `vr02`. This

also means that the permutation units are never used (they are mapped to `vr08`–`vr63`).

The results shown in figure 4.10(a) are surprising. They show that the "No register allocation" mode performs almost as well as the default mode. The default register allocator results in an average speed-up of 1.110x, the "No register allocator" results in an average speed-up of 1.105x. The does not mean that the default register allocator is bad, but rather than there is no such thing as a "good" register allocation when targeting the *EnCore* vector registers. There is, however, such a thing as a "bad" register allocation though, see section 4.9.4.1. The static assignment of the "No register allocator" happens to avoid one of the biggest performance killers with *EnCore* vector registers: outputs over-writing inputs.

Class	**Type**	**Count**	**Total%**
Local Scalar	Scalar to Vector	4860	19.6%
External Scalar	External Scalar to Vector	4352	17.5%
	Constant to Vector	8681	35.0%
Local Vector	Vector to Vector	2160	8.72%
External Vector	External Vector to Vector	1311	5.29%
	Last-loop Vector to Vector	3405	13.7%

Table 4.2: *The different sources of inputs to extension instructions summed across all 179 benchmarks.*

The *EnCore* vector register were designed based on the idea that the output of one instruction would feed into the next. This was the purpose of the permutation units, to rearrange the order of a vector to cover cases where the output order of one instruction did not match the input of the next. In reality, however, vector-to-vector dataflow is relatively rare. In table 4.2 there are 24,769 inputs recorded and only 27.8% of them come from vector registers. The main problem is the "External Scalars". If a given basic block with an extension instruction is inside a loop and that extension instruction takes some "external scalars" as input via `vr01` then these values generally stay constant for every iteration of the loop. They are loop-hoisted invariants. Thus if `vr01` gets overwritten the "external scalars" will need to be copied back into `vr01` in every loop iteration. As "external scalars" account for 52.6% of all vector input variables and each vector contains up to four variables this problem is extremely common. Thus the static assignment approach taken by "No register allocator" performs just as well as an allocator which frequently ends up picking a similar scheme.

Figure 4.10(b) shows the performance of the default register allocator with one-step permutations, no permutations and two-step permutations. Not using permutations results in a significant drop in performance, the average speed-up falls from 1.11x to 0.99x. This is actually worse than "No register allocation" even though it does not use permutations either. This

is because the heuristics in the default mapper rely on permutations to try and avoid overwriting values. The performance drop from one-step to two-step permutations is not as significant, 1.11x to 1.10x, thus two-step permutations do not quite provide enough of a benefit to make up for only have five vector registers available.

4.7.5 Commutativity Variations

Figure 4.11 shows the results of trying to exploit commutativity in the graph-subgraph isomorphism module of *MapISE*. It shows full commutativity (floating point and integer) versus no commutativity. The IEEE754 floating point standard guarantees that switching the arguments to naturally commutative operators does not affect the result.

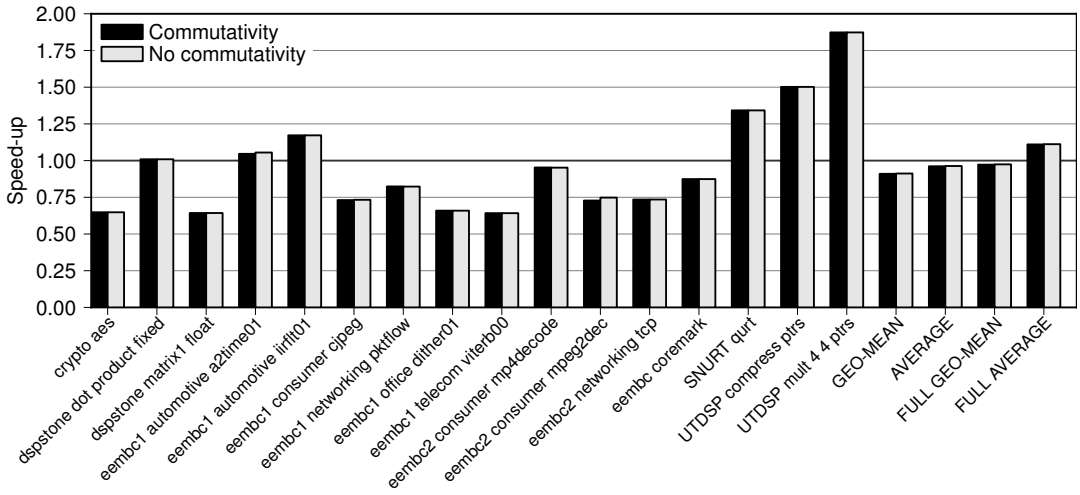
It can be seen in figure 4.11(a) that commutativity makes no significant difference to the speed-ups obtained. Figure 4.11(b) shows that commutativity does allow a few additional extension instructions to be mapped. The average number of mappings per benchmark go from 27.77 to 28.02 for no commutativity to full commutativity. The additional mappings actually fractionally slow-down the code, speed-ups go from 1.112x to 1.110x. This is indistinguishable from noise.

Note that the flat results are not due to commutativity checking being broken. During the development of the PASTA project *ISEGen* had a key bug where, under certain circumstance, it would swap a nodes edges in the extension instruction. At this point commutativity support was essential for using key extension instructions. Once the bug was discovered and fixed, however, little benefit remained.

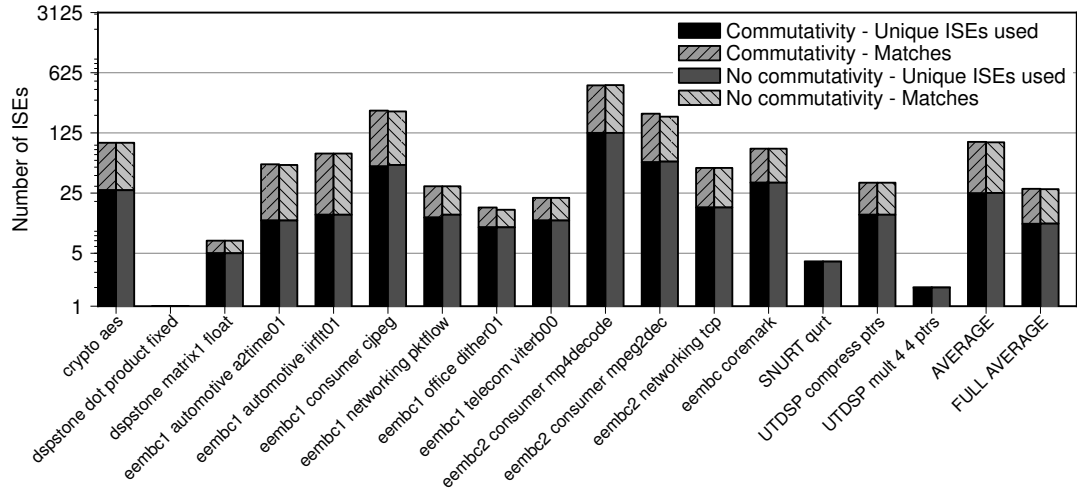
The fact that no commutativity is almost identical to have full commutativity is actually a useful result. In the *MapISE* there are approximately 500-600 lines of code spread through several sections of the code to support commutativity. Requiring code in several separate key areas meant that during development of *MapISE*, bugs related to commutativity took a significant amount of the overall debug time (passed only by issues related to *GCC's* virtual uses and definitions for representation pointer aliases and *ISEGen's* or issues related to *ISEGen* stripping casts from extension instructions). Therefore, the interesting result is that implementing commutativity checking in a graph-subgraph isomorphism based instruction mapper is not worth the effort.

4.8 Results - Retargeting Extension Instructions

This section evaluates *MapISE's* ability to take extension instructions generated for one benchmark and map them to a related benchmark.

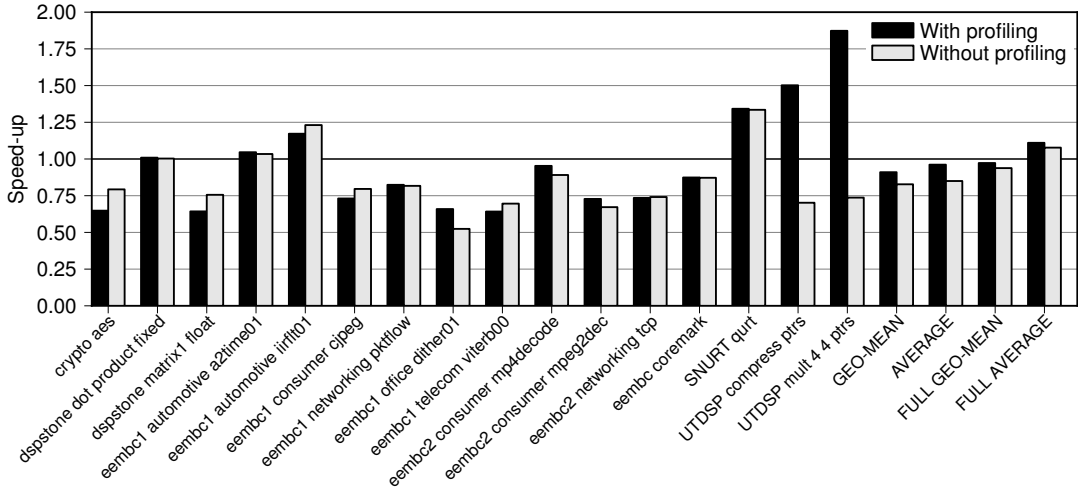


(a) This chart shows the speed-ups obtained when exploiting commutativity in *MapISE*. The left bar allows commutativity in both integer and floating point nodes for every arithmetic operation that is naturally commutative. The right bar does not consider commutativity at all. *Note: the full version of this chart is figure B.10 on page 174.*

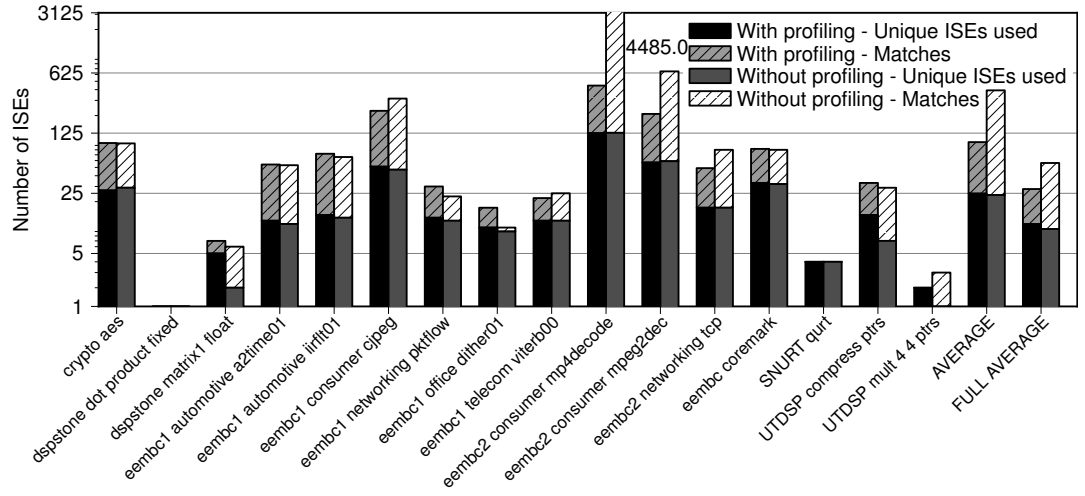


(b) This chart is similar to figure 4.7(b) except that every stacked bar relates to the number of extension instructions that *MapISE* can use. The lower bars are the number of unique extension instructions used, the upper bars are the total number of sites that extension instructions could be used. *Note: the full version of this chart is figure B.11 on page 176.*

Figure 4.11: *A comparison of different commutativity options.*



(a) Speed-ups obtained from the default (left-hand bar) and the alternative modes (right-hand bar). *Note: the full version of this chart is figure B.12 on page 178.*



(b) Mapping quality information for default (left-hand bar) and alternative modes (right-hand bar). *Note: the full version of this chart is figure B.13 on page 180.*

Figure 4.12: *An evaluation of* MapISE *when the compiler mode changes between extension instruction generation and exploitation. Withholding profiling data affects the decisions made by* GCC*'s optimiser.*

4.8.1 Compiler Differences

Figure 4.12 generates instructions using a profile driven *ISEGen*, but then attempts to use them on the same benchmarks without profiling data. This is not necessarily a common problem, but as *GCC's* optimisation choices change significantly when the profiling data is withheld this simulates using a different version of the same compiler. After a major upgrade a compiler could make different optimisation decisions, so it is necessary to check that *MapISE* would remain usable.

Figure 4.12(a) shows the speed-ups retained. The results are quite encouraging, there is a small drop in performance, from an average speed-up of 1.11x to 1.08x, but most of the performance is retained. Figure 4.12(b) shows that more instructions are mapped without profiling data. This occurs because if a large instruction can no longer be mapped, perhaps a hot loop is no longer unrolled, then two smaller matches may be mapped instead. This can mean more instructions are mapped but results in lower speed-up overall.

4.8.2 Modifying Programs

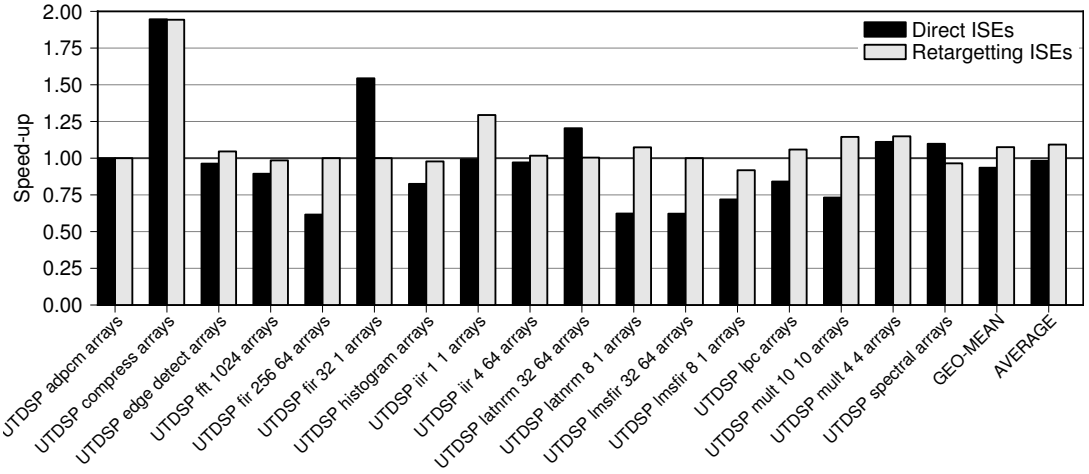
It is also necessary to check that when a program is changed, will the new version still be able to use the extension instructions generated for the old version. To simulate going from "v1.0" to "v1.1" of various programs, the UTDSP suite will be used as it has several versions of each benchmark. Figure 4.13 shows results for extension instructions generated for *pointer* versions of each benchmark and then mapped them to the *arrays* versions. Figure 4.14 does the reverse. Figure 4.15 generates extension instructions for the *arrays* version of each benchmark, and then tries to map them to a version where the loops have been *software pipelined*.

Counter-intuitively the results in figure 4.13(a) are actually faster when using extension instructions designed for a different benchmark, than when using ones designed for them directly. The speed-up increases from 0.98x to 1.09x. The results in figures 4.14(a) and 4.15(a) only small see improvements, 1.099x to 1.100x and 1.01x to 1.02x respectively, but for these experiments a small improvement is an excellent result: it indicates that *MapISE* is able to handle programs being updated.

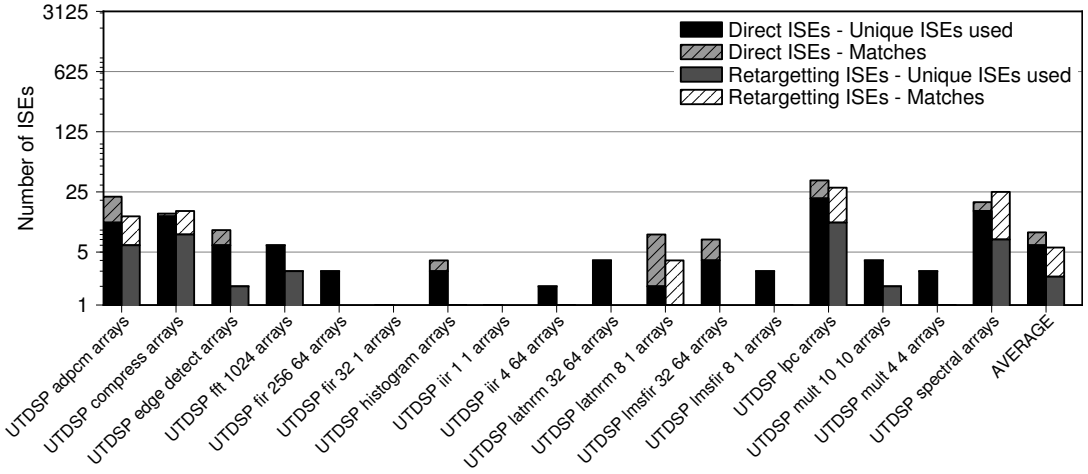
4.8.3 Using Different Implementations

The experiment used to generate figure 4.16 is a more complex than the other retargeting experiments. Extension instructions are generated for one benchmark and then used with other related benchmarks, see table 4.3.

These results in figure 4.16(a) show that large changes in the program between generation and exploitations (e.g. MPEG-2 to MPEG-4, or using a different FFT implementation) is rarely very effective. Although the average speed-up only drops little, from 1.02x to 0.99x, this is



(a) Speed-ups.



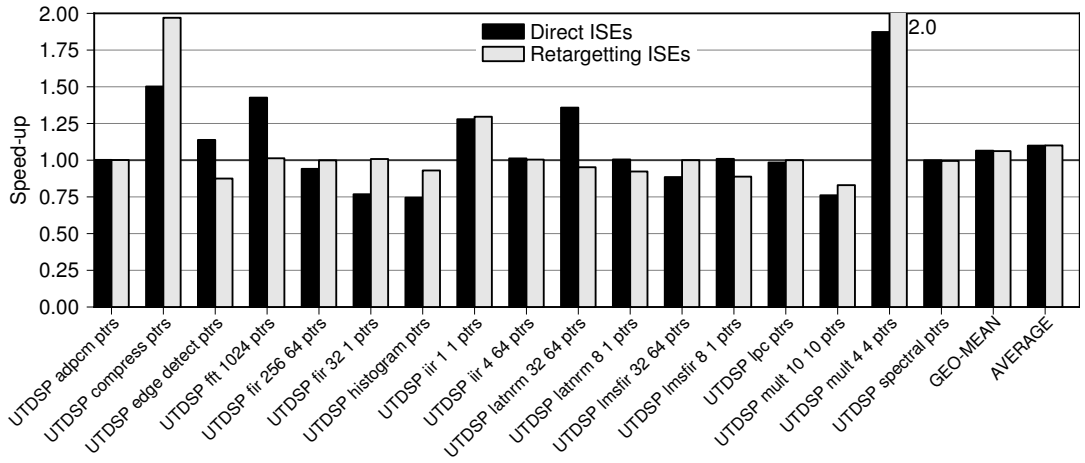
(b) Mapping quality information.

Figure 4.13: *Extension instructions are generated for* ptrs *benchmarks and then exploited on* arrays *benchmarks.*

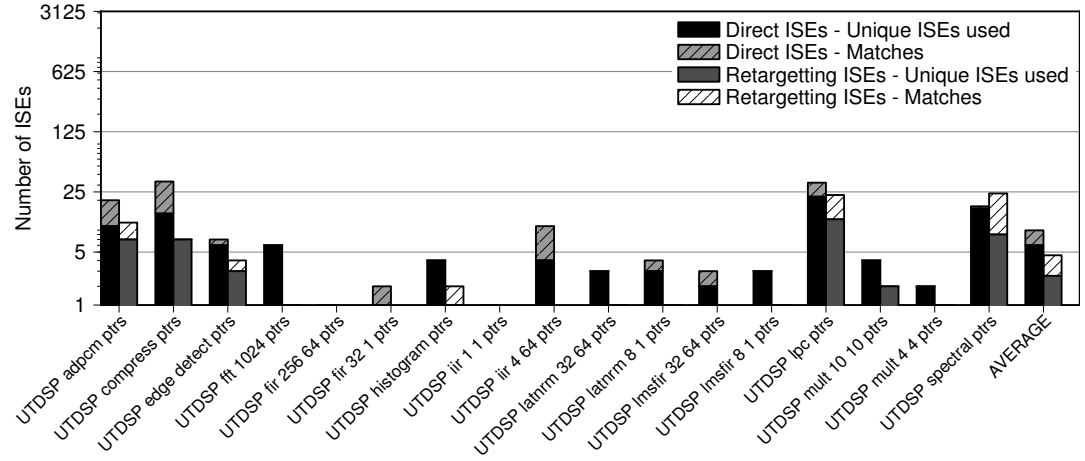
mostly because every benchmark moved closer to a 1.0x speed-up, in some cases that was an improvement, in other it was not. Figure 4.16(b) shows that far fewer unique extension instructions are used.

4.8.4 Combining Programs

Figure 4.17 shows the results of a small domain-based AISE experiment. Extension instructions were generated for five telecommunications benchmarks. The extension instructions for each were then merged into a single extension unit which was then used to accelerate the same five benchmarks. This had very little effect of performance, the average speed-up increases



(a) Speed-ups.



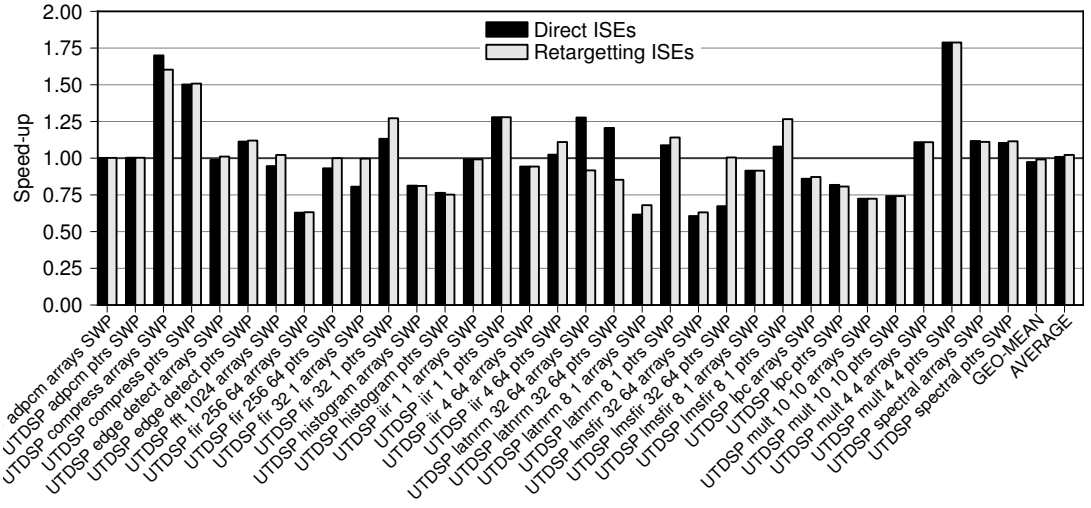
(b) Mapping quality information.

Figure 4.14: *Extension instructions are generated for* arrays *benchmarks and then exploited on* ptrs *benchmarks.*

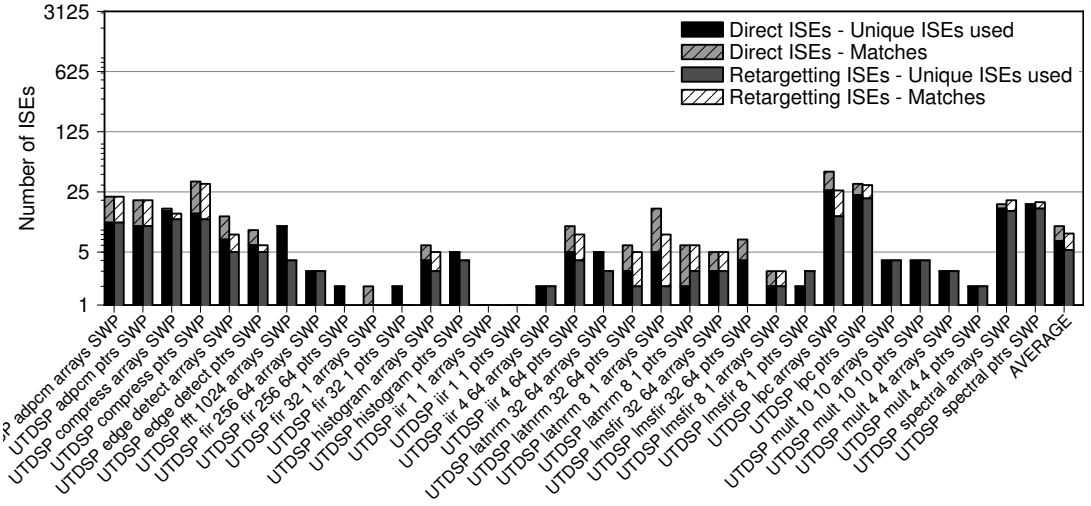
from 0.72x to 0.75x. For the other cases of retargeting extension instructions, small slow-downs are acceptable. As this experiment is purely additive, each benchmark has access to its own extension instructions and more, a larger increase in speed-up may have been expected.

4.9 Critical Evaluation

There are a number of distinct areas that need to be critically evaluated to explain the above results. This is undertaken here, along with some additional experiments to help demonstrate key points.



(a) Speed-ups.



(b) Mapping quality information.

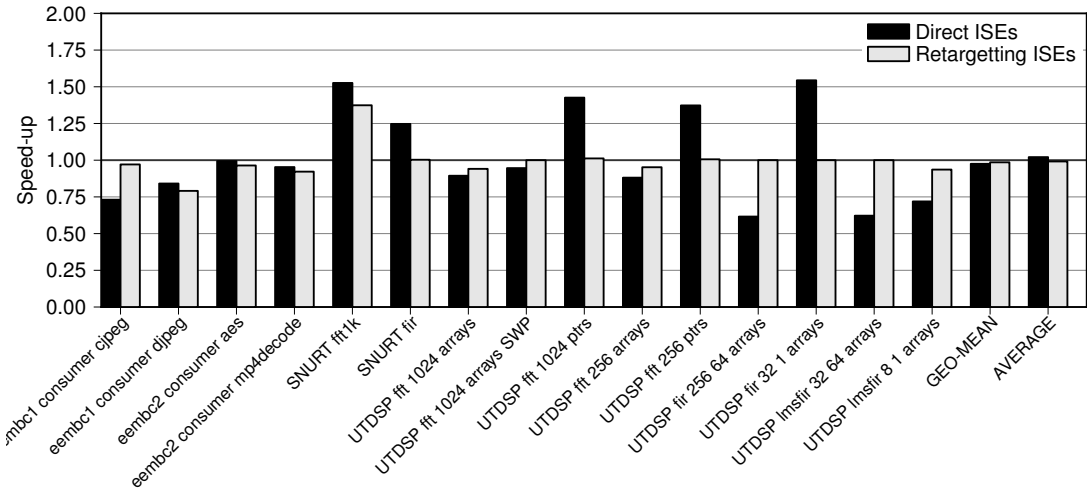
Figure 4.15: *Extension instructions are generated for* arrays *benchmarks and then exploited on* arrays-SWP *benchmarks.*

4.9.1 *ISEGen* Issues

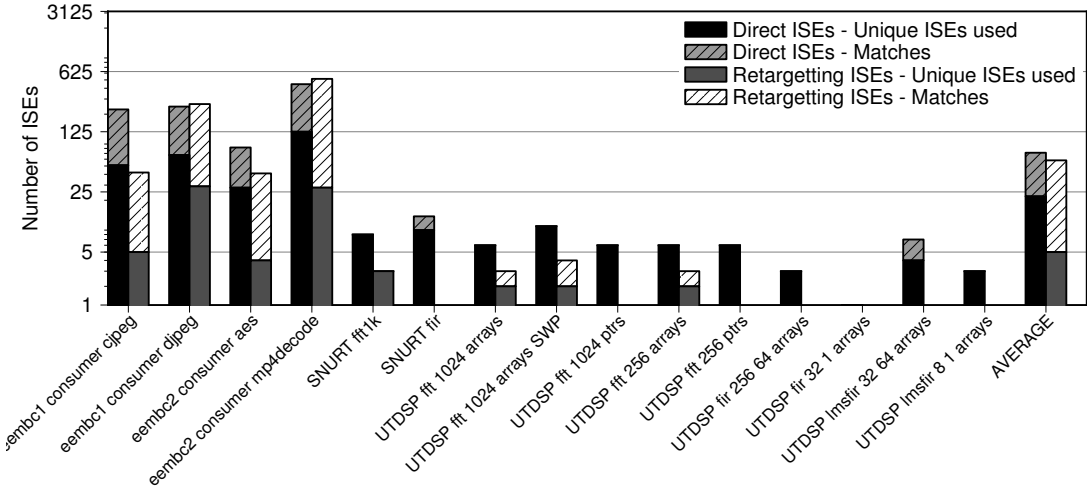
Figure 4.18 compares *ISEGen's* predicted performance with what *MapISE* was able to achieve. It can be seen that *MapISE* falls far beneath *ISEGen's* expectations, *ISEGen* estimated a speed-up of 1.54x, *MapISE* achieved 1.11x.

The reason for this are three-fold: firstly, *ISEGen* over-estimates the benefits of its extension instructions. Secondly, many of the extension instructions produced are actually not usable with the code they were generated for. Thirdly, *MapISE* uses a greedy algorithm for exploiting multiple extensions per basic block which is certainly sub-optimal (see section 4.9.3).

The most obvious issue with *ISEGen's* performance estimates are the way it calculates the



(a) Speed-ups.



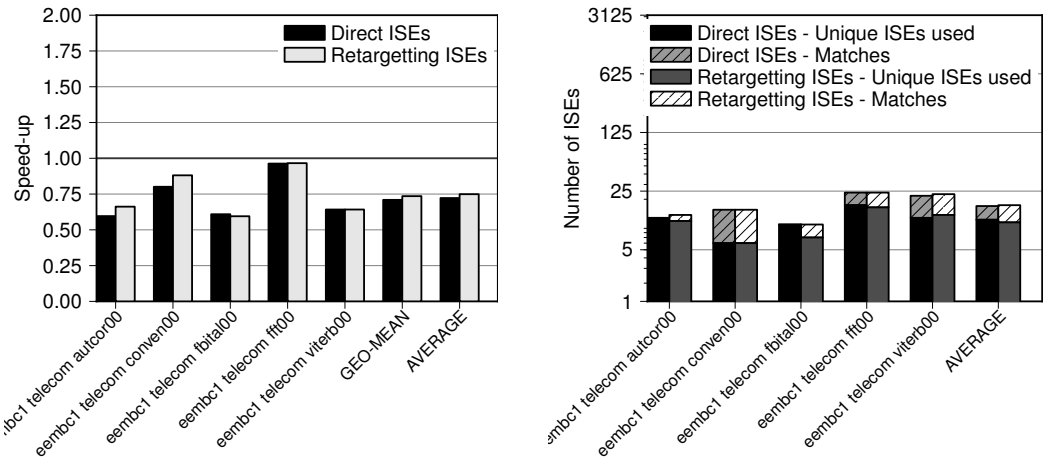
(b) Mapping quality information.

Figure 4.16: *Extension instructions are generated for one benchmark (see table 4.3) and then exploited on one or more related benchmarks.*

baseline number of cycles that it is apparently accelerating – it massively under-estimates how long programs will take to run. It is normally off by at-least a factor of 2x: e.g. for SNURT *jfdctint* it estimates 3485 cycles when the actual number of inside the performance counters is 7667; for UTDSP *histogram* it estimates 288,769 cycles when the actual number is 1,161,801. Mostly this is because it does not account for library code, the number of cycles that a function call takes, nor that dependent code and branches can result in pipeline bubbles – it purely considers the summed cycles of the dataflow graphs for every basic block. The inaccuracies of these numbers do not affect the results presented in this thesis at all. *ISEGen* picks extension instructions based on their benefit in dataflow graphs, the fact that is inflating the importance

Generating Benchmark	Exploiting Benchmark(s)
EEMBC2 *mpeg2dec*	EEMBC2 *mp4decode*
Crypto *aes*	EEMBC2 *aes*
EEMBC1 *cjpeg*	EEMBC1 *djpeg*
SNURT *jfdctint*	EEMBC1 *cjpeg*
DSPstone *fir float*	SNURT *fir*, UTDSP *fir 32-1 arrays*, UTDSP *fir 256-64 arrays*, UTDSP *lmsfir 32-64 arrays*, UTDSP *lmsfir 8-1 arrays*
SNURT *fft1*	SNURT *fft1k*, UTDSP *fft 1024 arrays*, UTDSP *fft 1024 arrays SWP*, UTDSP *fft 1024 ptrs*, UTDSP *fft 256 arrays*, UTDSP *fft 256 ptrs*

Table 4.3: *The benchmarks used to generate and exploit extension instructions in figure 4.16.*



(a) Speed-ups.

(b) Mapping quality information.

Figure 4.17: *Extension instructions are generated for each benchmark and then combined to create a large set of instructions,* MapISE *then maps instructions from the entire set to each benchmark.*

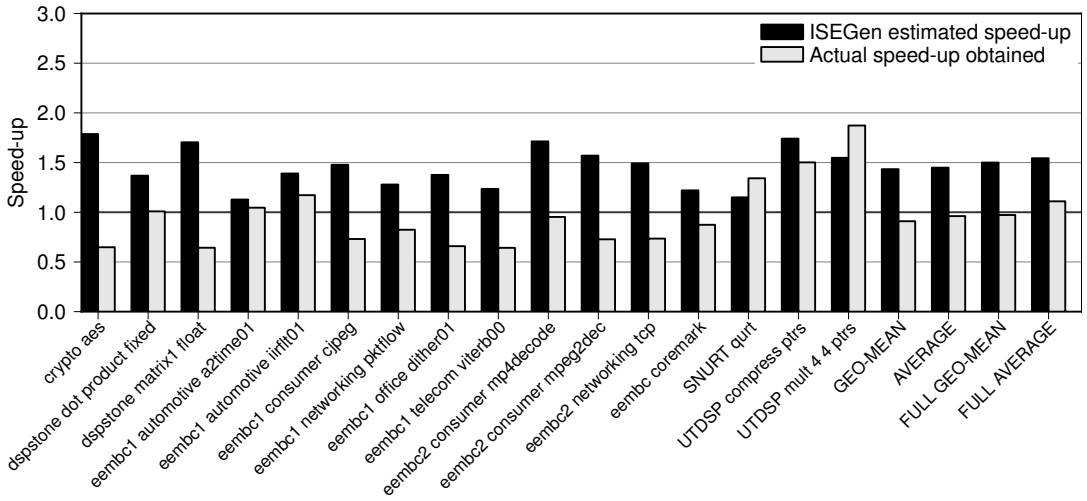


Figure 4.18: *A comparison of* ISEGen's *estimated benefit from extension instructions and the benefit actually achieved by* MapISE. Note: the full version of this chart is figure B.14 on page 182.

of those graphs does not change the instructions that are generated. It is just hard to compete with *ISEGen's* "estimated" speed-up because if it halves the number of cycles in the program it doubles the benefit from extension instructions and thus doubles the speed-up it estimates. *ISEGen's* estimates for the number of cycles that each instruction extension will save are not nearly as inaccurate as they are based on a single dataflow graph each, so of the above issues only pipelining inaccuracies apply.

Further regarding *ISEGen's* performance estimation model: it fails to take account of the compiler in several ways. The most significant are register allocation issues, these are discussed in section 4.9.4. The other major flaw in *ISEGen* is that it assumes that the back-end will have little effect on the produced code. The *GCC* back-end performs further strength reduction over that performed in the middle-end. This means that *ISEGen* thinks that it is implementing an expensive operation (e.g. divide) in parallel with other operations and that the extension instruction will provide a large benefit. In reality, however, the divide may be strength reduced to a shift, or a short sequence of custom arithmetic and the true savings provided by the extension instruction are negligible.

A more subtle back-end issue is that *ISEGen* assumes that every operation implemented in an extension instruction means that the time it would have taken to run that operation on the baseline core has now been saved. In reality that operation may have simply been sitting in a gap in the schedule, and now that the extension instruction is implementing it there is just a pipeline bubble instead. No cycle(s) have been saved. This is an extremely difficult problem for the AISE tool to deal with, it may just not be possible for a high-level tool like *ISEGen*.

It is an important issue though, because extension instructions absorb a large amount of data-parallelism the scheduler may have little left to work with. So in a basic block an extension instruction performs most of the arithmetic at the start, but then a series of dependent operations have to execute – so the total time to execute the block is only a little lower than before, despite the "savings".

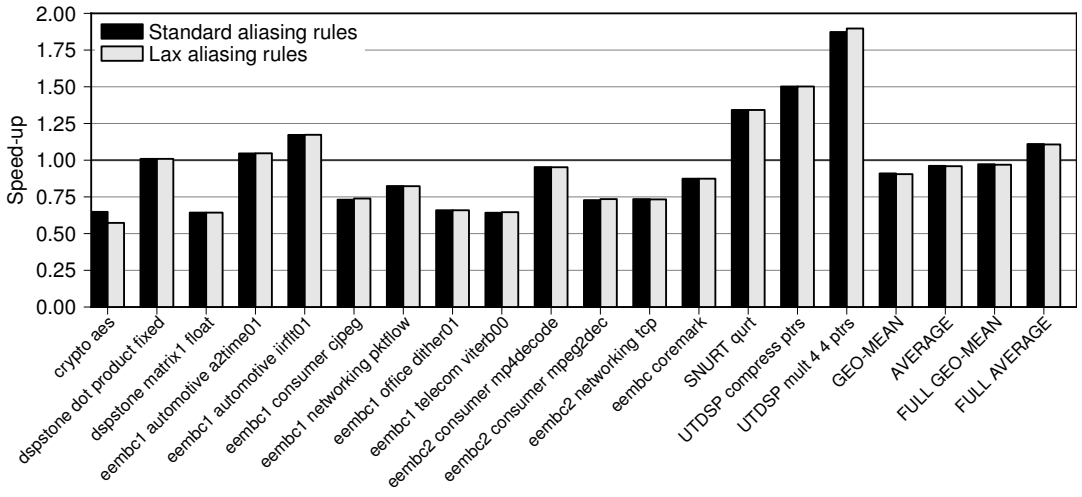
It should be noted that although these issues are being presented in terms of the *ISEGen* tool, any implementation of the ISEGEN algorithm would likely experience the same problems. In fact, any implementation of any AISE algorithm may be susceptible.

If figure 4.7(b) is reconsidered it can be seen that *MapISE* only manages to use 60% of the unique extension instructions that *ISEGen* produces. Partially this will be due to *MapISE's* greedy approach, but a significant portion is due to *ISEGen* not using virtual dependencies (section 4.9.2 unsuccessfully attempts find out what portion). This means that *ISEGen* often thinks sections of code are independent when they are, in fact, dependent. This causes it to create an extension instruction which implements two sections of dependent dataflow in parallel, and *MapISE* is unable to use the instruction. The XML CDFG format used by *ISEGen* does not have any way of describing virtual dependencies. For *MapISE*, the CDFG format was extended to represent virtual dependencies. This could also be added to the XML as well, but it is not clear if the ISEGEN algorithm could be extended to support virtual dependencies. The ISEGEN paper [Biswas et al., 2006a] does not mention this issue, so it may not be trivial to implement a solution.

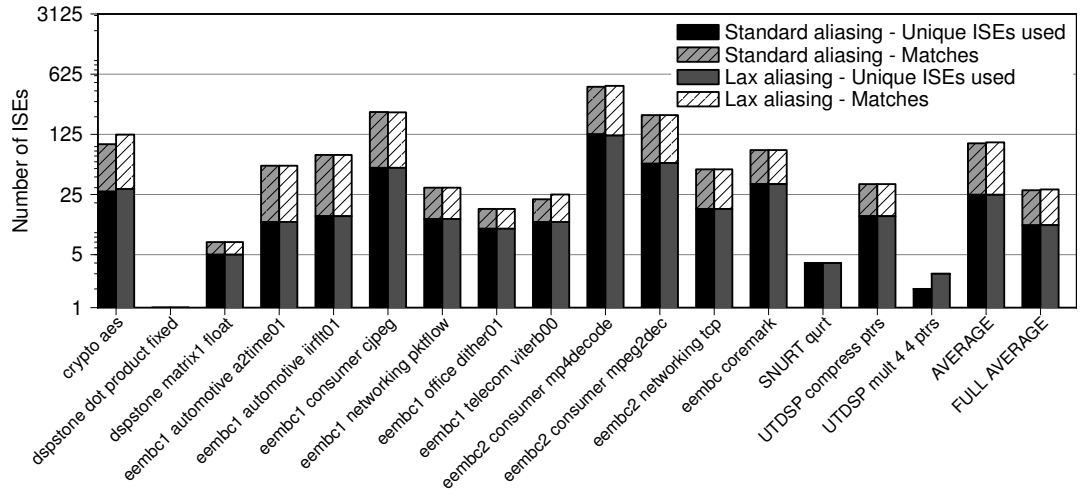
4.9.2 Effect of Aliasing Differences on Performance

This section tries to determine how large an affect *ISEGen's* lack of support for virtual dependencies has on *MapISE*. By considering figure 4.7(b) it is clear that the upper bound on the effect is 40% of the unique extension instructions that *ISEGen* produces (*MapISE* can use the other 60%). The lower bound is, of course, 0%. To attempt to find the true bounds *GCC* is encouraged to discard aliasing information.

This is achieved by using the *GCC* `-fargument-noalias-anything` command switch. This switch is not meant to be used directly as it is unsafe, the front-end developer should set this if it is appropriate for the language that the front-end accepts. Setting this switch specifies that for each pointer argument given to a function no other pointer in that function points to the same storage location, i.e. no other pointer argument or global variable points to the same block. This does not mean, however, that no aliasing information is produced, it is still calculated for local variables within a function and for the use of global variables unrelated to the function arguments. Despite this the switch is still unsafe as it does not conform to the expected C semantics. It used here purely for evaluation purposes, it is not proposed as a solution to any problem. Only one benchmark was miscompiled due the use of the switch: EEMBC *coremark*,



(a) Speed-ups obtained with standard C aliasing rules (left-hand bars) or less strict rules (right-hand bars). *Note: the full version of this chart is figure B.15 on page 184.*



(b) Mapping quality information. *Note: the full version of this chart is figure B.16 on page 186.*

Figure 4.19: *An investigation of the effect of weakening the aliasing rules in* GCC *with regards to* MapISE*.*

the results from the standard run for that benchmark were inserted into the data-set to avoid affecting the geometric mean or the average.

The reason that only one benchmark was miscompiled was likely that only a small amount of the total aliasing information is discarded as a result of using this switch. This can be seen indirectly in figure 4.19(b). On average standard aliasing rules allow 10.98 extension instructions to be used per benchmark, this only increases to an average of 11.01 extension instructions per benchmark with the `-fargument-noalias-anything` switch. This is also counter balanced by the total number of extension instruction sites, which is identical for the

two options. For benchmarks where lax aliasing rules allow more extension instructions to be used the affect on performance is mixed, e.g. Crypto *aes* slows down but UTDSP *mult-4_4_ptrs* speeds up. Overall the average speed-up decreases from 1.110x to 1.107x, another case showing that more extension instructions can sometimes hinder performance.

This either means that *ISEGen's* lack of support for virtual dependencies has no effect on *MapISE*, or *GCC* is unable to discard enough aliasing information to be helpful. Manual verification showed that *ISEGen* definitely produced extension instructions in shapes that *MapISE* would never be able to use, and that *GCC* still retained many virtual dependencies when apparently using reduced aliasing information. Manual verification, however, is unable to give an idea of how much of the 40% of unused instructions relate to this issue.

It is not possible to just ignore aliasing information in the *MapISE* alone. This was attempted by generating CDFG without virtual dependency annotations. The produced code was invalid and *GCC's* SSA verification passes would not accept the code (it was impossible to schedule it, extension instruction had to be place both before and after certain points). This issue of virtual dependencies is a good demonstration, however, of how *MapISE* is always "retargeting" extension instructions. The good side-effect of this is that *ISEGen* bug do not cause wrong-code, the bad side-effect is that *ISEGen* bugs can be hidden from view.

4.9.3 Matching Issues

There is a large deficiency in *MapISE's* approach to mapping multiple extension instruction in a block. It uses a greedy approach which will take an instruction which saves five cycles, even if that blocks two places where four cycles could be saved. In the area of tree-tiling this is a solved problem, but as matching tree-shaped instructions in DAGs is NP-COMPLETE [Proebsting, 1998], and graph-subgraph isomorphism is NP-COMPLETE [of Theorem-Proving Procedures, 1971] combining the two, even heuristically, is likely to have a huge effect on run-time.

4.9.4 Register Allocation Issues

4.9.4.1 Aggressive Register Allocation

Figure 4.20 shows the performance of the default register allocator without the load-balancing cost described in section 4.3. This was once the default mode of operation, the aim was to allow as many vector-to-vector moves as possible and to fully exploit the permutation units. As has already been discussed in section 4.7.4 this causes other problems. Focusing on vector-to-vector moves is wasted effort when scalar-to-vector moves dominate, doing so dropped the average speed-up from 1.11x to 0.96x. Figure 4.21 confirms that this holds for alternative permutation configurations as well.

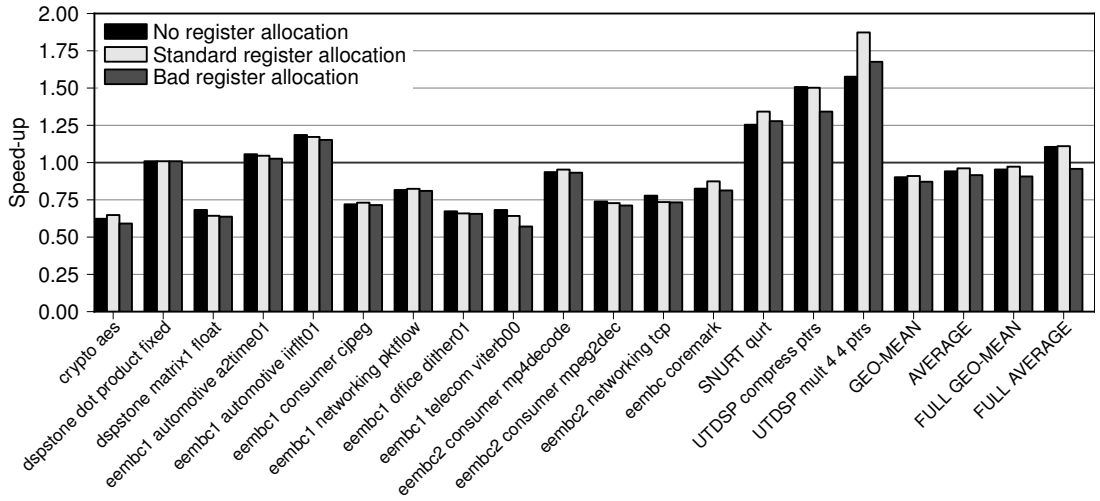


Figure 4.20: *The speed-ups obtained when using deliberately bad register allocation heuristics.* Note: the full version of this chart is figure B.17 on page 188.

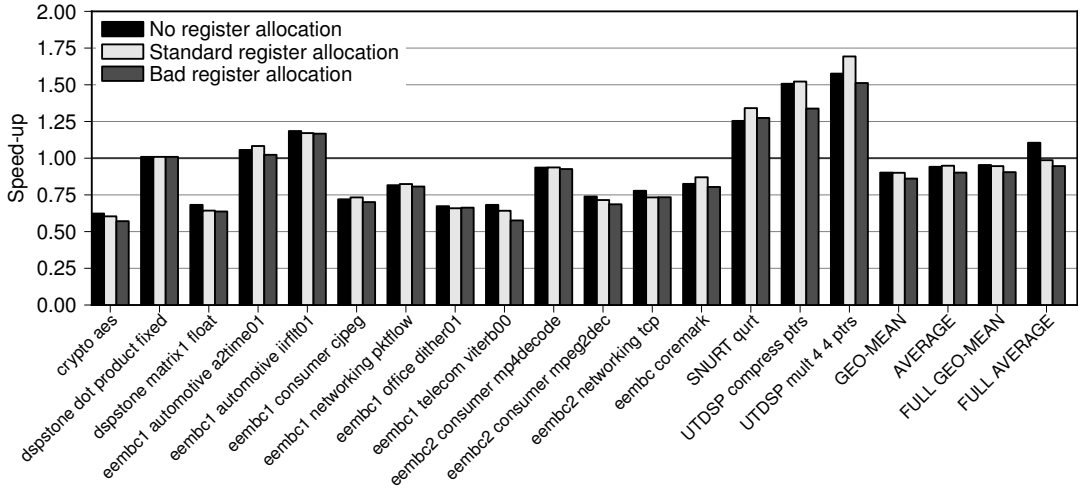
4.9.4.2 Caller or Callee Saved Extension Registers

Figure 4.22 compares caller-saved extension registers (the default) with callee-saved registers. Using callee-saved registers causes performance to fall from an average speed-up of 1.11x to 0.91x. The experiments are included here because it seems surprising that it would have such a large effect on performance (both options were evaluated before picking caller-saved as the default), and because the *EnCore* hardware engineers believe that callee-saved should be the default.

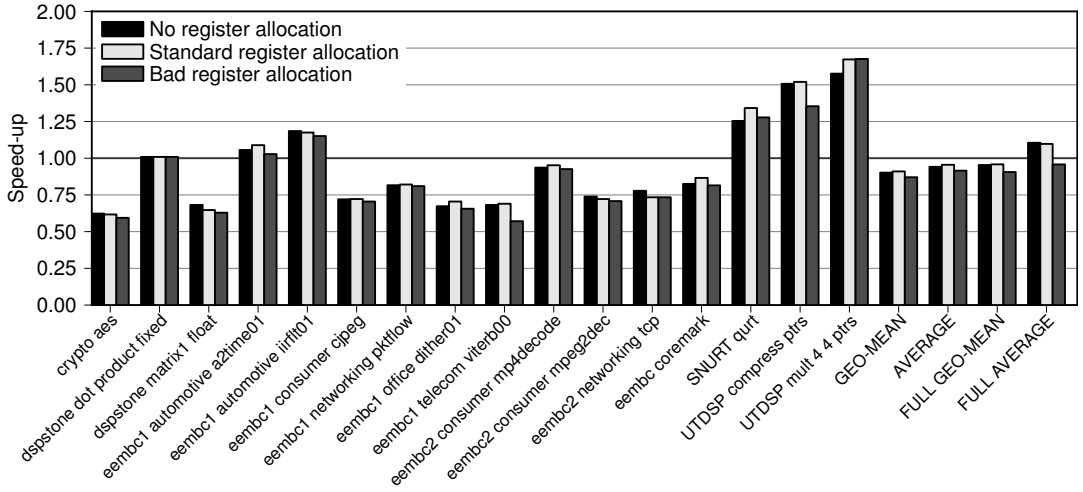
4.9.4.3 Summary of Register Allocation Issues

Section 4.7.3 demonstrated that eliminating extension instructions with high register overhead improved performance. Section 4.7.4 described how care should be taken to avoid registers colliding and that permutations are key when collisions do happen. This section demonstrated that trying to aggressively use the vector registers and permutations units actually hinders performance.

The trend is that vector registers can destroy performance, but never help it. Chapter 5 looks at how to better manage vector registers and proposes an alternative register scheme.



(a) No permutations used. *Note: the full version of this chart is figure B.18 on page 190.*



(b) Two-step permutations used. *Note: the full version of this chart is figure B.19 on page 192.*

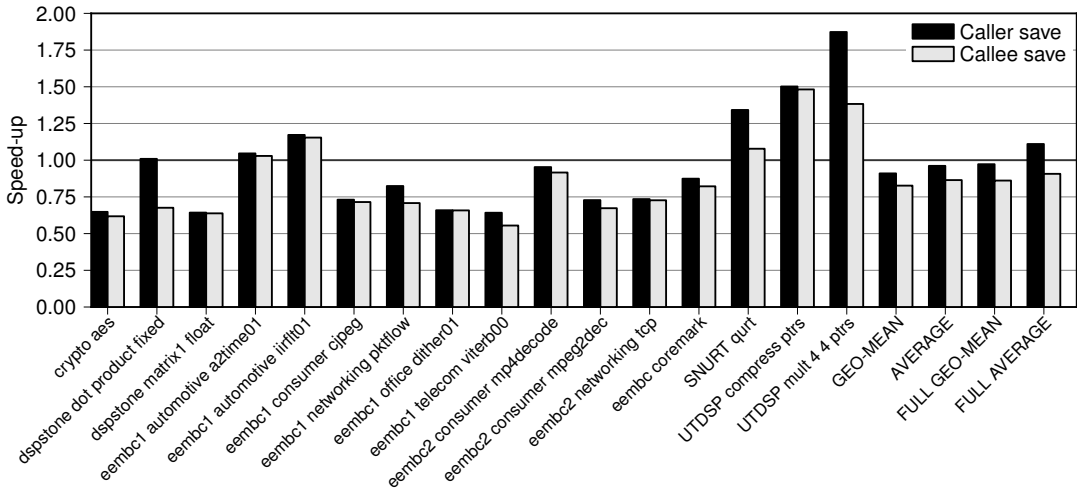
Figure 4.21: *The speed-ups obtained when using deliberately bad register allocation heuristics with permutation variations.*

4.10 Summary and Conclusions

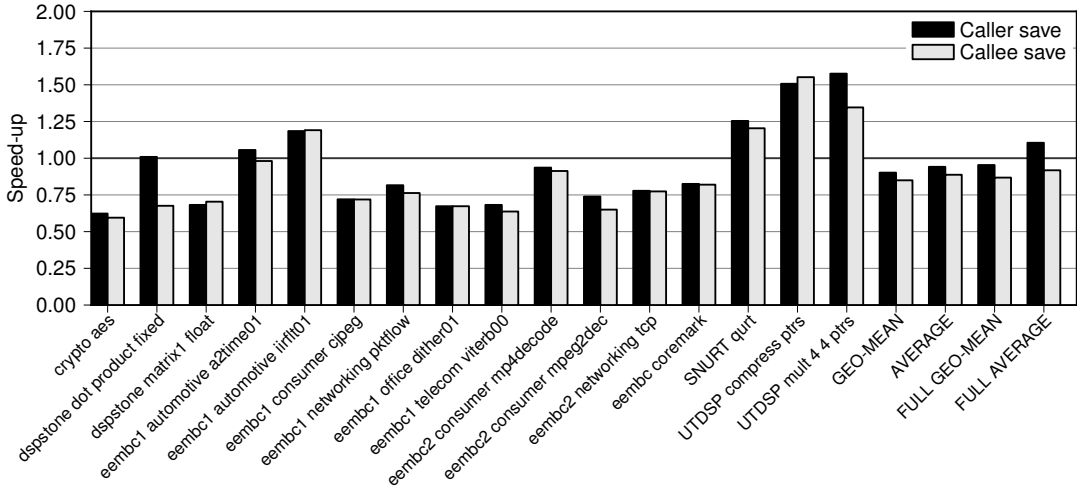
4.10.1 Future Work

Firstly, much of what could be considered future work is covered in chapter 5. The work in that chapter regards modifications to the AISE heuristics or modifications to hardware. The only *MapISE* changes in chapter 5 are those required to match hardware changes. Thus, any enhancements to *MapISE* that have not already been proposed are considered the future work of this chapter.

There exists graph-subgraph isomorphism checkers that can run in polynomial time [Jiang



(a) Default register allocator. *Note: the full version of this chart is figure B.20 on page 194.*



(b) No register allocator. *Note: the full version of this chart is figure B.21 on page 196.*

Figure 4.22: *An evaluation of callee-saved registers against caller-save.*

and Bunke, 1998] by constraining the problem slightly: edges must be ordered. *MapISE* actually needs to do its own edge-order checking so this would not be a large issue. The only limitation is that if *MapISE* is not doing its own edge-order it cannot handle commutativity. As section 4.7.5 found, however, commutativity provides no benefits, so losing the ability to support it is completely acceptable.

Partial mapping of instructions could be performed by using arithmetic identities. Parts of instructions which are not required could be transformed into null operators by applying arithmetic identities to them. For example an addition node could be nullified by mapping it to $X + 0$, or a multiple to $X * 1$, both nodes would be passing the value X onto the rest of the instruction. This may improve the usability of extension instructions as if a match was off by

only one or two nodes then they could be disabled, thus allowing the instruction to be used.

The technique described in section 4.5, which skips extension instructions which will likely slow-down performance, currently works statically at the beginning of *MapISE*. If it was dynamic and operated after the vector register allocator it could make accurate decisions, instead of estimates. As has already been mentioned, however, this whole technique is a work-around for a bigger problem, so this may not be worth pursuing.

4.10.2 Summary

This chapter has described a tool for mapping the complex instructions generated by AISE. An extensive evaluation was performed that demonstrated that the tool could achieve an average speed-up of 1.11x across 179 benchmarks. A significant understanding of underlying performance issues was also obtained which was used to design the set of experiments found in the next chapter.

Chapter 5

Instruction Set Extension and Code Generation

"Nothing in progression can rest on its original plan. We may as well think of rocking a grown man in the cradle of an infant."
— Edmund Burke, *member of the House of Commons for Wendover*, 1729–1797.

This chapter investigates methods by which AISE can generate extension instructions that the compiler is better able to exploit. Modification of *ISEGen's* parameters and heuristics is investigated, and then hardware changes are proposed and assessed.

Once it became clear that changes to the hardware were necessary it also became clear that many different changes had the potential to create extension instructions better suited for compilation. In fact, every variation tried provided some benefit. This meant, however, that design space exploration was happening, but without the benefit of full automation.

Section 5.1 describes several ways that *ISEGen* could create extension instructions which do not such put a high strain on *MapISE's* register allocator. Next, in an attempt to increase memory bandwidth and to take advantage of the existing vector hardware, section 5.2 proposes adding vector loads and stores. Section 5.3 proposes eliminating the vector registers. Sections 5.4–5.6 evaluate the proposals and finally section 5.8 critically evaluates the work of the chapter.

5.1 Reducing Register Pressure

A number of changes were made to *ISEGen's* parameters and one new value was introduced to its heuristic vector.

5.1.1 Reducing the Number of I/O Ports

The first change that was made was a simple one. As *EnCore* only has seven vector registers and a single extension instruction may use up to five vector registers at once there is not much that the register allocator can do to avoid collisions. It is hypothesised that if each extension instruction is limited in the number of vectors it uses then performance may improve due to better register allocation.

5.1.2 Hard-Wiring Constant Values

The second change requires changing the *ISEGen* mode of operation slightly. Table 4.2 on page 55 shows that 35% of the inputs to extension registers are constants. It is not only *MapISE* that can tell this, these nodes are marked as being constants in the XML CDFG that *ISEGen* receives. *ISEGen* was therefore modified so as to not consider these constants as general inputs but to instead hard-wire them inside the extension instructions.

This required a minor update to *MapISE* so that it only maps constants in instructions to constant nodes with the same value, but other than that a constant node is just a node like any other.

5.1.3 Modifying the ISEGEN Heuristic Parameters

The most significant change made to *ISEGen* was the addition of a new parameter. In its default mode of operation *ISEGen* assumes that there are no costs associated with loading an input value. Other operations have a cost, e.g. an addition might have a cost of one cycle. Therefore, the performance model was updated to assign a cost to each input. This is provided as a parameter and maybe any value, including fractions. The rest of the performance model builds costs based on cycles, so values should be picked accordingly (i.e. quite small).

The expected effect of this new addition to the performance model is smaller extension instructions. *ISEGen* should only include additional operations in an extension instruction if it will save enough cycles to offset the cost of loading the value. This heuristic is very similar to the “Eliminating Poor Mappings” method described in section 4.5. Where that method avoided using extension instructions with an expected register load cost that was higher than their benefit, this method avoids creating them.

5.2 Wide Memory Bus for Wide Registers

The three changes proposed above all work by reducing the amount of data that is sent to extension instructions (and thus easing the task of register allocation). This section takes an alternative approach which may make it easier to send data to extension instructions.

The extended *EnCore* processor already has vector registers, so adding vector loads and stores is a good conceptual fit. As a certain amount of the data passed to extension registers is loaded from memory, being able to load an entire vector at once may reduce the number of cycles required to initialise an extension instruction's inputs. This is actually an orthogonal issue to extension instructions (any architecture could benefit from this extension), but with vector registers already implemented and extension instructions requiring significant data bandwidth it is a good match for AISE extended processors.

5.2.1 Evaluation Methodology

This idea was evaluated without actually changing the hardware, the simulator or the compiler. Instead information from the simulator was used to provide a dynamic estimate.

The simulator is run in "trace mode", in this mode it prints exact information about what every instruction that is processed is doing. Information regarding memory access is extracted from this output, specifically the complete list of addresses read from and a separate list of all addresses written to. These lists were split into segments based on when the "program counter" changed by a value other than four (i.e. the programs execution jumped or branched to a different basic block). The resulting lists of memory accesses only relate to data accesses (via a `ld*` or `st*` instruction), the implicit loading of the next instruction for execution was not included.

Each segment in each of these lists was then searched for potential places to use vector loads or stores. This was achieved by looking for accesses to consecutive addresses in each segment. The order of access in each segment was assumed to be flexible. Every sequence of adjacent accesses was marked as being loaded (or stored) by a vector. Up to four words can be loaded/stored by one vector instruction but double and triple loads/stores were also considered. For every operation marked as being handled by a vector load or store, two cycles were removed from the total number of cycles executed. For every vector load or store added two cycles were added to the number of cycles executed. This means each vector load or store has a 2–6 cycle benefit.

This evaluation method assumes the compiler is able to statically determine when memory accesses are to consecutive addresses. As most accesses are to the stack or arrays this is not an unreasonable assumption. The cases where it is not possible to determine this statically but it does occur dynamically should be rare. E.g., two adjacent arrays may result in adjacent address dynamically but not statically – but this will only occur for the two addresses where they are adjacent. The dynamic case will also miss cases which may be statically exploitable, e.g. a compiler may be able to vectorise accesses from a loop, but this evaluation technique will not identify those as each loop iteration would trigger a new segment.

The limitations of this estimation method means that vector loads and stores are proposed

for the entire register file (i.e. `r0–r63`) even though currently only `r31–r59` are mapped to vectors. This is necessary because *GCC* rarely loads data directly to the upper half of the register file. If an actual hardware implementation were to be created it would possibly only be able to load to, or store from, the upper half of the register file. If compiler support for vector loads and stores was implemented then an appropriate register class could be added to the register allocator to specify that vector loads and stores could only access `r32–r59`. The benefits of vector loads and stores would mean these registers would be used.

For vector loads and stores to be able to operate on the entire register file in the current *EnCore* would require two changes. Firstly, the register file would need to be modified to allow this – the 128-bit path is currently only connected to the upper-half of the register file. Secondly, the instruction encoding would need to be modified to allow for the 8 additional vector registers that would be addressable. This could be avoided by not allowing extension instructions (other than vector loads and stores) access to the lower half of the register file.

5.3 Replacing Wide Registers with Wide Instructions

Another, more radical, hardware change would be to eliminate vector registers completely. This is considered because section 4.9.4 put forward a case that vector registers were harming the performance of extension instructions. The alternative proposed here is to eliminate the vector registers and the extended register file and just have the extension operations operate on the baseline register file. Extension instructions would specify registers individually rather than by vectors. This would require an instruction larger than the current 32-bits, however, hence the name “wide instructions”.

This is expected to increase performance because there will be no register movement overhead associated with this type of extension instruction, other than any overheads which would also affect baseline instructions. There would be no need to move data to specific registers as the extension instruction could always use the registers where the data is already residing.

The changes to the *EnCore* processor that are required to support this are minimal. The 128-bit vector register interface would be removed as well as the permutation units associated with it. The extended register file (`r32–r59`) could also be removed. Keeping it could potentially still provide a benefit by simply providing more scalar registers, but experience with the *EnCore* back-end in *GCC* has shown that it rarely manages to use more than 32 registers.

Additions to the processor are also required. The baseline *EnCore* processor register file currently has two read-ports and two write-ports (the extended register file has three-read ports and two write-ports). As data is no longer being accessed by vectors a larger number of ports would probably be required. Current *EnCore* extension instructions can have three input vectors and two output vectors, or twelve inputs and eight outputs. It would not be possible to have

this many ports to the register file.

Pozzi and Ienne [2005] have shown that pipelining extension instructions, so as they read and write results over multiple cycles, can reduce port requirements without harming performance. They do not give results for a 2/2 input/output port constraint, but with pipelining 2/1 extensions always perform as well as 4/1 extensions without pipelining, and in some cases as well as 3/2 or 4/2. If the number of ports are increased then 3/2 pipelined instructions can perform as well as 8/2 non-pipelined instructions. Unfortunately the technique was not very effective at reducing the number of output ports required. Pipelining of extension instructions is not implemented in this thesis and is therefore not evaluated, but the possibility is considered when evaluating this technique.

5.3.1 Avoiding Extremely Wide Instructions

The above section shows that register file port constraints mean that extension instructions which access scalar registers must use fewer inputs and outputs than vector-based extension instructions. An additional restraint is that every individual register must be addressed and this will take up space in the instruction encoding.

In	Out	Addressable	Address-Bits	Instr-Bits
12	8	64	120	144
12	8	32	100	128
12	8	16	80	96
6	2	64	48	64
5	3	64	48	64
4	4	64	48	64

Table 5.1: *Possible register configurations and the number of bits required in the instruction word to address them.*

Table 5.1 shows several possible register configurations based on the number of inputs and outputs, and the number of registers each of those may address. The fourth column in the table is the number of bits required to address the registers in that configuration, and the fifth column is the minimum instruction word-size that could contain that many addressing bits. The instruction word size is calculated by adding 14 (the number of bits ARCompact uses for modes, op-codes, sub-op-codes and flags) and then rounding up to the nearest multiple of 16 (*EnCore's* minimum alignable size).

Three 12-input, 8-output configurations are shown. The first allows all 20 addresses to access any register, which results in a 144-bit instruction. The next two configurations keep the

20 addresses but reduce the number of registers that they may address. Lowering the number of addressable registers would have a small effect even if the extended register file has been removed: registers `r59`–`r63` have special purposes in ARCompact. Scalar instructions could still address them however, and as they are only used for control-flow tasks or specifying long immediates then extension instructions should only occasionally need to address these and `mov` instructions could be used copy data as necessary. For addressing 16 registers the ARCompact ISA already specifies a subset of the full 64 registers that are used by its 16-bit instructions. So either this could be used or the set of 32 registers could be partitioned into two groups of 16 – though that would eliminates some of the regularity that using scalar registers offers. Also, having 20 addresses and only 16 addressable registers is worse than having 5 vector addresses and 7 addressable vectors.

All three 12-input, 8-output configurations require a large instruction word and a large number of register file ports. As current ARCompact instructions are either 16 or 32-bits the next logical step is 64-bits. These leaves room for 8 addresses in the instruction, three configurations are evaluated: 6/2, 5/3 and 4/4. These would not need many more register file ports than the 2/2 that *EnCore* already provides. According to the work of Pozzi and Ienne [2005], described above, extension instructions with 6/2 operands should work with the existing 2/2 ports. 5/3 operands may require one additional write port, 4/4 operands may require two additional write ports (it would definitely require at least one). Extending the register file to have 2/3 or 2/4 register file ports is quite feasible.

The *EnCore* processor is already able to support varying instruction widths – the 7-stage version of *EnCore* has an align stage, the 5-stage version performs alignment as part of other stages. Modifying *EnCore* to support a 64-bit instruction and 8 register operands would not be a trivial task, but it would be possible.

5.3.2 Evaluation Methodology

As with the evaluation of vector register based extension instructions two binaries are compiled, one without extension instructions and one with. These are actually identical to the previously produced binaries as neither *MapISE* nor *binutils* have been updated to support 64-bit instructions. An accurate estimate has been calculated instead.

The simulator plug-in which executes the extension instructions on the simulator's behalf (i.e. the one generated by *ToolGen* for each new extension unit) was extended with performance counters. The plug-in counts the number of times each extension instruction is executed while the cycle counters are being used (so as not to count extension instructions which happen to get mapped to I/O code). Once the program is completed, the number of times each extension was executed is multiplied by the number of cycles it was expected to save, the benefit of all extension instructions is the summed to find the total benefit. Finally, this total benefit

is subtracted from the number of cycles the vanilla binary took to execute, resulting in an estimate of the number of cycles a binary using scalar register extension instructions would take to execute.

This estimate may sound similar to *ISEGen's* estimate, but it has two key differences: firstly, a real simulated run is used as a baseline; secondly, only extension instructions which the compiler could use are counted. The issues related to not considering the compiler back-end apply (described in section 4.9.1), but the issues concerning register allocation do not, as the register overhead has been eliminated.

5.4 Results - Reducing Register Pressure

Several experiments which are intended to reduce register pressure were proposed, they are evaluated below.

5.4.1 Reducing the Number of I/O Ports

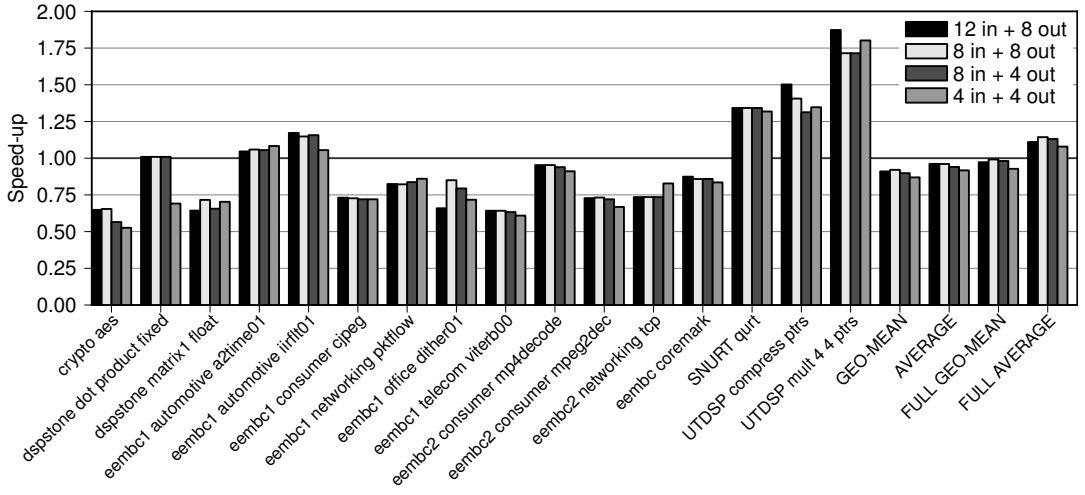
Figure 5.1 considers *MapISE's* effectiveness when dealing with extension instructions that have fewer inputs and outputs. Figure 5.1(a) shows the speed-ups that were obtained. The optimal number of inputs/outputs is 8/8. The default 12/8 resulted in an average speed-up of 1.11x, 8/8 instructions increased this to 1.14x. 8/4 operand instructions achieved a slightly lower average speed-up of 1.13x but 4/4 operand instructions resulted in the lowest average speed-up of all: 1.08x. From the perspective of *ISEGen* smaller extension instructions should result in slower code, though from the perspective of *MapISE* small instructions are easier to manage. The intersection of these two opposing views seems to be at 8/8 inputs/outputs.

In terms of the number of instructions that *ISEGen* produces and *MapISE* maps the search are more predictable: both increase as instruction size shrink (see figure 5.2 for *ISEGen*, and figure 5.1(b) for *MapISE*).

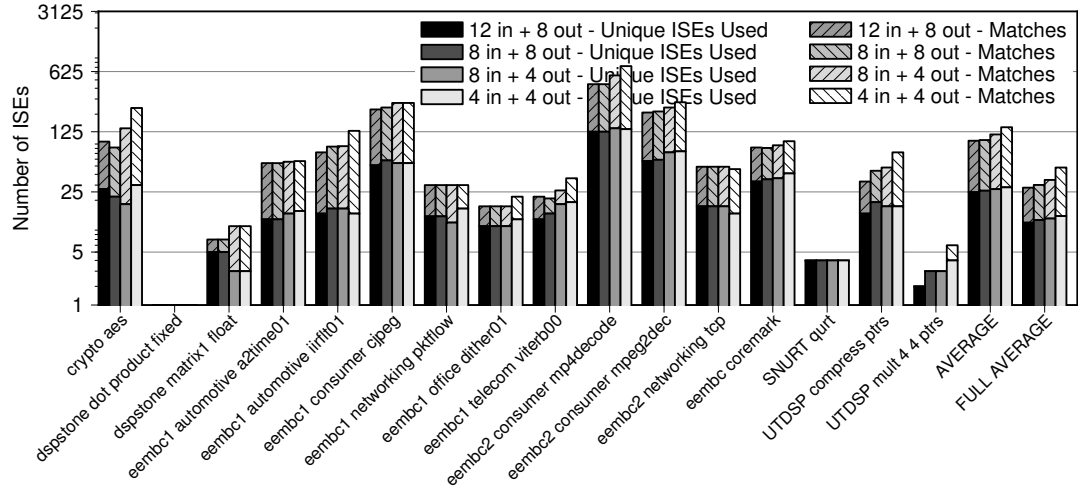
5.4.2 Hard-Wiring Constant Values

Hard-wiring constants into instructions was expected to increase performance by requiring less data to be passed via vector registers. Figure 5.3(a) confirms that hard-wire constants result in a significant speed-up. The default speed-up of 1.11x is increased to 1.20x. Very few benchmarks experience a slow-down because of hard-wired constants, none in figure 5.3(a), but there are a few benchmarks in the full data-set that do (see figure B.25).

The average number of extension instructions mapped, however, was reduced from an average of 28.01 instructions per benchmark to 23.24 instructions per-benchmark (see figure 5.3(b)). This is to be expected as hard-wiring constants into extension instructions makes them more application specific and thus reduces their applicability.



(a) Speed-ups. *Note: the full version of this chart is figure B.22 on page 198.*



(b) Mapping quality information. *Note: the full version of this chart is figure B.23 on page 200.*

Figure 5.1: *A comparison of different register port constraints.*

5.4.3 Modifying the ISEGEN Heuristic Parameters

This section evaluates the register load cost *ISEGen* heuristic proposed in section 5.4.3. The "Eliminating Poor Mappings" described in section 4.5 provided the motivation for the heuristic. It worked by preventing some extension instructions from being used in mapping. Equivalently it is expected that this register load cost heuristic will prevent some extension instructions from being evaluated. This is considered first using figure 5.4. Figure 5.4(a) displays how many extension instructions are generated for several different values of the register load cost heuristic parameter. This chart shows a strong and consistent trend towards fewer extension instructions as the register cost heuristic parameter increases. The results decrease from 18.30 unique and 33.12 total for a parameter value of 0.00 (i.e. the default) to 3.25 unique and 4.75

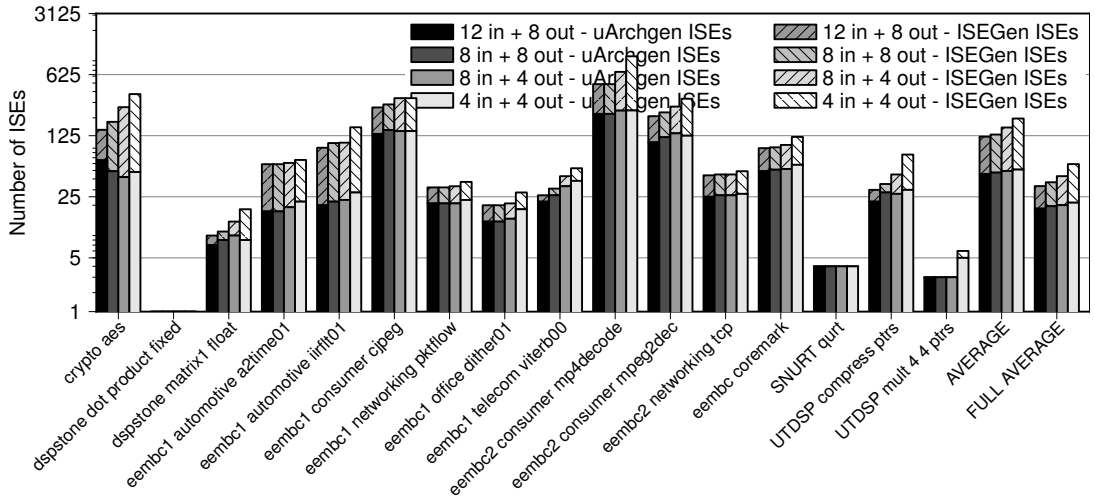


Figure 5.2: *The effect that different register port constraints have on the number of extension instructions that* ISEGen *and* uArchGen *will produce.* Note: the full version of this chart is figure B.24 on page 202.

total for a parameter value of 1.25.

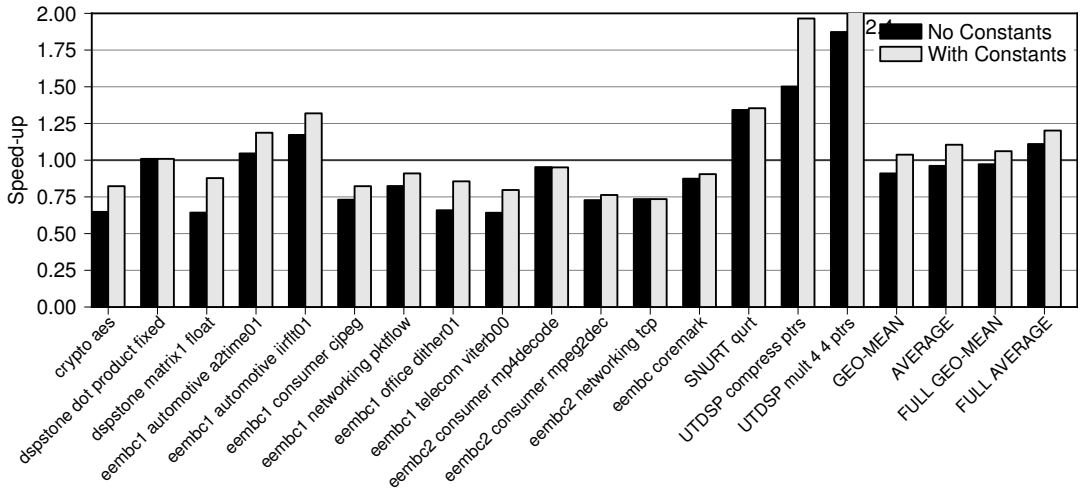
Figure 5.4(b) displays how many extension instructions *MapISE* is able to map with presented with instructions generated using different load cost heuristic parameters. The results decrease from 10.98 unique and 28.02 total for a parameter value of 0.00 (i.e. the default) to 2.27 unique and 4.07 total for a parameter value of 1.25.

Figure 5.5 shows the actual speed-ups found when using the extension instructions generated with the different load cost heuristic parameters. It was expected that this parameter would increase speed-up, but surprisingly the parameter which resulted in the highest speed-up was 1.25. This value causes 82% of the unique extension instructions to be discarded. A parameter value of 0.00 results in an average speed-up of 1.11x, a parameters value of 1.25 produces an average speed-up of 1.24x. Parameters of 0.50 and 1.00 have speed-ups of 1.17x and 1.22 respectively, going higher than 1.25, up to 1.50 slightly reduced the average speed-up to 1.23x (as can be seen in figure B.31 on page 216).

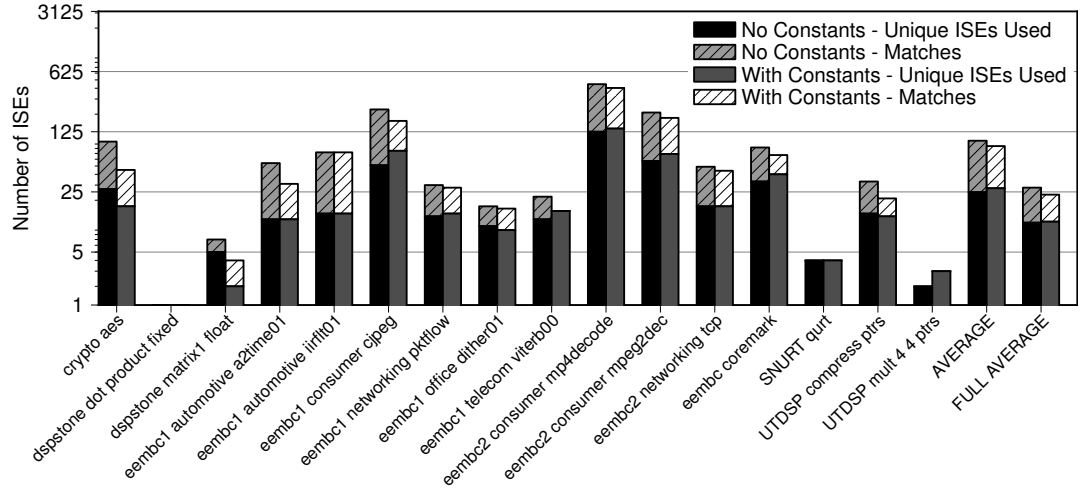
Focusing on such a small-number of key extension instructions which are able to produce high speed-ups, despite vector register related overhead can clearly provide good results. It also validates the hypothesis that most of *MapISE's* lost performance is due to the vector register extension interface.

5.4.4 Combining Techniques

As hard-wiring constant values in extension instructions provides a speed-up of 1.20x, and a register load cost heuristic parameter of 1.25 results in a speed-up of 1.24x, it was considered



(a) Speed-ups. *Note: the full version of this chart is figure B.25 on page 204.*

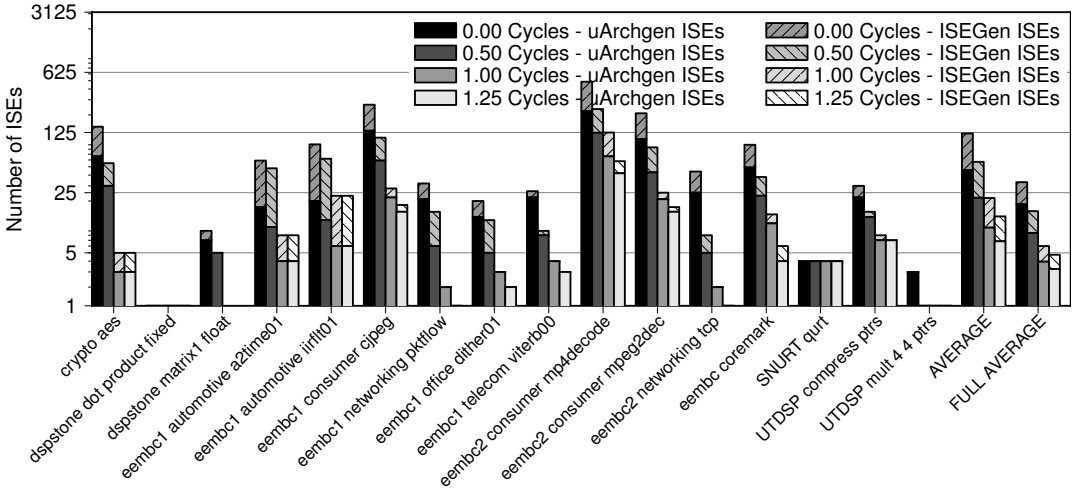


(b) Mapping quality information. *Note: the full version of this chart is figure B.26 on page 206.*

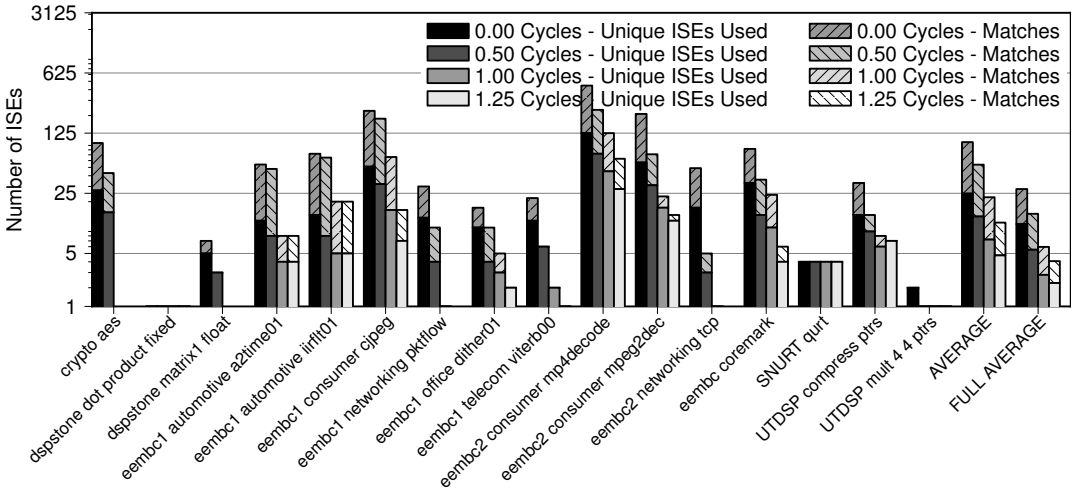
Figure 5.3: *The effect that introducing hard-wired constant values into extension instructions has on the end results and* MapISE's *ability to use*

possible that the two combined would be better than either performed alone. Figure 5.6 shows that the result is a higher average speed-up: 1.26x. This increase, however is marginal. A possible reason for this is that they may have both been benefiting from eliminating the same flaws in the default extension instructions.

No other techniques are considered for combination, as the resulting design space would be too large for consideration in this chapter.



(a) The effect that the register load cost parameter has on the number of extension instructions that *ISEGen* finds. *Note: the full version of this chart is figure B.27 on page 208.*



(b) *MapISE's* ability to use the instructions found with different register load cost parameter values. *Note: the full version of this chart is figure B.28 on page 210.*

Figure 5.4: *A comparison of different register cost values to provide to* ISEGen's *heuristics. Note: Figures B.35 and B.29 on pages 224 and 212 cover additional parameter values (0.00, 0.25, 0.75 and 1.50 cycles).*

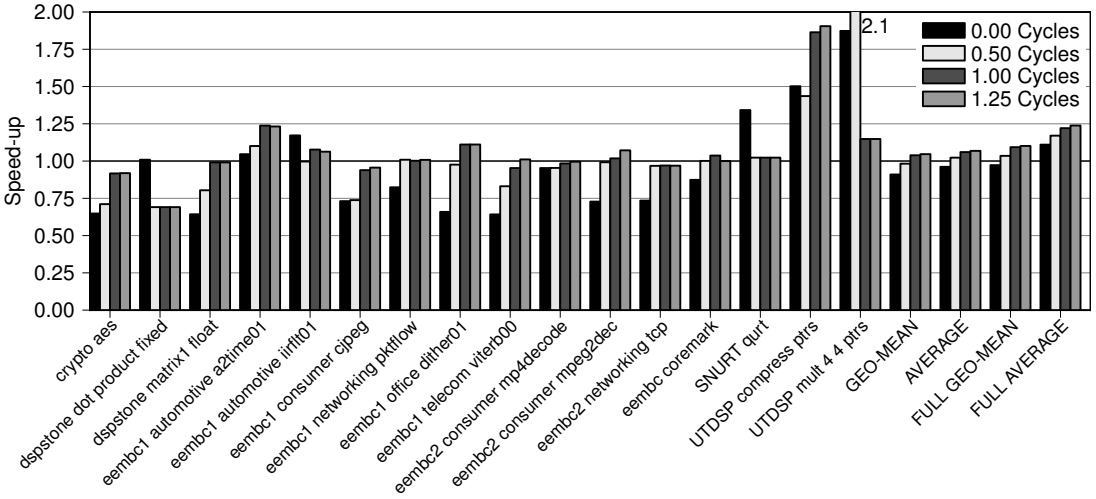


Figure 5.5: *The speed-ups obtainable for extension instructions produced with different register cost parameter values. Note: Figure B.31 on page 216 is the equivalent of this chart but with different parameter values (0.00, 0.25, 0.75 and 1.50 cycles).* Note: the full version of this chart is figure B.30 on page 214.

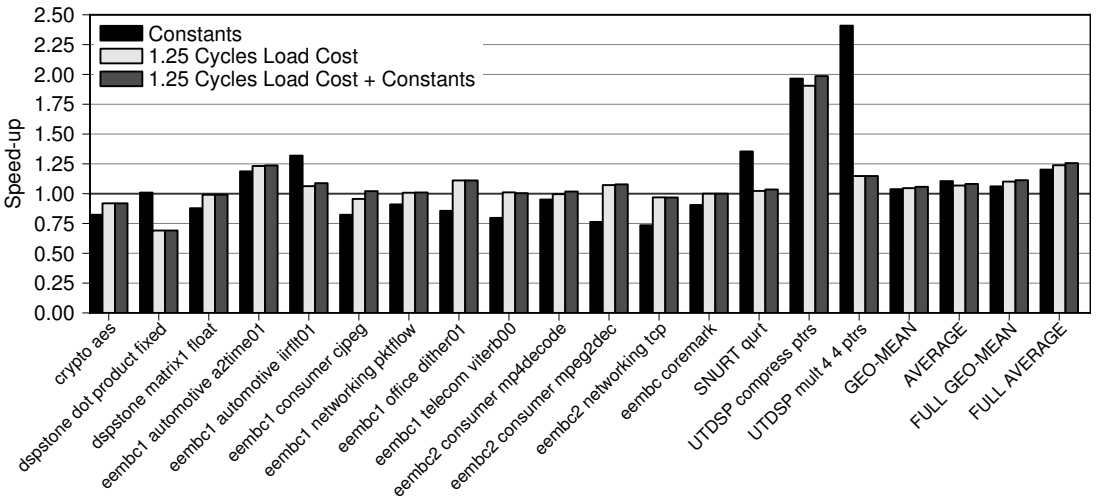


Figure 5.6: *A comparison of hard-wired constants alone, a register load cost of 1.25 cycles alone, and the two combined as single run.* Note: the full version of this chart is figure B.32 on page 218.

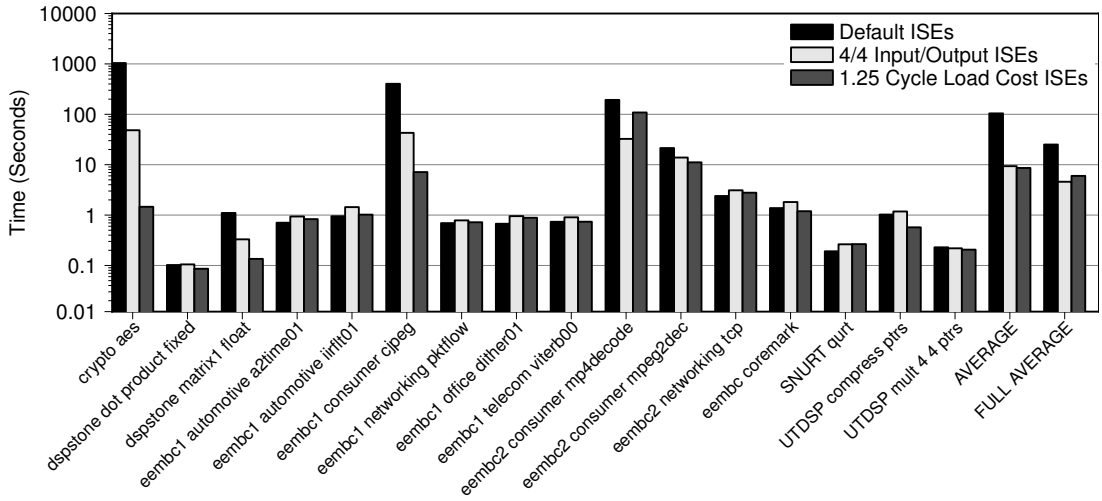


Figure 5.7: *A run-time comparison for* MapISE *when presented with default extension instructions, instructions with a 4/4 input/output constraint, and instructions generated with a 1.25 cycle load cost.* Note: the full version of this chart is figure B.33 on page 220.

5.4.5 *MapISE* Timing

Figure 5.7 evaluates whether simple extension instructions makes *MapISE* faster. The average time to run *MapISE* on one benchmark with the default extension instructions is 25.1 seconds, for 4/4 input/output extension instructions this falls to 4.6 second, setting a register load cost of 1.25 does almost as well, taking 6.0 seconds on average. This also shows that a large number of small extension instructions (4/4 input/output) are easier to compile with than a small number of large extension instructions (1.25 register load cost). This is because graph sub-graph isomorphism checking is an NP-Complete problem, so the run-time grows much more quickly with larger graphs than the linear increase of checking a long list of extension instructions. The worst-case run-time (Crypto *aes*) is also dramatically improved, it is 1040 seconds with default extension instructions, 48.29 seconds with 4/4 input/output instructions and 1.46 seconds with 1.25 register load cost.

5.5 Results - Wide Memory Bus

Figure 5.8 shows the speed-ups obtainable by adding vector loads and stores. It shows that the technique is able to provide some benefit for about two out of three benchmarks. The average speed-up increases from 1.11x to 1.14x. Not nearly as large a gain as some other results, but as it is orthogonal is extension instructions this gain could be added to any of the vector register experiments.

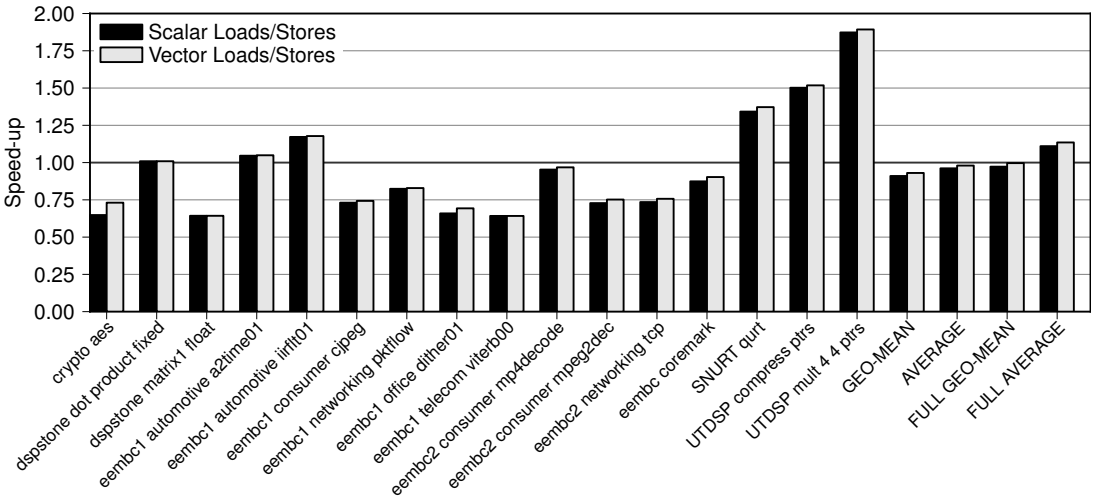


Figure 5.8: *A comparison of a system with no vector loads or stores (the default) with one that does have them.* Note: the full version of this chart is figure B.34 on page 222.

The loop-unrolled versions of UTDSP seem to benefit significantly from vector loads and stores, see figure B.34 on page 222. This makes sense, unrolling means larger basic blocks, which means more opportunity for parallel memory accesses. The whole UTDSP suite seems to benefit from this addition, this is likely because it is a DSP suite, which maps well to vector hardware.

5.6 Results - Wide Instructions

Figure 5.9 compares the new scalar register scheme to the old vector register scheme. This is based on 12-input and 8-output registers, although this has already been described as unrealistic it is the best choice for a direct comparison; instructions with fewer registers are evaluated below. Overall the new scheme is very successful, the average speed-up increases from 1.11x to 1.28x.

Additionally a different trend, compared to previous results, can be noted for the geometric mean. For vector registers the geometric mean is noticeably lower than the average (geometric mean: 0.97, average: 1.11), for scalar registers there is less of a difference (geometric mean: 1.23, average: 1.28). What this means is that the performance of scalar registers is more consistent than that of vector registers. If every benchmark experienced exactly the same speed-up then the geometric mean and the average would be identical, the larger the variance in the results the larger the difference between the two. The scalar register results may be more consistent because of the way they are evaluated. When calculating an estimated benefit, no

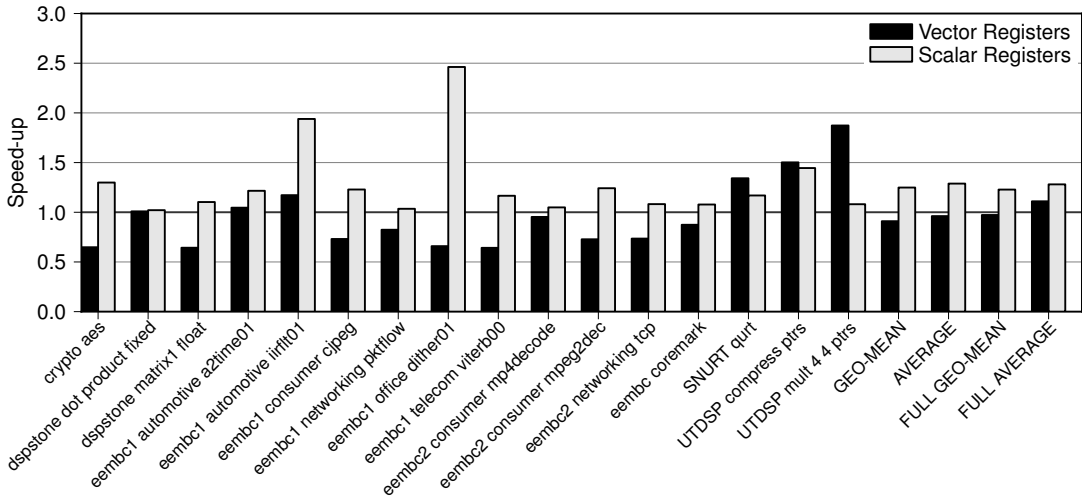


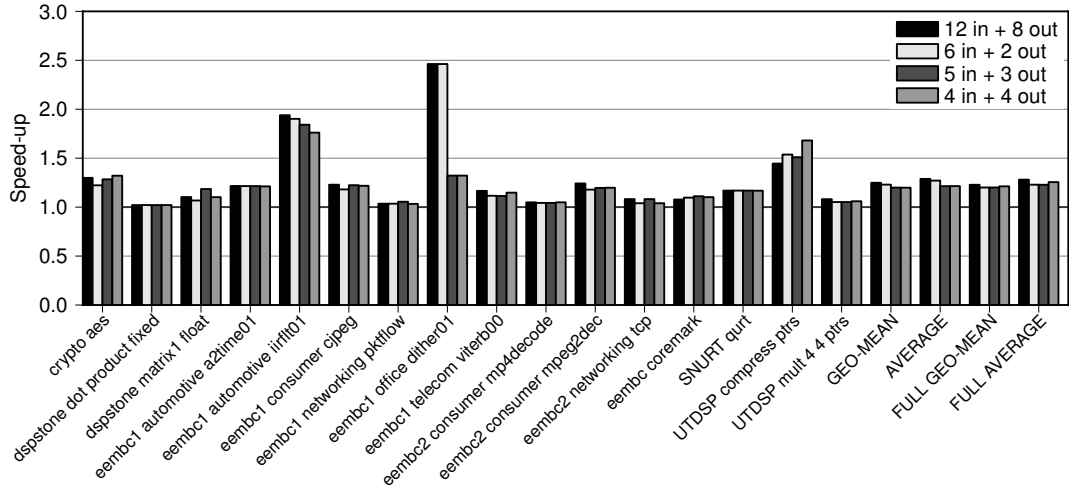
Figure 5.9: *The speed-ups provided by using extension instructions with vector registers (left-hand bar) or scalar registers (right-hand bar).* Note: the full version of this chart is figure B.36 on page 226.

benchmark will ever experience a slow-down, at worst their speed-up will be 1.0x. This is quite accurate though, extension instructions based on scalar registers should rarely introduce slow-downs.

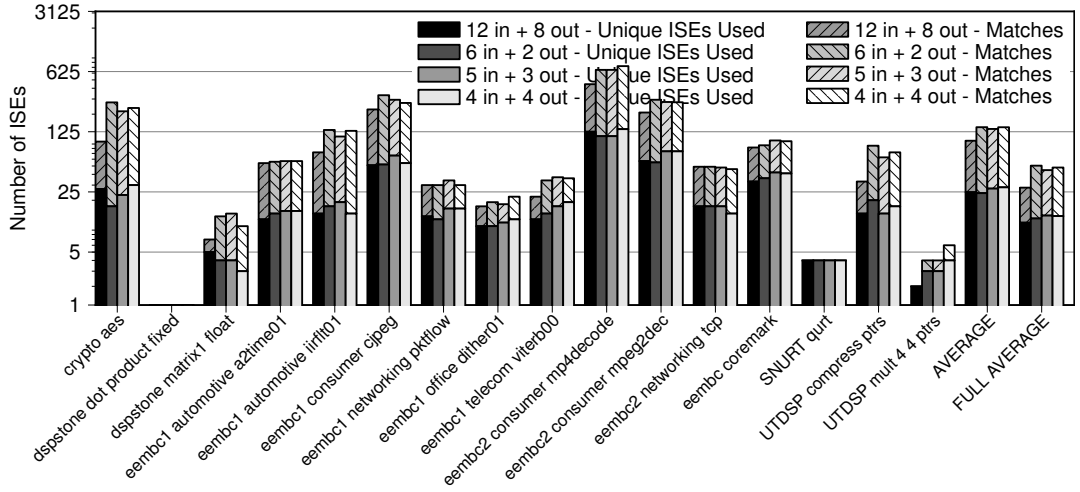
Individual benchmarks experience some large movements in either direction, though this is to be expected with a fundamental change like this. For example, EEMBC *dither01* moves from a speed-up of 0.66x to 2.46x, a factor of 3.73x change. Alternatively UTDSP *mult 4-4 ptrs* loses almost all of its acceleration when using scalar registers, going from a speed-up of 1.87x to 1.08x. In fact, for UTDSP benchmarks all the *ptrs* variations generally seem to prefer vector registers, whereas the *arrays* variations seem to prefer scalar registers. This is possibly due to a knock-on effect where vector registers are able handle data movement effectively in the *ptrs* variation, but not in the *arrays* set.

Figure 5.10 compares the scalar register results from figure 5.9 with more realistic scalar register configurations. Although a 12/8 inputs/outputs configuration resulted in the highest speed-up very little performance is sacrificed by using extension instructions with fewer inputs/outputs. Going from 12/8 to 4/4 changes the average speed-up from 1.281x to 1.256x. 4/4 is actually the fastest of the realistic configurations, 6/2 and 5/3 both have an average speed-up of approximately 1.23x, 5/3 is very slightly lower than 6/2. EEMBC *dither01*, which benefited so much from scalar registers, critically requires 6 input registers to achieve good performance, 6/2 inputs/outputs achieves the same performance as 12/8, but 5/4 or 4/4 both sacrifice almost of the application's acceleration.

That 4/4 instructions provide a better result than 6/2 or 5/3 is interesting. As most arith-



(a) Speed-up. *Note: the full version of this chart is figure B.37 on page 228.*



(b) Mapping quality information. *Note: the full version of this chart is figure B.38 on page 230.*

Figure 5.10: *An evaluation of different input and output constraints for scalar register based extension instructions.*

metic operations take two inputs and produce one output it is generally assumed that extension instructions should have more inputs than outputs. In a similar experiment with vector registers 4/4 was found to be the slowest of the set evaluated, see figure 5.1(a). As the result is an average over 179 benchmarks it seems unlikely that this is an anomaly. To investigate whether this is likely due to *MapISE* using 4/4 instructions more effectively than others, or whether 4/4 instructions performed better acceleration *ISEGen's* estimated speed-ups can be viewed in figure 5.11. That chart shows the extension instructions generated for 6/2 operands are markedly worse than those for 5/3 or 4/4. Instructions with 5/3 and 4/4 operands have similar predicted performance to each other, an average speed-up of 1.40x for 5/3 operands and 1.39x for 4/4. So

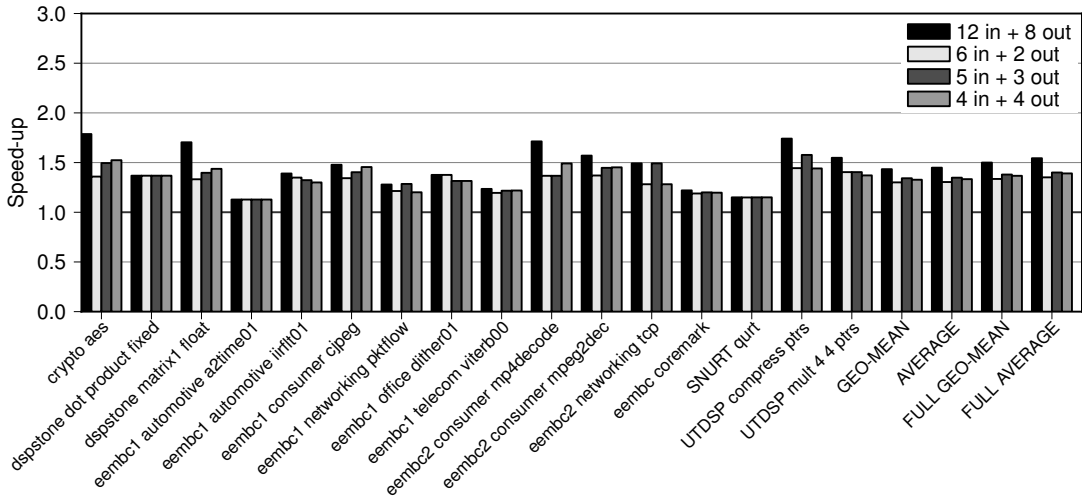


Figure 5.11: ISEGen's *predicted speed-ups for various register constraints which are realistic for scalar register based extension instructions, the default 12-input, 8-input mode is also included for comparison.* Note: the full version of this chart is figure B.39 on page 232.

extension instructions with 4/4 operands are predicted to be good by *ISEGen*, though it predicts 5/3 operands as the best but *MapISE* determined that they were the worst. The reason for 4/4 operands being the best is most likely a combination of those being good extension instructions (as shown by *ISEGen's* estimated speed-ups in figure 5.11) and that *MapISE* can map more 4/4 operand instructions than 5/3 (see figure 5.10(b)). So, in summary, it is quite possible that the changes proposed in this chapter will both improve performance and reduce hardware size.

5.7 Results - Retargeting Extension Instructions

The retargeting of extension results is considered in appendix C. In that appendix the experiments performed in section 4.8 are repeated for three of the changes proposed above: "4/4 Input/Output Registers", "Hard-Wiring Constant Values" and "Wide Instructions". The results are not summarised as they do not demonstrate anything that was not already shown in section 4.8 – they are just included in an appendix so as that may be verified. The only difference in the results is that "Wide Instructions" seem to be slightly worse for retargeting than vector register based extension instructions. This seems surprising and may just be an artifact of the evaluation model.

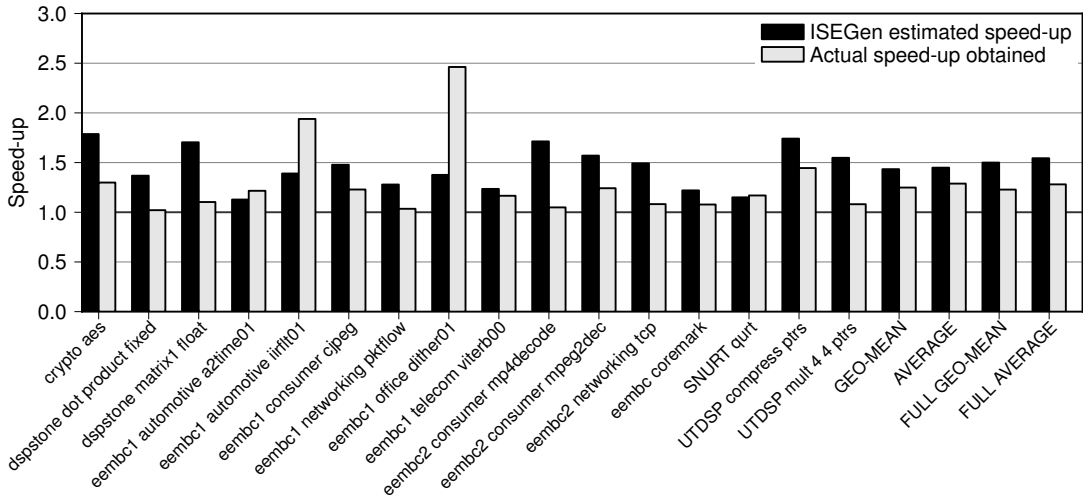


Figure 5.12: *A re-evaluation of the comparison of* MapISE's *performance with* ISEGen's*, but now* MapISE *is using scalar register based extension instructions (or "wide instruction").* Note: the full version of this chart is figure B.40 on page 234.

5.8 Critical Evaluation

Most of *MapISE* was critically evaluated in section 4.9. *MapISE's* relation to *ISEGen* can be re-evaluated, however.

The limitations of the estimation model for "wide instructions" have already been addressed in section 5.3.2.

5.8.1 Reevaluation of ISEGEN Issues

Figure 5.12 re-evaluates how *MapISE* with "Wide Instructions" (using 12-inputs and 8-outputs for comparative purposes) performs compared to *ISEGen*. It finds that *MapISE* still falls short with an average speed-up of 1.28x compared to *ISEGen's* estimate of 1.54x. This is because the same performance estimate issues described in section 4.9.1 still apply.

The figure does show that an interesting subset has arisen. Now that slow-downs have been eliminated a set of benchmarks has emerged that experience very little speed-up, in spite of *ISEGen's* estimates.

This chapter has proposed multiple hardware changes to the processor without any thorough analysis of how much more die-space would be required and what effect the additional complexity would have on *EnCore's* maximum clock frequency. Notes on the general effects of hardware changes where provided, and most changes to *ISEGen's* parameters and heuristics should result in smaller extension instructions providing lee-way for other changes. Figure 5.13 provides a brief overview of some potential hardware savings due to different *ISEGen*

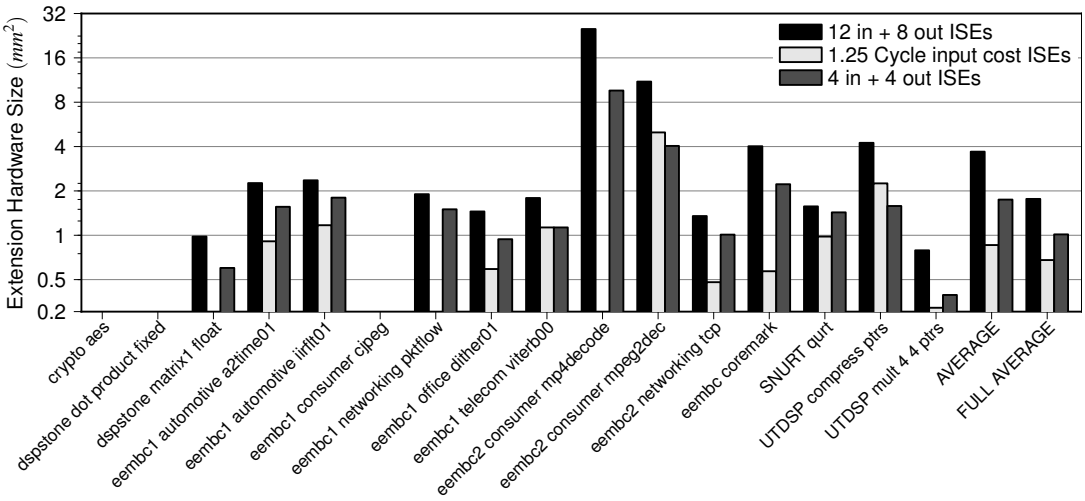


Figure 5.13: *Extension unit on-die hardware size per-benchmark for different hardware configurations, based on a* 130*nm process.* Note: the full version of this chart is figure B.41 on page 236.

parameters. The figures presented are *uArchGen's* estimated size of the extension unit required for each benchmark implemented using a 130*nm* process. Some benchmarks have no data as *uArchGen* was unable to estimate a size for the extension unit. To provide some meaning to the scale and very small *EnCore* processor with an 8KB of direct-mapped instruction cache and data cache was $1mm^2$ on a 130*nm* process. A larger *EnCore* processor with 32KB of 4-way associate instruction cache and data cache and a set of 4-input, 4-output extension instructions was $2.25mm^2$ on a 90*nm* process, which would approximately scale up to $6.76mm^2$ on a 130*nm* process. The trends in figure 5.13 are that setting a high register load cost dramatically reduces hardware size by selecting far fewer instructions (cross-reference with figure 5.4(a) to see this). Setting a 4-input, 4-output constraint also resulted in a reduced hardware size, though not by as much, this time more instructions were found but they were much smaller (cross-reference with figure 5.2 to see this).

There is one issue with the changes that have been made to *ISEGen* that potentially affects two experiments in this chapter. *ISEGen* considers constants as inputs (e.g. it would be one of the 12 inputs for default extension instructions). This affected the "Hard-Wiring Constant Values" experiment because it should have been able to have a larger number of inputs available (e.g. 12-inputs plus 5-constants). Though keeping the input constraints at 12 may have ensured that the size of the generated hardware and *ISEGen's* run-time stayed under control.

This issue also affected the experiment that combined "Hard-Wiring Constant Values" with the register load cost heuristic parameter. As each constant was considered an input they each had a cost of 1.25 cycles associated with them. There should not have been any cost, which

would have likely encouraged *ISEGen* to create extension instructions with many constant inputs.

5.9 Summary and Conclusions

5.9.1 Future Work

Two changes which could be made to *ISEGen* to enhance the experiments in this chapter are to accurately allow constants to be mixed with other heuristics and to make *ISEGen* vector load aware. This would require *ISEGen* to be aware how data is laid out in memory and access it in blocks of four.

Obvious future work would be to implement any of the estimation models presented in this chapter, first in the simulator/compiler/binutils as required, and then in hardware.

5.9.2 Summary

This chapter has presented multiple techniques for improving the usefulness of AISE. Every technique presented resulted in some level of improvement, and the general trend was that these techniques produced faster code, that took less time to compile and the resulting hardware was smaller.

The two best techniques were the addition of a register load cost to *ISEGen's* heuristic model and and new "Wide Instruction" format which allowed scalar registers to be addressed individually. The resulted in speed-ups of 1.24x and 1.26x respectively.

Chapter 6

Increasing Memory Bandwidth: Dual Memory Banks

"The question of whether a computer can think is no more interesting than the question of whether a submarine can swim."
— Edsger Dijkstra, *computer scientist, ACM Turing Award winner*, 1930–2002.

This chapter describes different methods of exploiting dual memory banks at a source-level. It reconciles the idea of increasing memory bandwidth with vector loads and store (see section 5.2) with the discover that vector registers hinder extension instruction performance (see section 5.3). Memory bandwidth is increased by using dual on-chip scratchpad memories which may be simultaneously, but independently, accessed. This allows up to double the memory bandwidth. Due to lack of hardware or simulator support within the PASTA platform, however, the techniques are evaluated on the TigerSHARC DSP platform using the UTDSP benchmark suite.

The previous two chapters presented methods of exploiting extension instructions in applications within the PASTA framework. Some other AISE-based extension methodologies, however, make use of scratchpad memories [Biswas et al., 2006b]. The PASTA framework avoided the use of these within extension instructions to keep separate concerns orthogonal. It is still possible, however, to add scratchpad memories into the framework while staying true to this design principle. This could be done by adding conventional scratchpad memories that are not directly integrated with the extension instructions but are connected to the baseline processor, as shown in figure 2.6 on page 22. An additional advantage of keeping scratchpad memories orthogonal to extension instructions is that it is not necessary to worry about the state of extension instructions on a context switch as they are stateless.

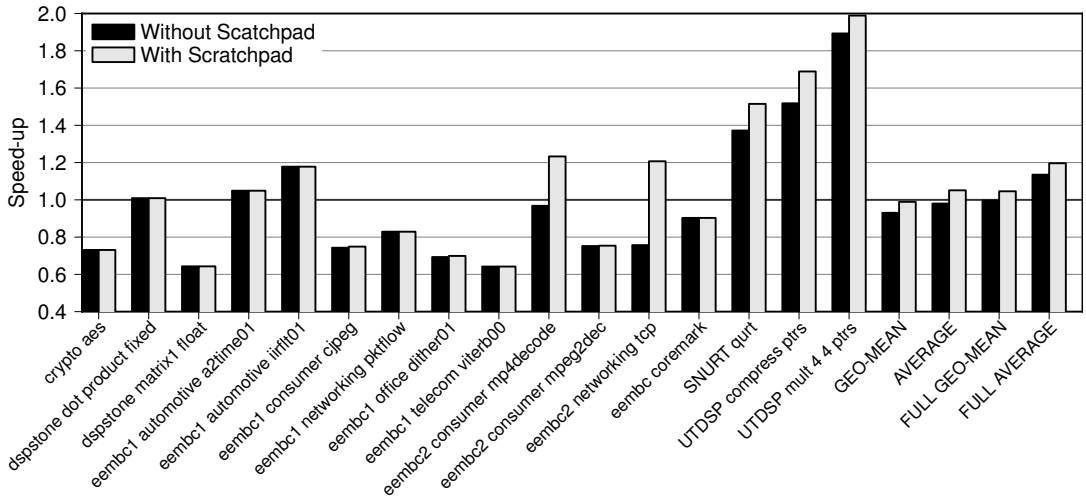


Figure 6.1: *These results are speculative estimates (see text) and are not as reliable as related results in chapters 4 and 5. The left bar for each benchmark is the speed-up achieved by adding extension instructions with vector registers and vector load/stores to the baseline processor, as described in section 5.2. The right bars add a scratchpad memory to this.* Note: the full version of this chart is figure B.42 on page 238.

6.1 Feasibility Study

Figure 6.1 shows the estimated benefit of extending the vector load/store model described in section 5.2 so that all of the projected vector load/stores load from and store to a scratchpad that is guaranteed to already contain the required data. For small benchmarks this is quite feasible, for larger benchmarks it would require that the scratchpad is managed by preemptive DMA operations. The results in figure 6.1 assume the compiler has inserted these operations and treat all vector loads and stores as cache-hits, which they effectively would be if the data was on a scratchpad memory. Thus these results are speculative but are a useful indicator as to what the actual effect of a scratchpad memory would be. The results estimate that the overall average speed-up will increase by an additional 6 percent over baseline, from 1.135x to 1.196x, when using a scratchpad memory. Many benchmarks already perform well with just a cache and thus see very little speed-up from the scratchpad (e.g. EEMBC *viterb00*). Data intensive benchmarks, however, see large speed-ups (e.g. EEMBC *mp4decode*).

The *EnCore* processor (used in chapters 4 and 5) implements the ARCompact ISA which supports dual memory banks through an extension called XY Memory [ARC International, 2010]. The *EnCore* processor, however, does not implement this extension. The processor could be extended to support the XY memory instructions and the simulator and its memory model modified accordingly. Though as extension instructions in the PASTA framework are

orthogonal to the memory model the two can be evaluated separately. Thus the evaluation performed in this chapter uses a TigerSHARC DSP processor which has dual memory banks. Any increases in memory bandwidth that are found may be applied to extension instructions results, much like the estimates in figure 6.1.

6.2 The Problem

Section 2.5 described dual memory banks, but did not discuss the problem of deciding how to partition data between memory banks. The primary requirement when performing this partitioning is that variables should be assigned in a way that allows both memory banks to be read simultaneously. This is a difficult problem, as will be discussed in section 6.2.1. Section 3.2 evaluated previous approaches to solving this problem. The rest of this chapter will describe some new approaches based on a C-to-DSP-C source-to-source compiler.

Operating at the source-level has two advantages. The first is that every technique in this chapter can be instantly targeted to any dual memory architecture that has a DSP-C compiler. The second advantage is that source-level partial pre-assignments are possible, so a library can ensure that certain data gets placed on a specific bank, or the programmer can manually assign performance critical variables if the compiler does a poor job.

6.2.1 Difficulty of the Problem

Efficient assignments of variables to memory banks can have a significant performance impact, but are difficult to determine. For instance, consider the example in figure 6.2. It shows the `lmsfir` function from the UTDSP *lmsfir-8_1* benchmark. This function has five parameters that can be allocated to two different banks. Local variables are stack allocated and outside the scope of explicit memory bank assignment. On the lower half of figure 6.2 four of the possible legal assignments are shown. In the first case, as illustrated in figure 6.2(a), all data is placed in the *X* memory bank. This is the default case for many compilers when no explicit memory bank assignment is specified. Clearly, no advantage of dual memory banks can be taken and this assignment results in an execution time of 100 cycles for the Analog Devices TigerSHARC TS-101 platform. The best possible assignment is shown in figure 6.2(b), where `input` and `gain` are placed in *X* memory and `output`, `expected`, and `coefficient` in *Y* memory. Simultaneous accesses to the `input` and `coefficient` arrays have been enabled and, consequently, this assignment reduces the execution time to 96 cycles. Interestingly, an "equivalent" assignment scheme, as shown in figure 6.2(c), that simply swaps the assignment between the two memory banks does not perform as well. In fact, the "inverted" scheme derived from the best assignment results in an execution time of 104 cycles, a 4% slowdown over the baseline. The worst possible assignment scheme is shown in figure 6.2(d). Still, `input`

and `coefficient` are placed in different banks enabling parallel loads, but this scheme takes 110 cycles to execute, a 10% slowdown over the baseline.

At first, the significant performance impact resulting from the inverted assignment scheme appears non-intuitive. Analysis of the generated assembly code found that the performance gap does not result from the code for the two `for`-loops in the `lmsfir` function, but is due to differences in the code generated for the statements between the two loops. For the assignment shown in figure 6.2(b), `expected[0]` and `sum` can be accessed simultaneously (the compiler has decided to not keep `sum` in a register). At the same time the subtraction of `expected[0]` and `sum` takes place, `gain` is fetched from memory. The inverse assignment scheme depicted in figure 6.2(c) results in sequential memory accesses for `expected[0]` and `sum`, because, under this scheme both of the data objects are stored in memory bank *X*. Also, the access to `gain` is scheduled in the next slot and, thus, adds an extra instruction. While the overall impact is relatively small for this particular example, greater performance differences can be observed if the code inside a loop body is affected.

This example demonstrates how difficult it is to find the best source-level memory bank assignment. Source-level approaches cannot analyse code generation effects that only occur later in the compile chain, but must operate on a model generic enough to cover most of these. In this chapter a refined variable interference graph construction is proposed together with a fast and scalable soft colouring algorithm capable of handling complex DSP applications and allowing for partial pre-assignments where required.

6.3 Methodology

The memory bank assignment schemes comprise of the following the stages:

1. **Group Forming.** In this stage groups of variables are formed that must be allocated to the same memory bank due to pointer aliasing.

2. **Interference Graph Construction.** An edge-labelled graph representing potential simultaneous accesses between variables is constructed during this stage.

3. **Colouring of the Interference Graph.** Finally, the nodes of the interference graph are coloured with two colours (representing the two memory banks) with the aim to maximise the benefit from simultaneous memory accesses.

Of these three stages only stage one is critical for correctness, whereas approximations are acceptable for stages two and three, i.e. an inaccurate interference graph or a non-optimal colouring still result in correct code that, however, may or may not perform optimally.

```
void lmsfir(float input[], float output[],
            float expected[], float coefficient[],
            float gain)
{
    /* Variable declarations omitted. */

    sum = 0.0;
    for (i = 0; i < NTAPS; ++i) {
        sum += input[i] * coefficient[i];
    }
    output[0] = sum;
    error = (expected[0] - sum) * gain;
    for (i = 0; i < NTAPS-1; ++i) {
        coefficient[i] += input[i] * error;
    }
    coefficient[NTAPS-1] =    coefficient[NTAPS-2]
                            + (input[NTAPS-1] * error);
}
```

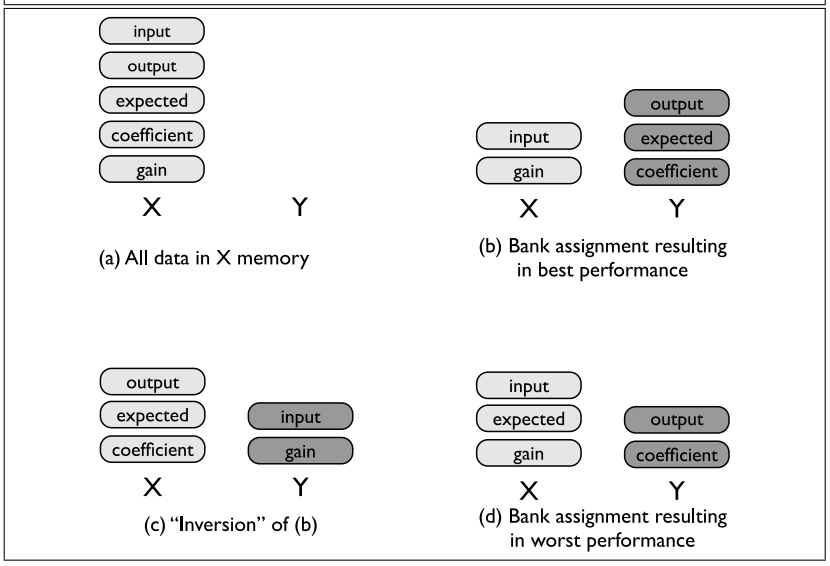
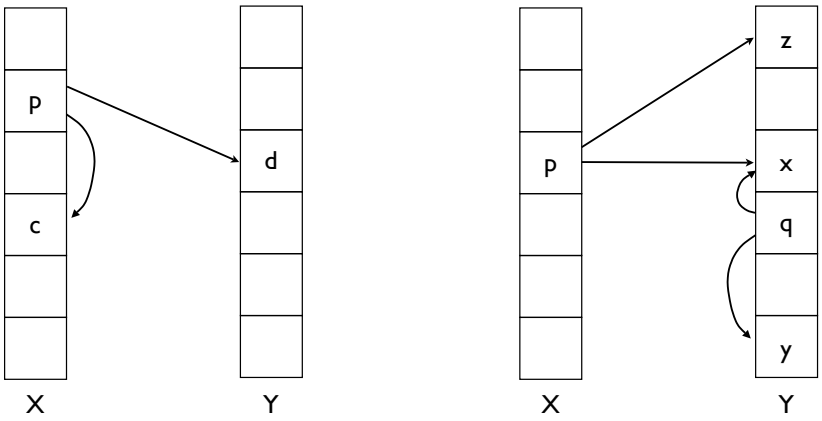


Figure 6.2: `lmsfir` *function with four memory bank assignments resulting in different execution times.*



(a) Incompatible pointer assignments. (b) Pointer induced variable grouping.

Figure 6.3: *Incompatible pointer assignments and pointer induced grouping.*

6.3.1 Group Forming

Group forming is the first stage in the memory bank assignment scheme. It is based on pointer analysis and summarises those variables in a single group that arise through the *points-to* sets of one or more pointers. All variables in a group must be allocated to the same bank to ensure type correctness of the memory qualifiers resulting from memory bank assignment.

Figure 6.3 illustrates this concept. In figure 6.3(a) the pointer p may point to c or d, which are stored in memory banks X and Y, respectively. This eventually causes a conflict for p because both the memory bank where p is stored and the bank where p points to must be statically specified. Thus, p must only point to variables located in a single bank. A legal assignment would place c and d in the same bank as a result of previous grouping. This grouping is shown in figure 6.3(b) for two pointers p and q. In this example, p may point to variables x and z at various points in the execution of a program and, similarly, q is assumed to point to x and y. Grouping now ensures that x and z are always stored in the same bank (due to p), and also x and y (due to q). By transitivity, x, y and z have to be placed in the same memory bank. Note that p and q themselves can be stored in different memory banks, only their targets must be grouped and located in a single memory bank.

In algorithm 1 a working list algorithm for group forming is presented. It is assumed that points-to sets for all pointer variables are available, e.g. through prior pointer analysis of the program [Rugina and Rinard, 1999]. The algorithm operates on the set of variables V to group. Initially, the algorithm places each variable v_k in a singleton group g_k, these are then merged into larger groups. For each pointer p in the set of variables V the points-to set is calculated and the groups of the corresponding variables merged. This process is repeated until all pointers

Algorithm 1 *Group Forming(Variables V)*

Require: Points-to for all pointers

Ensure: Type-safe variable grouping

1: **for all** $v_k \in V$ **do**
2: Form singleton group g_k containing v_k
3: **end for**
4: $L \leftarrow \{p | p \text{ is a pointer} \wedge p \in V\}$
5: **while** $L \neq \emptyset$ **do**
6: Select $p \in L$
7: Merge groups(points-to(p))
8: **end while**

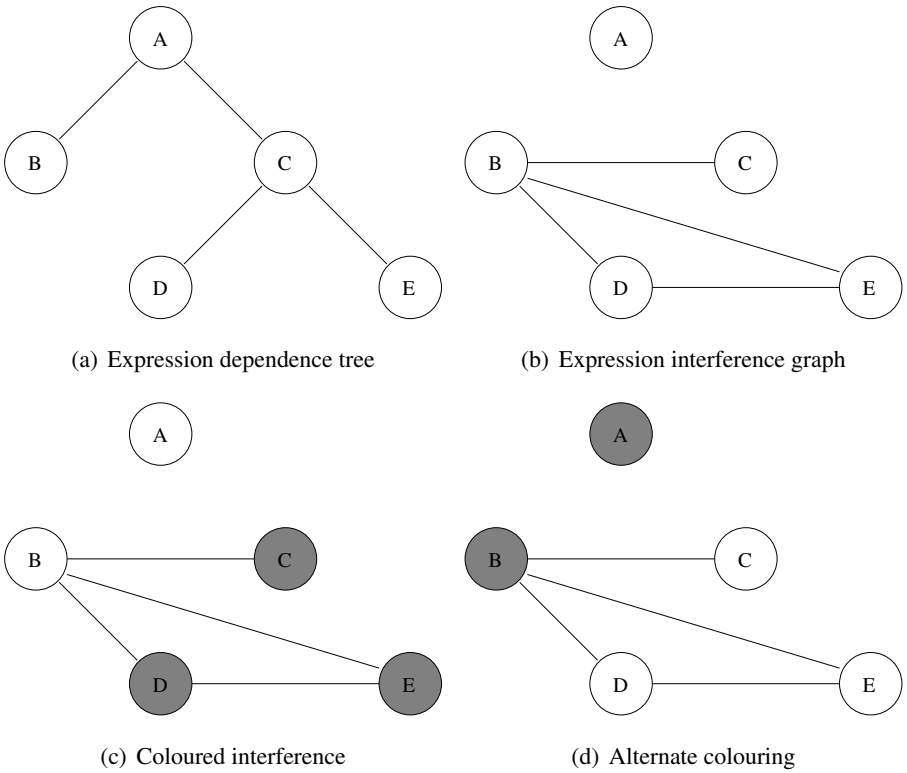
have been visited. The algorithm can be efficiently implemented and the main costs usually arise from the required pointer analysis phase.

"Aliasing" of different actual parameters from multiple call sites to a single set of formal function parameters are handled analogously.

6.3.2 Interference Model

To be able to effectively assign groups of variables to memory banks it is necessary to build an interference graph that represents the memory accesses in the program. This is done statically by taking the dataflow dependence graph for each expression and marking each pair of memory or variable accesses with no dependence between them as potentially interfering. This is demonstrated in figure 6.4, where figure 6.4(b) is constructed from figure 6.4(a). Nodes *B* and *C* interfere in figure 6.4(b) because there is no dependence between them in figure 6.4(a). Nodes *A* and *E* do not interfere because they are indirectly dependent, *E* must be executed before *A*, thus they can never be scheduled in the same cycle. When two nodes interfere (i.e. they are connected in 6.4(b)) it means that it is possible to schedule them in the same cycle, so it desirable that they access opposite memory banks.

Each of these potential interferences is given a weight that is equal to the estimated number of times that the expression will be executed. This estimate is determined by calculating each loop's iteration count (or using a value of 100 if the exact count cannot be statically determined), and assuming all branches are taken with 50% probability. The estimated call count for each function is also calculated this way by calculating how many times each call site is executed. The approximate information these static estimates provide is sufficient for determining which groups of variables are the most important, but using profiling information instead is listed as future work in section 6.9.2.



(a) Expression dependence tree (b) Expression interference graph

(c) Coloured interference (d) Alternate colouring

Figure 6.4: *Mapping dependencies to potential interferences.*

This variable interference graph is then reduced to a group interference graph using the assignments described in section 6.3.1. Every variable is a member of exactly one group, for every group all variables belonging to that group are collapsed down to a single node. Interference edges are merged and weights are summed. The result is a graph where each node represents all the interferences for a single group of variables.

This model is optionally extended to be able to handle variables that must be assigned to a fixed memory bank, e.g. *automatic* variables which must be placed on the stack or variables which the programmer has already assigned to a specific memory bank. The model is extended with a single additional node per fixed group of variables, if this group is the set of automatic variables it is named the *automatic node*. It is not possible to assign these nodes to a memory bank, they are always placed on a fixed one, but it is possible to interfere with them. It may, therefore, be possible to use these additional interferences to more accurately determine the assignments of other groups.

6.3.3 Partial Pre-assignments

This technique is fully compatible with manual pre-assignments, although no evaluation is performed using them. There are a few observations to make regarding how these fit in with

the rest of tools. The techniques in this chapter assign groups of variables to a memory bank whereas a programmer would assign individual variables. One way to avoid this mismatch is to perform the group forming described here, then have the programmer perform a partial pre-assignment of groups to memory banks before using the automatic assignment techniques described below. This would be safe, but is not compatible with library headers which may force function parameters to specific memory banks. It also means that the programmer can only perform assignment on an intermediate source-form.

A more practical approach is to support the pre-assignment of individual variables. This is easily handled by placing any group that contains a pre-assigned variable on the memory bank where the variable was assigned. This should work without issue, unless the programmer is able to make more accurate deductions about which variables must be placed on the same memory bank. The compiler is limited by the quality of the aliasing information and must make conservative choices, the programmer can be more imaginative.

If the programmer tries to place two variables from one group on different memory banks the compiler can no longer guarantee that the program will execute correctly. The potential for problems is entirely self-contained within this group though. The compiler can assign the rest of the groups and will not introduce any new errors, it just cannot guarantee that the programmer has not introduced an error. The automatic assignment may be sub-optimal though, the assignment algorithms all assume that a group can only be placed on one bank and use interference on a per-group basis rather than a per-variable basis.

6.4 ILP Colouring

6.4.1 Single Solution

A reference Integer Linear Programming (ILP) colouring approach that is approximately equivalent to the model by Leupers and Kotte [2001] is implemented. An ILP model is constructed from the interference graph, $I = (G, E)$ where G is the set of groups of variables (vertices in the graph) and E is the weighted interferences between them.

$$\forall g_i \in G: \quad X_i, Y_i = \begin{cases} 1, & \text{if } g_i \text{ is placed in bank } X/Y \\ 0, & \text{otherwise} \end{cases}$$
$$X_i + Y_i = 1$$

This ensures that each group g_i is placed in exactly one memory bank as if it was placed on neither or both then $X_i + Y_i$ would not equal 1.

$$\forall g_i, g_j \in G: \quad U_{ij} = \begin{cases} 1, & \text{if } X_i = Y_j \text{ or } Y_i = X_j \\ 0, & \text{otherwise} \end{cases}$$
$$U_{ij} \leq 2 - X_i - X_j$$
$$U_{ij} \leq 2 - Y_i - Y_j$$
$$U_{ij} \geq X_i - X_j$$
$$U_{ij} \geq Y_i - Y_j$$

The above constraints ensure that U_{ij} is set to 1 if and only if groups g_i and g_j are placed in different memory banks. The first two constraints ensure that $U_{ij} \leq 1$ if g_i and g_j are in different banks or 0 otherwise. The next two constraints set $U_{ij} \geq 1$ if g_i and g_j are on different banks or 0 otherwise. When combined these ensure that U_{ij} is always set correctly.

The linear program solver then optimises the following objective function while obeying the above constraints. As the values of the U_{ij} variables are set by the values of the X_i and Y_i variables it is only these latter variables that the solver may set to attempt to optimise the objective function. Thus it is effectively assigning variables to memory banks and attempting to maximise the available parallelism.

$$\max \left(\sum_{0}^{i} \sum_{0}^{j} U_{ij} \cdot W_{ij} \right) \text{ where } W_{ij} \in E$$

Here W_{ij} is the interference weight associated with each edge $(g_i, g_j) \in E$ calculated as described in section 6.3.2. If there is no edge between g_i and g_j in E then a weight of 0 is used. By maximising the above equation the linear programming solver is finding a set of assignments for the variables in G that favours placing variables with a high interference on different memory banks. This means that it should be possible to perform the most critical memory operations in parallel.

This set of assignments is not necessarily truly optimal though, it is only an optimal solution in terms of the interference graph. This ILP model is based on the model described by Leupers and Kotte [2001]. Their model used an interference graph built after the back-end of the compiler had run so it was a reasonably accurate model of the potential parallelism in the program. For this technique, however, the interference model is based on the program's source-code, the entire target compiler still has to be run on the program after variables have been assigned to memory banks. Thus this technique is re-evaluated in section 6.7 to ensure that it is still a valid colouring model at the high-level.

6.4.2 Multiple Solutions

Building the interference graph at a high-level also means that the problem is less constrained suggesting that there may be many optimal solutions to the ILP model above. For example, if

a node is completely disconnected in the interference graph then the score to be maximised by the linear solver will be the same whichever memory bank that group of variables is assigned to.

Integer linear program solvers generally work by first reducing as much of the program to a non-integer linear problem as possible, that can be solved quickly, and then using a branch-and-bound technique to solve what remains. If there are multiple optimal solutions then they may only be found during the branch-and-bound stage, where it is possible to continue searching even after an optimal solution has been found. In the process of reducing the integer problem to a non-integer one, however, many of the alternate optimal solutions may be lost and there will be fewer solutions for the branch-and-bound technique to find.

For the above memory bank assignment ILP model the branch-and-bound technique always only found a single optimal solution, as the problem reduced to a non-integer problem very well. This reduction is a crucial part of the ILP optimisation process and omitting it would make all but the most trivial problems intractable. Therefore, a different method is used to find multiple optimal solutions.

The alternative approach finds sets of nodes that may be inverted, where each node is a variable group. All the variable groups in a set are connected to at least one other node in the set (equation 6.1 below) and every node that is connected to a node in the set belongs to the set (equation 6.2. below).

$$\forall g_i \in S \subseteq G : \exists g_j \in S \text{ s.t. } (g_i, g_j) \in E \tag{6.1}$$

$$\forall g_i \notin S, \forall g_j \in S : \nexists (g_i, g_j) \in E \tag{6.2}$$

These sets have now been defined such that if an optimal solution to the ILP program is taken then the memory assignment of every node within a set may be flipped simultaneously to give a new solution that will still be optimal in terms of the ILP model. The exception to this is that any set that contains a node which is fixed to a specific memory bank is not invertible, e.g. if the *automatic node* belongs to the set. If $|S|$ is the number of sets in G that may be inverted then there are at least $2^{|S|}$ different possible optimal memory assignments for G.

As an example, consider the interference graph in figure 6.4(b). This would be split into two sets: $\{A\}$ and $\{B,C,D,E\}$. Assuming that all the edges have equal weight then figure 6.4(c) contains one possible optimal set of memory assignments. Some potential parallelism between accesses to D and E has been blocked due to them being assigned to the same memory bank, but as this graph is not two-colourable this is inevitable. There are three additional optimal assignments that can be found by inverting groups, assuming that none of the nodes are fixed to a specific memory. The first additional assignment can be found by flipping the assignment of $\{A\}$, the second by flipping the assignments of $\{B,C,D,E\}$ and the third by flipping both.

Figure 6.4(d) shows the assignment after flipping both sets.

This simple example also shows that the set of optimal solutions found by inverting sets is not necessarily the full set of optimal solutions. A representative subset of the optimal solutions may be simply and efficiently computed. In figure 6.4(b) there are six optimal ways of colouring $\{B,D,E\}$, C will always be the opposite of B meaning there are six optimal ways of colouring $\{B,C,D,E\}$. Combine this with the two ways of colouring $\{A\}$ and you have twelve different optimal solutions instead of the $2^2 = 4$ found by inverting sets. Though as the aim is not to find every possible optimal solution to the ILP program but only a representative set for evaluation this approximation is sufficient. In fact, not all $2^{|S|}$ colourings are taken as there would be too many possibilities for many benchmarks, instead just $|S|+1$ are used. Specifically, the colouring found by inverting all sets and then the colourings found by inverting each set individually.

6.5 Soft Colouring

As the previously described ILP assignment solution finds an optimal solution to an NP-hard problem it has exponential run-time. For small and simple problems the ILP solver is generally able to reduce most of the integer problem to a linear problem, this part can then by solved in polynomial time. For larger and more complex problems (or interference graphs) the reduction is less effective. This means that small changes to the interference graph can change its reducibility, resulting in a large increase in the time it takes to solve the model. Both the exponential run-time and the unpredictability of the solving time make the ILP assignment solution undesirable in many cases. A solution which finds good colourings quickly and with more predictable solving time would seem advantageous.

6.5.1 Single Solution

Graph colouring is well established within compilers, primarily for register allocation. Graph colouring for memory bank assignment is slightly different from graph colouring for register allocation. In register allocation the colouring is done under the ‘hard’ constraint that two interfering virtual registers must not be placed in the same physical register. Memory bank assignment operates under the ‘soft’ constraint such that it is preferred that two interfering variables to not be placed in the same memory bank. This maps directly the concept of “soft colouring” described in section 2.7.2.

Therefore conventional graph colouring approaches are unlikely to be adequate. Instead an algorithm designed for a distributed environment is used. Within this environment colouring problems frequently operate under soft constraints. Thus the distributed stochastic soft-colourer described by Fitzpatrick and Meertens [2001] was serialised for use in memory bank

assignment in algorithm 2.

Algorithm 2 *Soft Colouring(Variable Groups G)*

Require: An interference graph

Ensure: Locally-optimal memory assignment

1: **for all** $g_i \in G$ **do**
2: $C_i \leftarrow \text{rand}(\{0,1\})$
3: **while** G is still not a local optimum **do**
4: Determine C_i^{opt}
5: Inform central controller whether $C_i = C_i^{opt}$
6: With probability P: $C_i \leftarrow C_i^{opt}$
7: **end while**
8: **end for**

Here, C_i is the current colouring of g_i and C_i^{opt} is the current locally optimal colouring of g_i. The results of step 5 for all nodes allows the central controller to make a decision for step 3. If every node is already at an optimal colour then the algorithm terminates, as the colouring will no longer change. If any node still is not an optimal colour then it may change, which would then cause other nodes to change colour in the next iteration, so the loop continues to execute. It is also worth noting that in step 6 C_i may already be equal to C_i^{opt}. The implementation of this algorithm is limited to 10,000 iterations to ensure termination, but this limit was never reached for any of the benchmarks evaluated against.

Step 4 can be calculated using the equation below, where X_i, Y_i and W_{ij} are defined as for ILP in section 6.4.1 and g_i refers to the current node as in the above algorithm.

$$\begin{aligned} \forall e = (g_i, g_j) \in E : \\ costX &= \sum X_j \cdot W_{ij} \\ costY &= \sum Y_j \cdot W_{ij} \\ cost &= \min(costX, costY) \end{aligned}$$

Essentially this calculates the weighted value of how many of the neighbours of g_i are on memory banks X and Y and then picks the colour with the lowest value, i.e. the one with the fewest conflicts. Although this method of minimising conflicts is different from the ILP optimisation metric, where the potential parallelism is maximised, they are actually equivalent.

6.5.2 Changes To Interference Graph

In addition to using a different algorithm from ILP some changes to the interference graph $I = (G, E)$ are also required for soft colouring. The set of vertices G stays the same, but some additions are made to the set of weighted interferences E. Specifically, the graph is made to be fully and weakly connected by connecting every unconnected pairs of nodes in G with an edge with a very low weight. This weight is set low enough that it will always be lower than any weight relating to an actual detected interference.

This low weight means that these extra nodes never change a colouring decision between two interfering nodes. What it does do is provide a balancing metric for all unconnected nodes (such as $\{A\}$ in figure 6.4(b)) so that they will be roughly equally distributed between the X and Y memories.

6.5.3 Multiple Solutions

As there are stochastic elements in the soft colouring algorithm it is possible to get a range of colourings by repeated execution of the technique. Every set of assignments returned will be some local optimum. Note that to achieve different results for each run the pseudo-random number generator is provided with a different seed each time. In a production compiler, however, a constant seed could be used to ensure that multiple runs always produce identical binaries, otherwise debugging programs would be problematic.

6.6 Genetic Program Colouring

This section introduces a genetic programming approach to generating adaptive, yet highly effective colouring heuristics for the dual memory bank assignment problem. In a training phase a heuristic is constructed from a number of benchmark programs executed and profiled on real hardware. The generated light-weight heuristic is subsequently used to drive the graph colouring process, thus eliminating the need for potentially exponential ILP solvers in a production compiler. Genetic programming is described in general terms in section 2.6.

6.6.1 Single Solution

The full set of features used are described in table 6.1, this set of features is replicated for each of the X, Y and *unassigned* nodes. The full set of mathematical and logical operators available are described in table 6.2. The genetic programming library produces trees consisting of these nodes and it ensures that all nodes have the correct number of children. The only data-type is a floating point number. Because every function produced by the genetic programming library

Feature	Description
Parallel Interference	The no. of interferences at an immediate parallel level.
Para. Interfere. Accuracy	The estimated accuracy of the parallel interferences.
Expression Interference	The no. of interferences at an expression level.
Expr. Interfere. Accuracy	The estimated accuracy of the expression level.
Symbols: Aggregate	No. of aggregate symbols (e.g. structs) in group.
Symbols: Arrays	No. of array symbols in group.
Symbols: Pointers	No. of pointer symbols in group.
Symbols: Scalar	No. of scalar symbols in group.
Type: Integer	No. of integer symbols in group (e.g. an array of ints).
Type: Float	No. of floating point symbols in group.
Type: Complex	No. of non-numerical symbols in group (e.g. a void pointer).
Size	Total no. of bytes occupied by all variables in this group.
Size Accuracy	The estimated accuracy of the size of this group.

Table 6.1: *Program features available to a genetic program.*

is guaranteed to be valid, and every colour assignment possible is valid it is assured that every function produced will be accurately evaluated.

6.6.1.1 Training

Genetic programming is used off-line to create a heuristic that can be inserted into a compiler. It is required that the genetic programming library produces a function that will colour every node in an interference graph. It is unrealistic to expect genetic programming to produce a function that will return a complete graph colouring, so instead the graph is coloured one node at a time. A view of the graph is given from the perspective of the current node from the interference graph, as described in section 6.3.2. The three neighbour nodes are constructed (assigned to X, assigned to Y, not yet assigned). Then the function produced by the genetic programming library (which will be an entirely random tree initially) is run twice for each node, once saying if this node is assigned to X then what score would you give it, and the same for Y. This is done by mapping the X and Y nodes to the *Same Colour* and *Different Colour* nodes – and vice versa for testing a Y assignment. The node is assigned the colour with the higher score. For each function this process is repeated on every benchmark in the training data.

Once all the nodes on all the benchmarks have been assigned a colour the function can be assigned a fitness. A table of previously generated exhaustive results is used to look up the

Function	No. Inputs	Description
Add	2	$A + B$
Sub	2	$A - B$
Mul	2	$A * B$
Div	2	$A \div B$
Sqrt	1	$\sqrt{A}$ (returns 0.0 for negative inputs)
Abs	1	$\lvert A \rvert$
Min	2	$\min A, B$
Max	2	$\max A, B$
EQ	4	`if (A == B) { C } else { D }`
GE	4	`if (A >= B) { C } else { D }`
GT	4	`if (A > B) { C } else { D }`
LE	4	`if (A <= B) { C } else { D }`
LT	4	`if (A < B) { C } else { D }`
Const	0	`random` $(0 \leq X < 1000)$

Table 6.2: *Functions available to a genetic program.*

performance of this colouring, this is possible because the training happens off-line. If it is equivalent to the best possible colouring, the function is given a fitness of 0.0 (best possible). If it is equivalent to the worst colouring, it is given a fitness of 1.0 and results in-between are assigned a fitness proportionally. The overall fitness of a function is its average fitness across all benchmarks in the training data. Functions with a better fitness have a higher chance of being used in breeding and surviving into the next generation.

6.6.1.2 Deployment

The result of the training stage will be a single heuristic that can be used inside a compiler without any additional libraries. It will be a tree consisting purely of the features and functions described in tables 6.1 and 6.2 that operates on a graph view as described in the previous section. This means that the produced heuristic has an extremely low run-time overhead.

Figure 6.5 is a graphical representation of a heuristic produced by genetic programming. It can clearly be seen that is complete different from what a programmer might write. It was trained on all benchmarks but *fir-256_64* with extremely high parsimony, to keep the function small for readability, the results presented in section 6.8 use larger functions than this. Clearly this heuristic is trivial to evaluate at run-time, it only contains two `if`-statements but it is able to find the optimal solution for *fir-256_64*. It seems to "work" because if there are a large

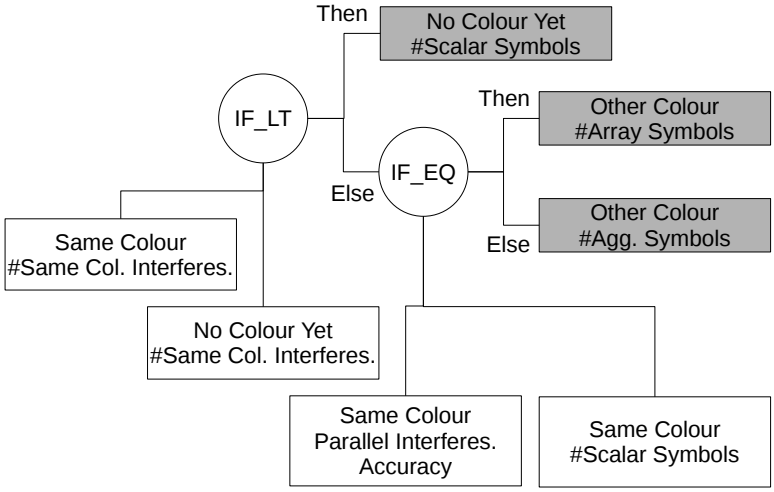


Figure 6.5: *An example heuristic generated by genetic programming (trained on all benchmarks but* fir-256_64 *with extremely high parsimony). The white boxes are input values, the circles are operations and the dark gray boxes are values that may be returned.*

number of interferences for uncoloured nodes it prioritises based on the number of uncoloured symbols. Otherwise if there are already a large number of symbols of "this" colour it prioritises for the opposite colour. This is an example of a heuristic that is logical but it is unlikely that a programmer would ever choose to write it this way.

6.6.1.3 Variations

To try and improve the performance of the functions produced by genetic programming, three modifications were attempted. The modifications are described here, their effects are described in section 6.7.

Firstly, the way the evolved program was changed to consider the order in which the nodes are coloured, two variations were attempted. In the first variation the nodes were sorted according to the sum of their interferences, so the most critical nodes are coloured first. In the second variation, instead of just running the generated function on one node it was run on every uncoloured node in the graph. The node with the highest score overall score was assigned a colour, effectively letting the generated function pick the order in which to colour the nodes. This affects the time it takes to perform the colouring, for a graph with n nodes it takes $O(n)$ time to produce a graph view for a given node. With pre-ordered nodes $O(n)$ nodes are evaluated resulting in a colouring time of $O(n^2)$. For a self-ordering method $O(n^2)$ nodes are evaluated resulting in a colouring time of $O(n^3)$.

Secondly, the way that the evolved program was evaluated was changed to penalise larger

programs by using a selection method that encourages parsimony, specifically Ratio Bucket Tournament Selection, described by Luke and Panait [2002]. The aim was to stop huge functions highly specialised to the training data from being generated.

Thirdly, the amount of potentially extraneous information available to the evolved program was reduced by attempting to perform the assignment without using the *unassigned* node. The idea behind this was that nodes that are not yet assigned to a memory bank do not affect immediate assignment decisions – it was not clear if the functions would be able to use the information to "plan ahead" or if it was just noise.

6.6.2 Multiple Solutions

This technique does not produce multiple solutions, as the heuristics produced are simple trees with deterministic behaviour. Although a heuristic may assign the same score to multiple choices, this is resolved in a deterministic manner that was implicitly included in the heuristic during training.

6.7 Evaluation Methodology

All the speed-ups presented in section 6.8 use the compilers default assignment as a baseline (i.e. a 1.0x speed-up). This default assignment is to place all data on a single memory bank. The speed-ups are calculated using the target processors internal cycle counters. For all benchmarks in this chapter all I/O happens at the beginning and end of the program so the cycle counts are simply recorded after the initial data is loaded and just before the results are saved.

6.7.1 Platform and Benchmarks

The source-level C to DSP-C compiler was implemented using the SUIF compiler framework [Wilson et al., 1994]. The C program is converted into the SUIF intermediate format which is then annotated with aliasing information using the SPAN tool [Rugina and Rinard, 1999]. This information is used to form groups of variables as described in section 6.3.1 and output DSP-C with group identifiers in place of memory qualifiers. The C preprocessor may be used to assign a group of variables to a specific memory bank according to the generated group to memory bank mapping.

Both the ILP colourer and the GP colourer are implemented in Java. The ILP colourer makes use of the *lp_solve* [LPS, 2010] library, which is implemented as a native binary, with the default pre-solve and optimisation settings. The GP colourer uses the *ECJ* [ECJ, 2008] package to evolve and execute genetic programs. The evolutionary settings used were ECJ's default Koza [1992] parameters (90% chance of breeding, 10% chance of mutation, tournament

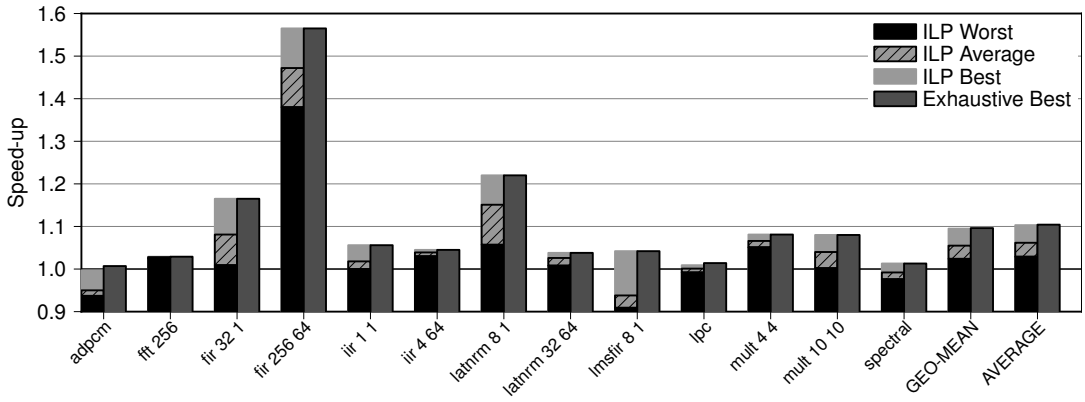


Figure 6.6: *A comparison of the range of ILP solutions against the true optimum.*

size of 7), with a population of 1024 and 50 generations. Additionally a small amount of elitism is used, the best 2 functions from each generation always survive into the next.

The colourings were done on a Linux system with two dual-core 3.0GHz Intel Xeon processors and 4GB of memory. The experiments were run on an Analog Devices TigerSHARC TS-101 DSP operating with a clock of 300MHz and the DSP-C programs were compiled using the Analog Devices VisualDSP++ compiler. The technique was evaluated using the UTDSP benchmark suite [Lee, 1998] (the arrays versions of each benchmark, as described in section 2.8). Each colouring was only run once as the TigerSHARC's static pipeline and lack of cache result in deterministic hardware.

6.7.2 Evaluating Genetic Programming

The benchmarks were evaluated using leave-one-out cross-validation, a standard statistical technique. For each of the 13 benchmarks used for evaluation the GP colourer was trained on the other 12 benchmarks and then tested against the 13th. This ensures the results are representative of how the colourer will perform on an unseen program.

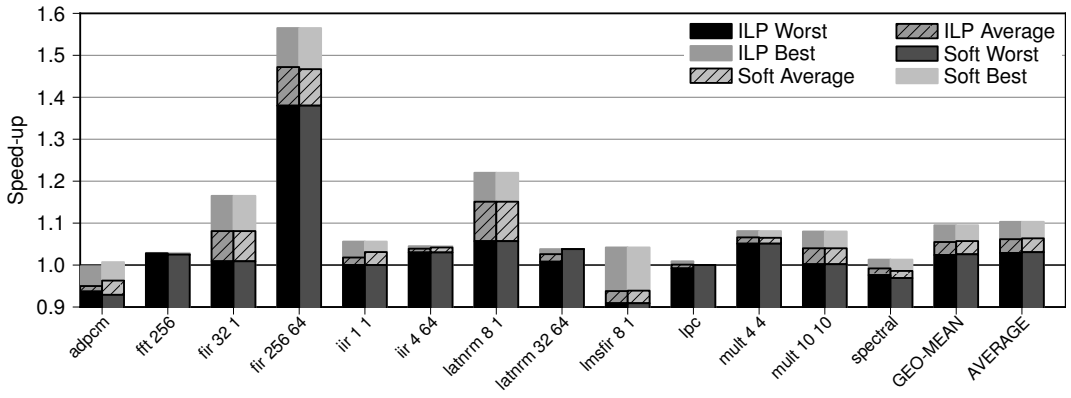
This is essential for performing useful evaluation of any machine learning technique in a compiler. If the technique is evaluated using the same benchmarks as those used for training purposes, it will generally perform well, but no claims can be made about how it perform on future programs. This is because during the training phase thousands of variations will have been evaluated and if these are reused for evaluation the results will represent a brute-force search. Leave-one-out-cross-validation allows every benchmark to be used for evaluation while never tainting the results with training data.

6.8 Results

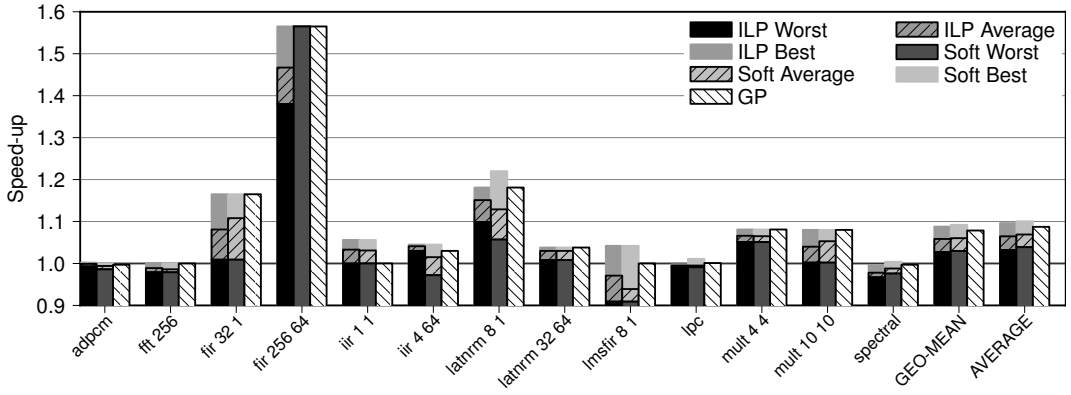
Initially, the effectiveness of the colourings provided by ILP were evaluated against an exhaustive set of results. These exhaustive results were obtained by running every possible colouring for each benchmark. The 'ILP Best' and 'ILP Worst' bars in figure 6.6 correspond to the highest and lowest speed-ups, relative to the performance of ISO C, in the set of equivalent ILP solutions found per benchmark, as described in section 6.4.2. The 'ILP Average' bars represent the average speed-up of these sets. It can be seen that most of the benchmarks cover a significant range of results, and *lmsfir-8_1*, *lpc* and *spectral* see performance improvements for some ILP colourings but degradations for others. Another observation is that for all benchmarks, except *adpcm*, the ILP colourer was able to find the optimal solution. On average the range of speed-ups due to ILP was between 3.0% and 10.3% with an overall average of 6.2%, compared to 10.4% for the best of the exhaustive results.

It can be noted that the true optimum results seen in figure 6.6 are nowhere near the 2.0 speed-up that you might expect from doubling memory bandwidth in a signal processing application. In earlier work on low-level techniques, (such as the work by Saghir et al. [1996], Gréwal et al. [2003], Leupers and Kotte [2001], Ko and Bhattacharyya [2003] or Gréwal et al. [2006b]), speed-up figures are reported for individual compute loops or kernels only. In these papers, asymptotic speed-ups approach the theoretical limit of 2 for loops with large iteration counts, very small loop bodies and parallelisable data accesses. In this chapter, however, the evaluation is performed using whole applications that include sequential sections that do not benefit from simultaneous data accesses. According to Amdahl's law this leads to smaller, but more realistic, overall speed-up figures. For the same small loops from the same benchmarks, though, the source-level approach described here matches the reported performance of the low-level techniques very accurately, while providing a portable approach and avoiding modifications to the back-end compiler.

After this, the soft colouring technique was compared against the baseline ILP colourer. The results of this are shown in figure 6.7(a), where the ranges of both the ILP and the soft colouring results are shown. Here it can be seen that despite not being guaranteed to solve the colouring model optimally, soft colouring does just as well as the ILP colourer, finding almost exactly the same range of results. The range is still quite wide, however, therefore to attempt to constrain the assignments an additional *automatic node*, described in section 6.3.2, was added. The effects of this are shown in the ILP and soft colouring columns of figure 6.7(b). The *automatic node* does shorten the range of solutions for both ILP and soft colouring, but not in the same way. For ILP colouring mostly good results are eliminated (going from 3.0%-10.3% to 3.2%-9.6% on average), for soft colouring mostly bad results are eliminated (going from 3.1%-10.3% to 3.9%-10.1% on average). Also, the *automatic node* allows soft colouring to always find the truly optimal solution for *fir-256_64* and to find better solutions than the



(a) A comparison of the range of ILP solutions against the range of soft colouring solutions without an *automatic node*. The left bar of each benchmark is the ILP result and the right is soft colouring.

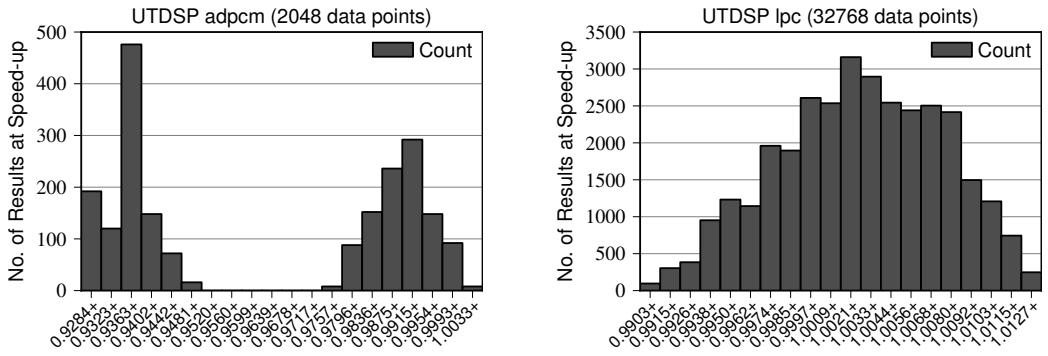


(b) Figure 6.7(a) with an additional *automatic node* and genetic programming results from the best genetic programming mode at the far right of each benchmark.

Figure 6.7: *A comparison of different techniques. The maximum available performance for each benchmark is the same as the "Exhaustive Best" in figure 6.6.*

ILP colourer for *lpc* and *spectral*. These result in the average performance of the soft colourer being slightly higher than that of the ILP colourer.

The GP colourer was then compared against both the ILP and soft colourers. Figure 6.7(b) shows the speed-up achieved by the best GP colourer (nodes may be coloured in any order, constrained size of genetic function and has access to information on uncoloured nodes) against the range of ILP and soft colouring results. All results in figure 6.7(b) make use of an *automatic node*. The average speed-up achieved by the GP colourer is 8.7%, which although not as good as the upper range of the ILP colourer's potential, it is much higher than the lower end. The GP colourer achieves 78.3% or 85.9% of the performance available in ILP's range of potential performance for without and with an *automatic node* respectively. Equivalently it achieves 77.7% or 77.4% of the potential performance compared to soft colouring. This is significant



(a) Island distribution

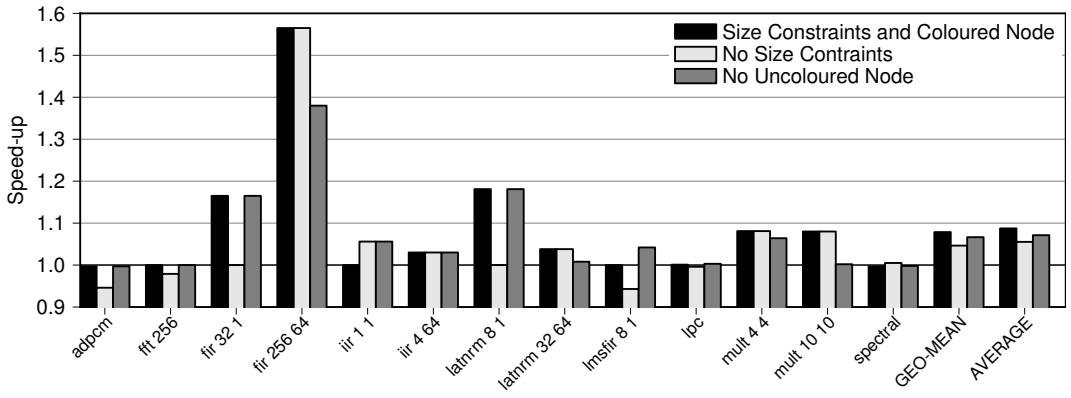
(b) Normal distribution

Figure 6.8: *Distributions of speed-ups. Note: the charts for the rest of the benchmarks are in figure B.43 on page 240.*

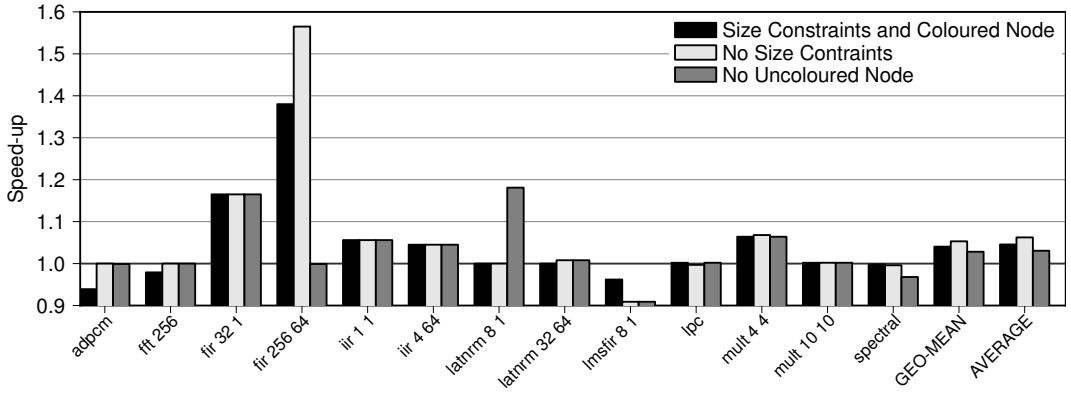
because it means that given the generally uniform distribution of ILP performance across its range of potential results (see below), the GP colourer will out-perform the ILP method without an *automatic node* 78.3% of the time. This can be confirmed by noting that the average GP performance in figure 6.7(b) is significantly higher than the average ILP and soft-colouring result. Other points of note are that the GP colourer only results in a slow-down for two benchmarks (*lpc* and *spectral*), whereas the ILP colourer may result in slow-downs for five benchmarks.

It is assumed above that the spread of ILP performance is generally uniform between the worst and best ILP results. This general trend is demonstrated by the average speed-up across a set of ILP solutions generally being equidistant between the best and worst solutions. Figure 6.8 (a subset of figure B.43 on page 240) shows that the details are somewhat more complicated. The charts in figure B.43 show the distribution of speed-ups for each benchmark. Most of the benchmarks exist in small islands of results which achieve similar results with large gaps to the neighbouring island (see figure 6.8(a)). The only absolute exception to this is UTDSP *lpc* (see figure 6.8(b)) for which the speed-up distribution looks like a "normal distribution". The assumption of uniform distribution is not unreasonable when viewed across the entire set of benchmarks.

In the process of developing the GP colourer, different methods were evaluated. Here the three variations described in section 6.6 are compared to the method that was found to be best. Firstly, self-ordered vs pre-ordered nodes results are presented in figures 6.9(a) and 6.9(b) respectively. The general trend is that letting the GP colour the nodes in any order is almost always better than arranging the order beforehand. If keeping the other variants fixed then average speed-ups of 8.7% and 4.5% are achieved respectively, but if parsimony is abandoned



(a) Self-ordering GP.



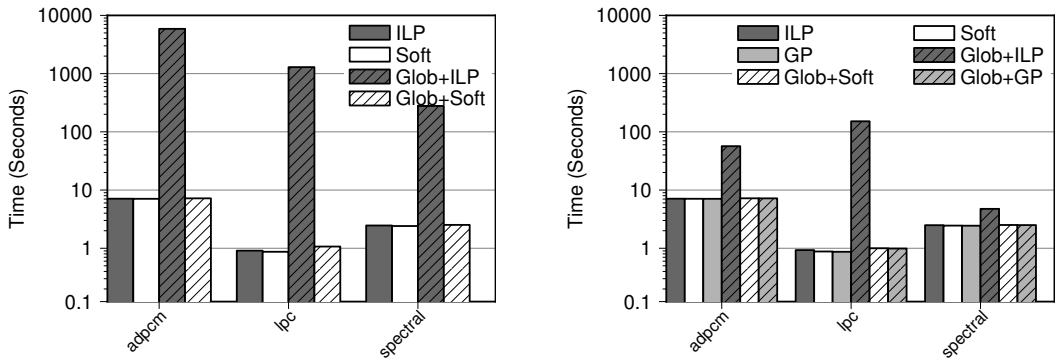
(b) Pre-ordered GP.

Figure 6.9: *A comparison of the different modes of operation for the GP.*

then the trend reverses with 5.5% and 6.2% speed-ups respectively. This is possibly due to the additional complexity of self-ordered nodes making the problem susceptible to over-fitting, i.e. the heuristic becomes too specialised to the training benchmarks and does not work as well on the test benchmark.

Secondly, the fitness of larger functions were penalised. It was found that this penalty significantly improved performance with self-ordered nodes, the effects of eliminating it may be seen in the first and second columns of each benchmark in figures 6.9(a) and 6.9(b).

Thirdly, the uncoloured node was eliminated from the reduced interference graph (see section 6.6.1.1). The effects of this may be seen by comparing the first and third columns of each benchmark in figures 6.9(a) and 6.9(b). In most cases this made little difference. A few benchmarks, however, suffered without this node so eliminating it slightly reduces the performance of the colourer on average.



(a) Time taken to perform memory assignment.

(b) Timings with an additional *automatic node*.

Figure 6.10: *Timings for each technique on the largest benchmarks. The Genetic Programming technique always uses the additional* automatic node *so is only included in figure (b).* Note: the full versions of these charts are figures B.44 and B.45 on page 242.

6.8.1 Scalability

The final consideration is how long it takes to execute each of the colouring algorithms. Figure 6.10(a) shows the time taken by both the ILP and soft colourers for the largest of the UTDSP benchmarks. The remaining benchmarks all had total colouring times of under a second. The GP colourer is not included in this graph as this graph is for results without an *automatic node*. The time reported is the time taken to perform alias analysis and then to execute the colouring algorithm. The alias analysis takes a notable amount of time for these benchmarks, up to 7 seconds for *adpcm*. The *Glob* timings are the time it takes to perform colouring if as many automatic variables as possible are made global (any declared in non-recursive functions). This is an artificial change that always results in slower code, however it allows results to be obtained for larger interference graphs. It is difficult to run larger programs on the target architecture as it has limited program memory available. Globalisation roughly doubles or quadruples the number of nodes in the interference graph. The larger interference graph due to globalisation exposes the dangers of the ILP solvers optimisation strategies and exponential run-time. Although for the non-globalised code ILP and soft colouring take roughly the same time, for the globalised code the soft colourer is several orders of magnitude faster. Most notably for the globalised *adpcm* the soft colourer takes 7.2 seconds, but the ILP colourer takes over one and half hours (5873 seconds). Figure 6.10(b) shows the effect of adding the additional *automatic node* to the interference graph has on the time it takes to do the colouring. It also includes the colouring time for the GP colourer. There is little change for soft colouring, but it has a dramatic affect on the ILP colourer for globalised code, *adpcm* is now coloured in under a minute

(57 seconds). This possibly means that this single extra node allowed a larger part of the integer problem to be reduced to a linear problem, demonstrating the fragility of ILP colouring. Additionally, it can be seen that the GP colourer is just as fast as the soft-colourer.

6.9 Summary and Conclusions

6.9.1 Critical Evaluation

The most effective colouring technique in this chapter was the genetic programming colourer. Due to the training approach, however, it needs exhaustive data on the performance of extra possible colouring of its training benchmarks. This limits it to being trained on small benchmarks, the biggest in this chapter had 15 variable groupings results in 32,768 different colourings. This mode of operation is quite common with machine learning techniques though, training on small programs should still be effective for operating on larger programs. This claim is not verified however, that is left as future work since running larger benchmarks is not currently possible.

The primary limitation of this technique is that it assumes that all data will fit into the scratchpad memories. There exist mature techniques for allowing large programs to successfully exploit small scratchpad memories [Verma and Marwedel, 2007] and the use of these techniques should be orthogonal to what is presented in this chapter. None of the related work evaluated in section 3.2 discusses integration with techniques for managing scratchpad memories either. They also all use small benchmarks for evaluation, the DSPstone suite is the most popular. Some of the related work use synthetic benchmarks for evaluating scalability. These are randomly produced “large” programs that will still fit in small memories. The work in this chapter improves on this by producing semi-synthetic benchmarks but artificially modifying genuine programs.

Finally, one key technology that has not been considered in this chapter is dual-port memories (as opposed to dual memory banks that are the topic of this chapter). Dual memory banks have two memory banks which handle one access each, dual-port memories have one memory bank which can handle two accesses simultaneously. This completely eliminates the need for partitioning data while still providing the same speed-ups. This does not reduce the usefulness of the work in this chapter, however, as dual-port memories are about twice the size of dual memory banks for the same amount of memory space. Thus for a fixed hardware budget dual-port memories halves the amount of scratchpad memory that is available. Thus, if compiler support allows dual memory banks to behave like a dual-port memory most of the time it seems unlikely that dual-port memory would be the best choice for a system designer.

6.9.2 Future Work

There are two obvious areas of that the work in this chapter may be extended. The first has already been mentioned in section 6.3.2: the static estimates currently used to assign weights to the interference graph could be replaced with profiling data. The increased accuracy would result in the interference model more closely matching reality, which should improve the quality of the assignment decisions. None of the related work in section 3.2 uses profiling data, however, and there may be a reason for that. Using the exhaustive results it can be seen that the upper range of ILP and soft-colouring results are already extremely close to the true optimum. Thus it is not clear if a more refined profile would actually improve these results.

The second step would be to try and replace the genetic programming colourer with a similar one based around a support vector machine (SVM). The form of the problem as it is presented to the genetic program would also be suitable for presentation to an SVM. SVMs are a more recent development in machine learning than genetic programming and the current literature suggests they are far better at classification. In this regard they are slightly different to GPs in their purpose, GPs assign a score, SVMs classify. Therefore, the GP colourer described above can "choose" what order to assign variable groups to memory banks, and increase the speed-up achieved by doing so. An SVM-based colourer, however, would have to be provided an ordering as it could only classify a variable group as requiring placement on bank *X* or *Y*. As there is no scoring there is no way to order variable groups for assignment.

Finally, section 8.4 mentions that if *EnCore* has support for XY memory implemented the techniques described here could be applied. This would allow a direct evaluation of combined extension instructions and dual memory banks.

6.9.3 Summary

This chapter described a method for performing dual memory bank assignment at the source-level, using a C to DSP-C compiler. The technique was evaluated on the UTDSP benchmark suite where a 10.3% speed-up was achieved on average, which is extremely close to the true optimum, so these techniques should be competitive with hand-colouring. This and the fact that this scheme accepts and completes partial manual pre-assignments makes it seem likely that this technique would be effective in an industrial scenario.

It has been demonstrated that with the addition of an *automatic node* results in slightly better average speed-ups with the soft colourer (6.9%) than the "optimal" ILP colourer (6.5%) and the genetic programming colourer was able to do significantly better (8.7%).

The described technique may be easily introduced to an existing DSP toolchain due to operating at the source-level. The soft colouring technique is fast enough to be used with large programs while still producing excellent results. Alternatively, the ability to train the genetic

programming technique offline means that a large suite of benchmarks could be used to train the tool. This results in performance that out-performs the “optimal” ILP colourer 78.3% of the time with an extremely low execution cost.

Chapter 7

Code Transformation and Instruction Set Extension

"A worker may be the hammer's master, but the hammer still prevails. A tool knows exactly how it is meant to be handled, while the user of the tool can only have an approximate idea."
— Milan Kundera, *writer*, 1929–.

Chapters 4 and 5 looked at different ways of enabling the compiler to use extension instructions and improving their effectiveness. This chapter investigates how the compiler can be used to improve the quality of the extension instructions that are generated. This is done by applying source-to-source transformations to the target application *before* automated instruction set extension (AISE) is performed. A large set of transformation sequences are used on a benchmark suite to perform design space exploration over this area. It is demonstrated that the selection of "good" instruction templates is strongly dependent on the shape of the C code presented to the AISE tool. A methodology is proposed that combines the exploration of high and mid-level program transformations and low-level instruction templates.

There are three additional points worth noting concerning the scope of this chapter. Firstly, while compiler-based transformations could also be used to improve the effectiveness of an extension instruction mapper this is not considered, except as future work. Secondly, although the AISE performed in chapters 4 and 5 does not consider compiler transformations as a design space, a static set of transformations are performed by the compiler before both AISE and extension instruction mapping. The set of transformations used in those chapters is the entirety of the GCC "middle-end" at the `-O2` setting. The primary benefit of these from the perspective of the PASTA toolchain is to normalise code and eliminate redundancy. Thirdly, this chapter considers the code-size benefits of both applying transformations and performing AISE. Previous chapters did not discuss the code-size benefits of AISE at all because without transformations they are not very significant, sometimes a program will shrink by 1-2%, sometimes register allocation overhead will cause it to grow by 1-2%.

Disclaimer. This chapter is based on experiments already described in a paper [Murray et al., 2009], which was in turn an extension of another paper [Bennett, Murray, Franke, and Topham, 2007]. The author of this thesis was the lead author of the paper describing the experiments presented in this chapter [Murray et al., 2009], but there was also an additional non-supervisory author: Richard Bennett. His role, however, primarily focused on the AISE tools. The experiments described in this chapter were designed by the thesis author, but used some tools developed by others, as with previous chapters.

7.1 Limitations of Methodology

Although this chapter is placed at the end at the thesis it actually describes work which was undertaken before any of the work in chapters 4–6. Therefore the tools used in this chapter are actually the predecessors of the ones used in the previous chapters. This means that they are not as capable, as they were built by a small number of people in a short amount of time. The full toolchain used in previous chapters was developed by a team of people over several years.

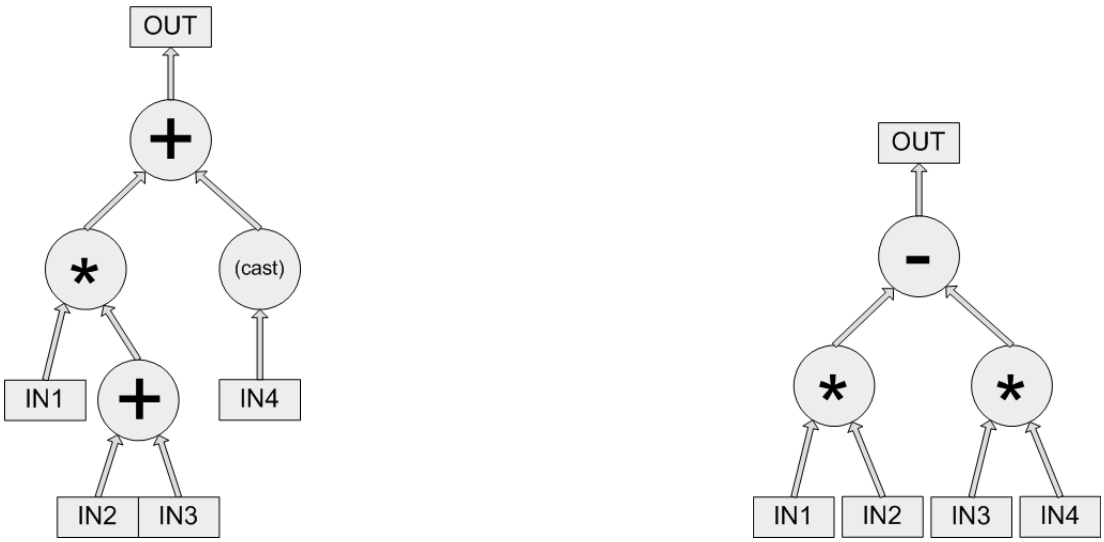
This does not mean that the results in this chapter are unusable, they still lead to interesting conclusions. This section, therefore, will discuss the limitations of the tools and techniques used in this chapter so as the rest of the chapter can focus purely on methodology and results. The critical evaluation (section 7.6.1) will evaluate the experiments, not the tools. The future work section (section 7.6.2) will describe several ways that these experiments can be enhanced using the full PASTA toolchain, but will not merely state that these experiments should be re-run as the results already obtained are still valid.

Chapter 5 found that *ISEGen* was probably off in its acceleration estimates by a factor of 2x, the AISE tool in this chapter uses a similar performance model, so it could also be off.

The AISE tool used in this chapter is not nearly as advanced as the one used in chapters 4 and 5. Previously, it had only been evaluated on the SNURT suite, using UTDSP was already requiring the tool to process larger programs than it had ever dealt with before. It is based around an integer linear program solver which does not scale as well as *ISEGen*.

7.2 Motivating Example

As an example consider the code excerpt in figure 7.2. The function `fft` is the core kernel of the UTDSP *fft* benchmark and implements a fast fourier transform. The key features of this code are that it has loops nested three levels deep and a significant amount of the code is spent on performing complex address calculations. Presented with this baseline code, the Atasu AISE tool (see section 2.3.1) generates extension instructions which result in a 7.1% performance improvement.



(a) Instruction 1 (b) Instruction 2

Figure 7.1: *Complex instruction templates generated for the transformed FFT code in figure 7.3.*

In figure 7.3 the main differences due to source-level transformation of the code in figure 7.2 are shown. While the code is functionally equivalent, it outperforms the code in figure 7.2 by a factor of 1.31x. The transformed code has had *loop-invariant hoisting* applied followed by *common sub-expression elimination*. Most of the code in an FFT is related to address calculation, and the expressions that get hoisted and eliminated are all array index calculations. Having moved these array index calculations to a common place the AISE algorithm was then able to generate complex address calculation instructions, such as those shown in figure 7.1. Commercial digital signal processors hand-designed by engineers often contain specialised addressing modes for FFT calculations, so it is interesting to note that AISE constructs specialised FFT address calculation instructions.

Running the AISE tool on the transformed code in figure 7.3 results in a further 31% improvement (over the transformed code), or a total combined speed-up of 1.51x over the baseline. Only a certain part of the performance gain can be directly attributed to code transformations, the rest is due to the enabling effect of the source-level transformations on AISE. So it can be seen that by transforming the code it is not only possible to get improved performance on a baseline processor, but the gains that extension instructions provide can be increased as well – from 7.1% to 31% in this case.

This short example demonstrates the interaction between the two techniques that makes it difficult to predict the best source-level transformation sequence for a given application when instruction set extension will be performed. Combined exploration of both the software and

```
void fft(float *data_real, float *data_imag, float *coef_real, float *coef_imag) {
    /* Variable declarations. */
    groupsPerStage = 1;   buttersPerGrp = 512;
    for (i = 0; i < 10; i++) {
        for (j = 0; j < groupsPerStage; j++) {
            Wr = coef_real[(1 << (unsigned int)i) - 1 + j];
            Wi = coef_imag[(1 << (unsigned int)i) - 1 + j];
            for (k = 0; k < buttersPerGrp; k++) {
                temp_real =   Wr*data_real[2*j*buttersPerGrp+buttersPerGrp+k]
                            - Wi*data_imag[2*j*buttersPerGrp+buttersPerGrp+k];
                temp_imag =   Wi*data_real[2*j*buttersPerGrp+buttersPerGrp+k]
                            + Wr*data_imag[2*j*buttersPerGrp+buttersPerGrp+k];
                data_real[2*j*buttersPerGrp+buttersPerGrp+k] =
                    data_real[2*j*buttersPerGrp+k] - temp_real;
                tmp = &data_real[2*j*buttersPerGrp+ k];
                *tmp = *tmp + temp_real;
                data_imag[2*j*buttersPerGrp+buttersPerGrp+k] =
                    data_imag[2*j*buttersPerGrp+k] - temp_imag;
                tmp2 = &data_imag[2*j*buttersPerGrp+k];
                *tmp2 = *tmp2 + temp_imag;
            }
        }
        groupsPerStage = groupsPerStage<<1u; buttersPerGrp = buttersPerGrp>>1u;
    }
    return;
}
```

Figure 7.2: *Original UTDSP* fft *implementation*

hardware design spaces generates a significantly better solution than isolated optimisation approaches could produce. This chapter contains an empirical evaluation of this hardware/software design space interaction and shows that significant performance improvements can be achieved by exploring the combined optimisation space.

7.2.1 Combined Design-Space

The combined design space in question here is that of transformation and AISE, with the intention of demonstrating that there is promise for automated techniques to manage the design in such a large space. It is also important that the results of a cooperative automated framework can outweigh the sum of their separated components.

The hope is, as with compilers, that the actual efforts of the search of the combined space can remain a phased searching of each space individually. The most important factors in this scenario are the accuracy and detail of the modelling employed in any decision making. This work attempts to contribute to the understanding of which transformations will need to be made

```
void fft(float *data_real, float *data_imag, float *coef_real, float *coef_imag) {
    /* Variable declarations. */
    groupsPerStage = 1;   buttersPerGrp = 512;
    for (i = 0; i < 10; i++) {    /*      Loop runs at least once */
        if (0 < groupsPerStage) { /* <--- invariant hoisting.     */
            tmp = (float (*)[])coef_real;       /* Loop invariant hoisted   */
            tmp0 = (1 << (unsigned int)i) - 1;  /* variables.               */
            tmp1 = (float (*)[])coef_imag;      /*                          */
            for (j = 0; j < groupsPerStage; j++) {
                Wr = ((float *)tmp)[tmp0 + j]; Wi = ((float *)tmp1)[tmp0 + j];
                if (0 < buttersPerGrp) { /* As with previous loop guard. */
                    tmp3 = (float (*)[])data_real;
                    tmp4 = 2 * j * buttersPerGrp;  /* Common sub-expression   */
                    tmp5 = tmp4 + buttersPerGrp;   /* elimination temporaries.*/
                    tmp6 = (float (*)[])data_imag;
                    for (k = 0; k < buttersPerGrp; k++) {
                        temp_real=  Wr*((float*)tmp3)[tmp5+k]
                                  - Wi*((float*)tmp6)[tmp5+k];
                        temp_imag=  Wi*((float*)tmp3)[tmp5+k]
                                  + Wr*((float*)tmp6)[tmp5+k];
                        ((float *)tmp3)[tmp5+k]=((float *)tmp3)[tmp4+k]-temp_real;
                        tmp = &((float *)tmp3)[tmp4+k];
                        *tmp = *tmp + temp_real;
                        ((float *)tmp6)[tmp5+k]=((float *)tmp6)[tmp4+k]-temp_imag;
                        tmp2 = &((float *)tmp6)[tmp4+k];
                        *tmp2 = *tmp2 + temp_imag;
        }}}}
        groupsPerStage = groupsPerStage<<1u; buttersPerGrp = buttersPerGrp>>1u;
    }
    return;
}
```

Figure 7.3: *UTDSP* fft *implementation after application of the source-level transformations that resulted in the best combined performance.*

extension-aware, and which are beneficial under extension.

The use of compiler transformations when developing automated design space exploration must be very carefully considered, so as not to disturb the context in which design-space decisions are made. Transformations which are run prior to an AISE tool must be re-applied with the same parameters to the areas in which the tool identified mappings. Otherwise, the AISE will not find the same mapping in code-generation without having the areas explicitly defined by manual means.

7.3 Experiment Methodology

The primary concern of these experiments was to determine which transformations or combinations thereof infer the greatest execution speed improvement from AISE. Secondly, the experiments were to find limits of performance gain and loss from the combined design space defined by transformation and AISE over a baseline design employing neither.

To create the transformation design space in these experiments, a source-to-source transformation tool built upon the SUIF1 [Wilson et al., 1994] compiler framework is used. It is worth noting that this tool operates as a transformation tool rather than an optimiser, so when a transformation is used it is applied everywhere that it is legal to do so, without any analysis of whether it is likely to be beneficial. The tool generates large volumes of transformed source code samples rapidly from a definition of:

1. **The source code, in C**. A variety of single-function benchmarks are tested, as well as a few larger applications.

2. **The Transformation Space Definition**. A list of transformations to use and the minimum and maximum transformation sequence length permitted. The tool supports a wide array of source-to-source transformations to be used in the exploration (see appendix A). Some of the transformations can be considered high-level transformations (e.g. *loop unrolling*), others are commonly classified as generic platform-independent transformations (e.g. *common sub-expression elimination*).

3. **The number of samples** to take from the transformation space, and hence the number of transformed source codes to produce. This can be set to either a fixed limit or be unbounded, for example, to exhaustively enumerate the entire transformation space (up to a given sequence length).

This framework was used to conduct two experiments. The first experiment investigated the scope of improvements available by combining source-level transformations and instruction set extensions. Samples were taken with uniform probability at random points across the entire space of potential transformations. A sample in this sense represents a single point in the transformation space, and results in the ordered set of transformations selected at that sample point to be applied to the code. In the second experiment, all transformation sequences of a reduced set of transformations up to length three were exhaustively enumerated. Here the 15 most relevant transformations have been selected based on an analysis of the first round of experiments, thus keeping this experiment within practical limits.

The benchmarks used in this experiment were taken from the UTDSP [Lee, 1998] and SNURT [Seoul National University - Real-Time Research Group, 2008] suites. These two suites were chosen because most of the benchmarks contained are small in size. This was

desirable, in part, because of the limitations described in section 7.1, but also because it allows for easier evaluation of which transformations are most effective. If large benchmarks are used, many transformations will be beneficial for one part and detrimental for another. With smaller benchmarks this is less of a problem.

7.3.1 Selection of Transformations

The algorithm controlling the source-level transformation of the input program for the first experiment is a simple probabilistic algorithm. It generates transformation sequences of a random length (up to a given maximum) and selects a transformation for inclusion in the sequence with a uniform probability.

For the second experiment all permutations without repetitions are generated. Skipping sequences with repeated transformations may result in beneficial sequences being missed but removing them allows for clearer analysis of what effect each transformation has individually. Duplicates due to "non-effective" transformations are filtered out by checking to see if the form of the intermediate representation (IR) has changed after each application. Once this reduced set has been generated, the program is compiled and run on an x86 platform and has its outputs compared to the original reference. If it does not match then the sequence is discarded. In the case of floating-point benchmarks a total of 1% of bits are allowed to be flipped while still considering the output identical to allow for small variations in results due to moving code around. The output must be checked to make sure the code has not been accidentally damaged by an incorrect SUIF transformation pass. Many of the sequences considered are unusual in conventional terms and using them uncovers latent compiler bugs. Of the theoretical 97,548 permutations (across 33 benchmarks with 15 transformations in sequences of up to length three with no repeated applications within a sequence), 20,730 remained after removing duplicate sequences, and 20,394 remained after discarding sequences that resulted in incorrect code. This was reduced to 20,348 after AISE as the tool failed to process some of them.

7.3.2 Extension Instruction Identification

The experiments in this chapter use a tool based on the AISE algorithm originally described by Atasu et al. [2005a]. The algorithm is described in section 2.3.1. The tool operates in three phases:

1. **Instrumentation**; wherein the AISE tool augments the intermediate representation of the application with counters for profiling. The CoSy-based tool emits the i686 assembly for this profiling executable which is then assembled and run using the standard GNU tool chain.

2. **Execution**; running the instrumented binary records per-basic-block execution frequencies, which are stored in a file for use by the extension phase.

3. **Extension**; The IR is augmented with profiling statistics, which are then used to select the top four instructions using the Atasu ILP AISE algorithm [Atasu et al., 2005b], which is briefly described in section 2.3.1. The AISE tool's profiler combined with a latency table for the given target architecture produces run-time and code-size performance metrics for the original transform-space sample. These metrics and the generated instructions are stored alongside the transformed code and transform-point definition.

7.3.3 Performance Evaluation

Run-time performance statistics are estimated for the whole program using the same profiling information that is used to rank the instruction templates by their run-time potential. The profiling provides per-basic-block execution frequencies representing the number of times a basic block will be executed in a single execution of the whole program. Each of these frequencies are multiplied by the latency of their basic block as software, and all the resulting values are summed; this provides the baseline number of cycles that the program will take to execute. In the absence of performing a full cycle-accurate simulation, this forms a reasonable prediction of performance. The hardware accelerated performance is obtained by multiplying each basic block frequency with the cycle-savings made for the templates in that basic block, and summing these values to get the total saving in cycles made. This saving is then subtracted from the software run-time in cycles to produce the cycle count for the hardware accelerated application. The relative reduction in cycles of hardware versus software is taken as the speed-up. If there are any inaccuracies in the models predicted cycle-count for each basic block then it will affect both the software and the hardware accelerated numbers. So even if the speed-up ratio is slightly inaccurate the trends between different runs will remain the same, so for the purposes of evaluating transformations this model is sufficient. See the critical evaluation, section 7.6.1, for a consideration of how effective this evaluation methodology is compared to those used chapters 4 and 5. This is similar to the wide instruction format specified in section 5.3 but here perfect use of extension instruction is assumed.

The improvement in code size is estimated in a similar fashion to run-time performance, although any code which is linked without analysis, such as system libraries, is not considered. The size of each instruction is assumed to be identical, which can be achieved through implicit operands for instructions with large numbers of operands, although this complicates register mapping. Where a complex instruction has been used, the number of original instructions that this covered is summed and subtracted by one to get the code size improvement in instructions. The total number of instructions in the program is summed, and then each of the calculated

code size improvements are subtracted from this value to form the code size improvement for the entire application.

7.4 Evaluation Methodology

For the purposes of this experiment, the instructions latencies were configured to those of an Intel XScale PXA270 processor [Intel, 2007], a high-performance embedded micro-architecture based upon the ARM7 instruction set. An input/output port constraint of 8/8 is set, to allow a wide range of potential extension instructions and avoid limitations due to the synthetic micro-architectural constraints set in the AISE algorithm. It has been shown [Pozzi and Ienne, 2005] that pipelining of extension instructions is possible to reduce per-cycle register file I/O to suit actual requirements.

Therefore, for each benchmark, for each of up to 10,000 transformation-space sample points there are:

1. Source code after transformation.

2. Instruction Set Extensions defined as data-flow templates.

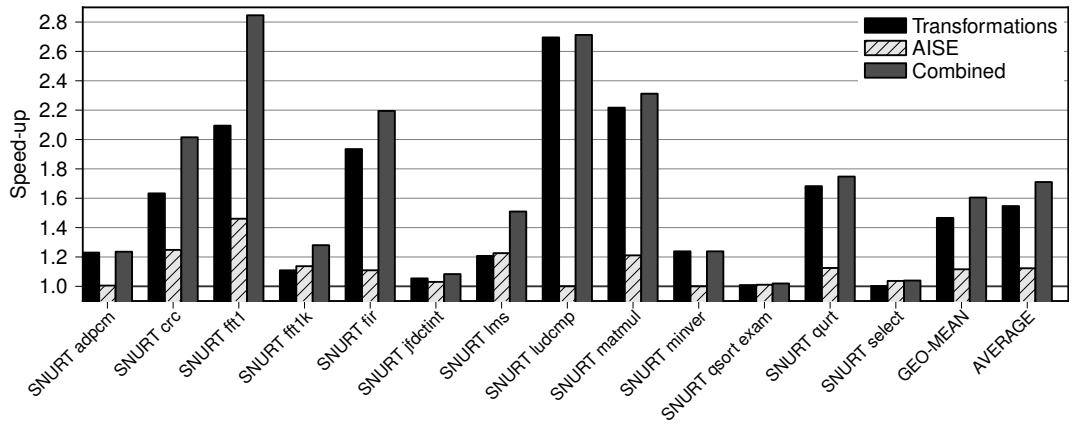
3. A record of performance in cycles (run-time) and instructions (code-size) before and after the transformations are applied to the benchmark.

4. A record of the improvement to each of the performance metrics for each of the instructions generated by AISE for the transformed source.

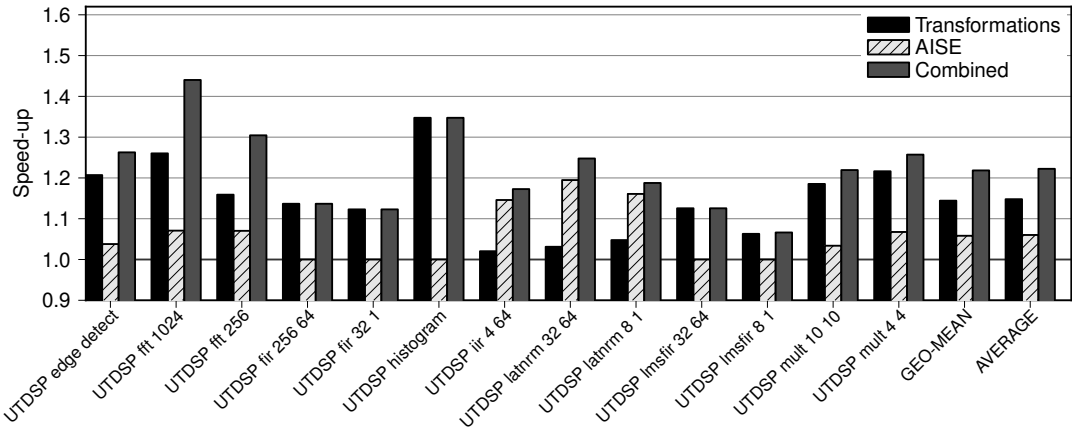
5. Aggregation of the results of the top four of these instructions to calculate the overall benefit to the transformed code.

So that there is a control point for reference, there is always a baseline in the transformation space that utilises no transformations. The number of extension instructions used in these experiments is arbitrarily limited to four in order to only include the largest and best performing extensions, such as those that are expected to be revealed through transformation. Some commercial approaches such as the Tensilica XPRES [Tensilica Inc., 2005] tend to use large numbers of small instructions to preserve generality.

This entire experiment was run on a quad-core machine, over the course of several days in order to allow for the large-scale sampling. The tools are “pipelined” in their operation to speed up results generation.



(a) Speed-ups achieved on the SNURT benchmarks.



(b) Speed-ups achieved on the UTDSP benchmarks.

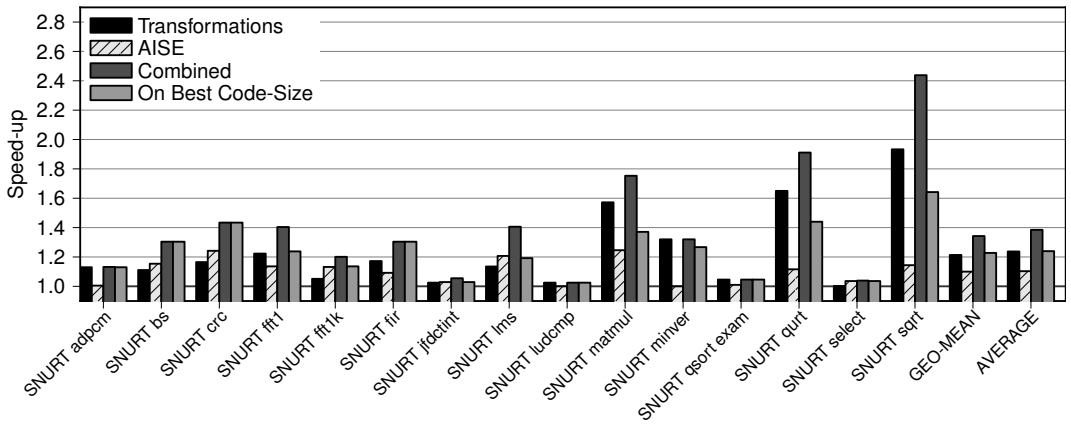
Figure 7.4: *Maximum speed-ups on experiment one (random sampling) with transformations alone, AISE alone, and combined transformations and AISE.*

7.5 Results

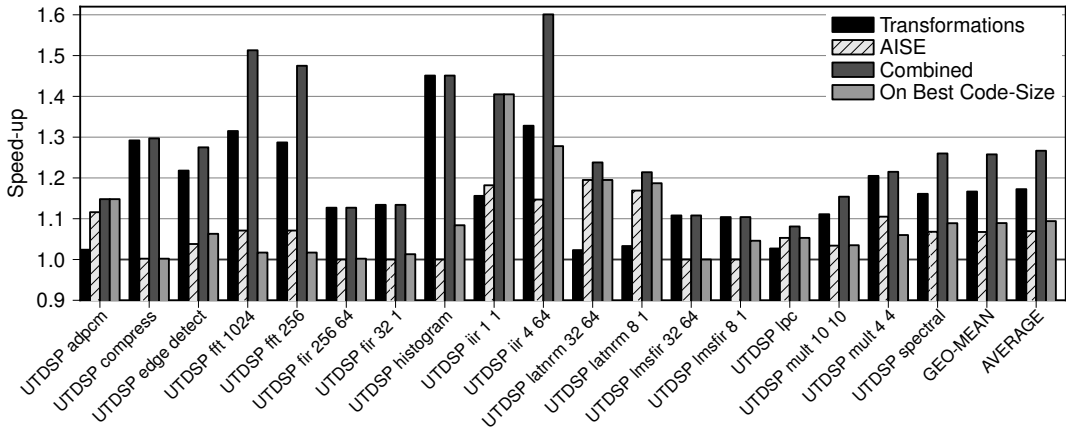
In this section results are presented and discussed, initially concentrating on performance and code size improvements due to combined source-level transformation and AISE. The data collected from the exhaustive enumeration of the transformation space is then analysed. The focus of the analysis is the detailed interaction between the transformations and extension instructions for each benchmark. Finally, the individual contributions of each transformation is examined.

7.5.1 Performance and Code Size Results

Figures 7.4 and 7.5 show the speed-ups achieved on a selection of benchmarks from the SNURT and UTDSP suites and summarise the random and exhaustive enumeration experiments respectively. For each benchmark the first three bars represent the best improvement seen in the search



(a) Speed-ups achieved on the SNURT benchmarks.

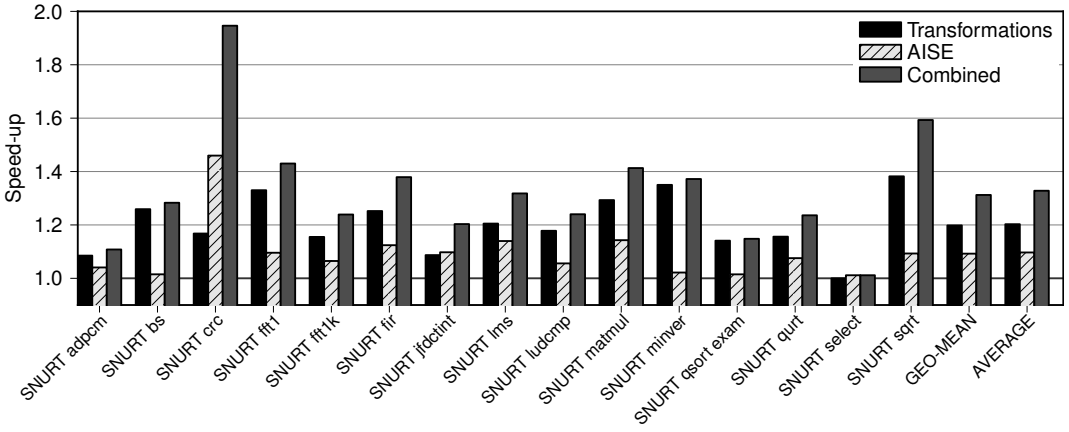


(b) Speed-ups achieved on the UTDSP benchmarks.

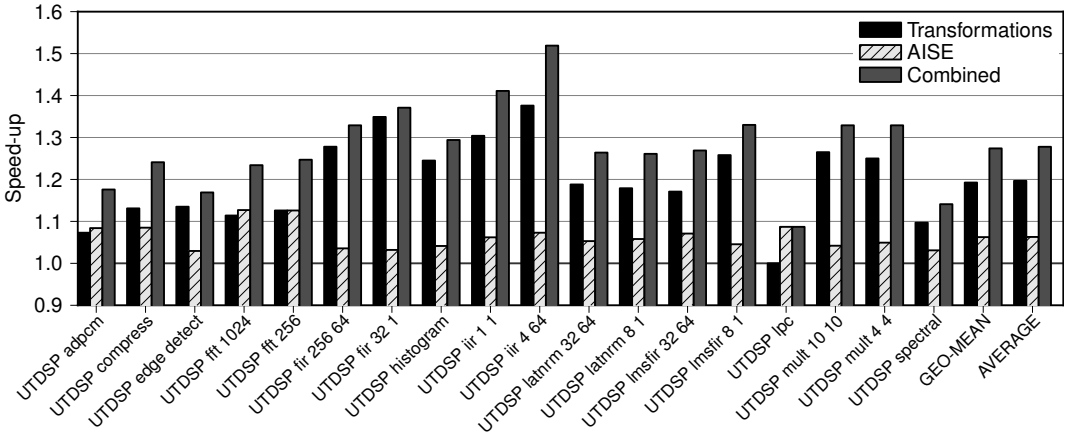
Figure 7.5: *Maximum speed-ups on experiment two (exhaustive enumeration) with transformations alone, AISE alone, and combined transformations and AISE.*

space for each technique: transformations alone, AISE alone, and the combination of the two. Peak speed-ups by a factor of 2.70x (SNURT *ludcmp*), 1.46x and 2.85x (both SNURT *fft*) are seen, respectively. The average speed-ups across both benchmark suites for experiment one are 1.35x, 1.09x and 1.43x respectively. It can also be seen that of the 25 benchmarks considered by experiment one, five of them see a combined transformation and AISE speed-up of over 2.0x and only seven see an improvement of less than 1.15x. The average speed-ups for experiment two are 1.20x, 1.08x and 1.32x respectively.

The fourth bar per-benchmark included in figure 7.5 is the combined run-time speed-up achieved with the transformation sequence that resulted in the smallest combined code-size, i.e. the run-time speed-up associated with the results in figure 7.6. As one would expect it can be seen that in general the smallest code is not the fastest, most notably so on the UTDSP



(a) Code-size improvements achieved on the SNURT benchmarks.



(b) Code-size improvements achieved on the UTDSP benchmarks.

Figure 7.6: *Maximum code size improvements over both experiments (random and exhaustive enumeration) with transformations alone, AISE alone, and combined transformations and AISE.*

benchmark suite. The overall average speed-up for this property is 1.16x, which is much lower than the 1.32x speed-up achieved by the fastest sequences.

The reason for the dip in performance with SNURT between experiments one and two is a combination of fewer transformations being applied per-sample in experiment two and the addition of I/O code to SNURT between the two experiments. The time spent actually doing I/O or in the standard library is not included in the performance measurements, It is just exists to allow additional verification, but the time spent in I/O related code within the benchmark itself is included. This I/O code is difficult to improve on but accounts for a non-negligible proportion of the run-time and thus the improvements to the core kernel of the benchmark are impacted. This issue was later corrected for the experiments in chapters 4 and 5 by using cycle-

counters in the simulator. This option, however, was not available when the experiments in this chapter were performed.

Figures 7.6(a) and 7.6(b) show the code-size improvements achieved on the same benchmarks but presents a combined summary of experiments one and two. These graphs are not based on the same sample points that speed-up figures are, but separate transformation sequences that were found to be effective at reducing code-size. Peak code-size improvements of factors of 1.18x (SNURT *minver*), 1.46x and 1.95x (both SNURT *crc*) are seen, for transformations alone, AISE alone and the combination of the two, respectively. The average code-size improvements across both benchmark suites are 1.20x, 1.08x and 1.30x respectively.

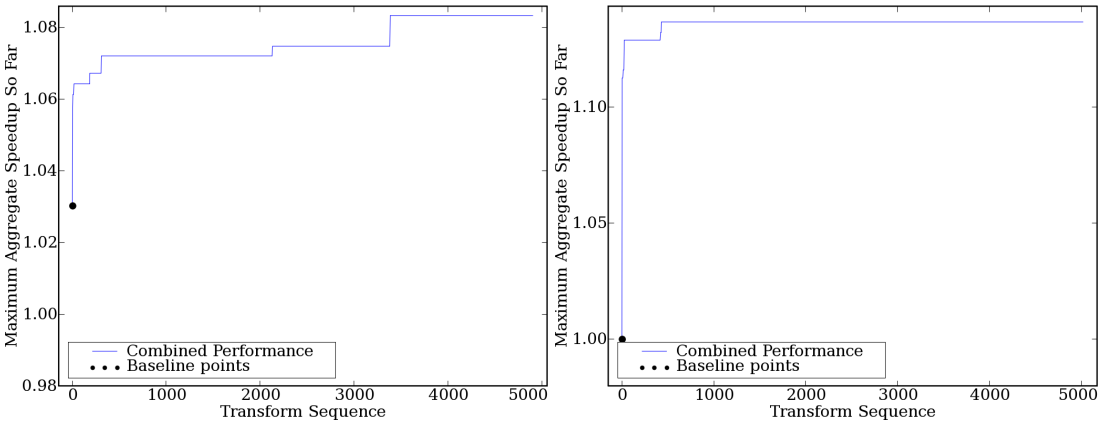
It can be seen that in figures 7.4, 7.5 and 7.6 that on average the results for the SNURT benchmarks are noticeably higher than for UTDSP. The primary reason for this is that the SNURT benchmarks are smaller, so the potential selection space is reduced and thus better suited to uniform sampling. Although only a tiny fraction of the overall search space is explored (small number of combinations of transformations for the "random" experiment and short sequences of a reduced set of transformations for the exhaustively enumerated space) good results were still obtained. It seems likely, however, that exploring a larger portion of the search space will yield further improved results, especially for larger programs. Larger programs are also likely to benefit from a more directed search technique that can quickly focus on the promising areas on the search space, such as the ones described in Franke et al. [2005] and Chow and Wu [1999].

This shows that in many cases simply choosing transformations that allow effective use of an extension instruction will not give good overall performance. Strong examples of this are the UTDSP fir-256_64 and *fir-32_1* benchmarks, where the optimal combined performance is given by a set of transformations that did not allow any speed-up through AISE at all. Examples where combined performance is significantly better than either transformations or AISE alone are SNURT *crc* and the SNURT *fft* benchmark.

7.5.2 Application-Oriented Evaluation

Initially, some examples are used to demonstrate how many iterations of the "random" algorithm are required before a significant performance improvement can be achieved. Then each benchmark is examined in turn to identify the best transformation sequences resulting from the exhaustive enumeration experiment. This is undertaken with the aim of finding commonalities between benchmarks and how the best transformation sequence changes under the influence of AISE.

Figure 7.7 shows the performance of the best transformation sequence found so far as each point in the "random" sample space is evaluated. Figure 7.7(a) shows an example (for the SNURT *jfdctint* benchmark) that has the kind of characteristics that led to evaluating such a



(a) SNURT *jfdctint* (b) UTDSP *fir-256_64*

Figure 7.7: *Performance improvement (y-axis) in relation to the number of evaluated sample points/program versions generated by the source-level transformation tool (x-axis).*

high number of samples in the transformation space. It contains several steps in the performance of the best sequence found so far, with the very best not being found until after several thousand samples were evaluated. This was not typical of most benchmarks, figure 7.7(b) is an example (for UTDSP *fir-256_64*) that shows the typical behaviour. It also has steps in the performance of the best sequence found so far, but they are much closer together and the very best is found in about five hundred runs, with none of the remaining sequences evaluated doing better. This suggests that considering a smaller number of samples can be sufficient to produce acceptable results for some applications if, for example, a single transformation is responsible for the majority of the performance gain. Considering a larger number of samples may find more steps leading to even greater performance, though.

In table 7.1 the best transformation sequences as per the exhaustive enumeration experiment are shown for all UTDSP and SNURT benchmarks. The most striking observation is the clear dominance of transformations T_{49} (*loop unrolling*), T_{59} (*forward propagation*), and T_{12} (*dead code elimination*) in the UTDSP applications. Notable exceptions to this rule are the *adpcm*, *compress*, *fft* and *lpc* benchmarks. An inspection of the source codes reveals that the applications that benefit most from T_{49}, T_{59} and T_{12} contain one or more nested computational loops performing linear array traversals. Benchmarks containing loops and non-linear array traversals, such as FFT, still benefit from *loop unrolling*, but require additional transformations such as *common sub-expression elimination* and *loop invariant hoisting* to develop their full performance. This is in line with the observations that for this benchmark (a) a significant number of operations are dedicated towards address computation, (b) large amounts of redundancy can be exploited by moving parts of the address computations out of the inner loop, and

Suite	Benchmark	Transformation Sequence	Suite	Benchmark	Transformation Sequence
	adpcm	T_{51}, T_{59}, T_{12}		*adpcm-test*	T_{12}, T_{40}, T_{59}
	compress	T_{49}, T_{44}, T_{51}		*bs*	T_{59}, T_{12}, T_{8}
	edge-detect	T_{49}, T_{44}, T_{51}		*crc*	T_{51}, T_{59}, T_{12}
	fft-1024	T_{47}, T_{8}, T_{49}		*fft1*	T_{51}, T_{59}, T_{47}
	fft-256	T_{47}, T_{8}, T_{49}		*fft1k*	T_{37}, T_{59}, T_{12}
	fir-256_64	T_{49}, T_{59}, T_{12}		*fir*	T_{46}, T_{59}, T_{12}
	fir-32_1	T_{49}, T_{59}, T_{12}		*jfdctint*	T_{49}
	histogram	T_{49}, T_{59}, T_{12}		*lms*	T_{49}, T_{12}
UTDSP	*iir-1_1*	T_{37}, T_{59}, T_{12}	SNURT	*ludcmp*	T_{37}, T_{59}, T_{12}
	iir-4_64	T_{49}, T_{51}, T_{59}		*matmul*	T_{40}, T_{12}, T_{47}
	latnrm-32_64	T_{49}, T_{59}, T_{12}		*minver*	T_{49}, T_{59}, T_{12}
	latnrm-8_1	T_{49}, T_{59}, T_{12}		*qsort-exam*	T_{40}, T_{59}, T_{12}
	lmsfir-32_64	T_{49}, T_{59}, T_{12}		*qurt*	T_{49}, T_{59}, T_{12}
	lmsfir-8_1	T_{59}, T_{49}, T_{12}		*select*	T_{37}, T_{12}, T_{59}
	lpc	T_{49}, T_{47}, T_{51}		*sqrt*	T_{49}, T_{59}, T_{12}
	mult-10_10	T_{49}, T_{59}, T_{12}			
	mult-4_4	T_{49}, T_{59}, T_{12}			

Table 7.1: *Overall best transformation sequences for the exhaustive enumeration experiment.*

(c) the remaining address calculations can be made more efficient by implementing them in extension instructions. The *adpcm* program is more control-flow intensive than the other codes and, hence, benefits most from control-flow optimisations. It is also noteworthy to mention that same transformation sequence generates optimal results for benchmarks differing only in the size of their data sets, e.g. *fft*, *fir*, and *latnrm*, but not *iir*.

Transformations T_{49}, T_{59} and T_{12} also play an important role for the SNURT benchmarks, but the situation is less obvious than with UTDSP. Still, the loop and array based codes benefit in the same way from these transformations as before. As can be seen from the *fft* example, variations in the coding style of what is otherwise the same algorithm can lead to different optimal transformation sequences. Analogous to UTDSP, the more control-flow or bit-level manipulation oriented codes such as *adpcm*, *bs* or *crc* react to a different set of transformations than the loop-oriented codes.

The distinct correlation between certain code characteristics (“code features”) and transformations enabling a performance gain suggests it may be possible to exploit this relationship

and, for example, employ machine learning techniques to *predict* a "good" transformation sequence without the need for a costly exploration of the optimisation space. This, however, is outside the scope of this chapter.

7.5.3 Transformation-Oriented Evaluation

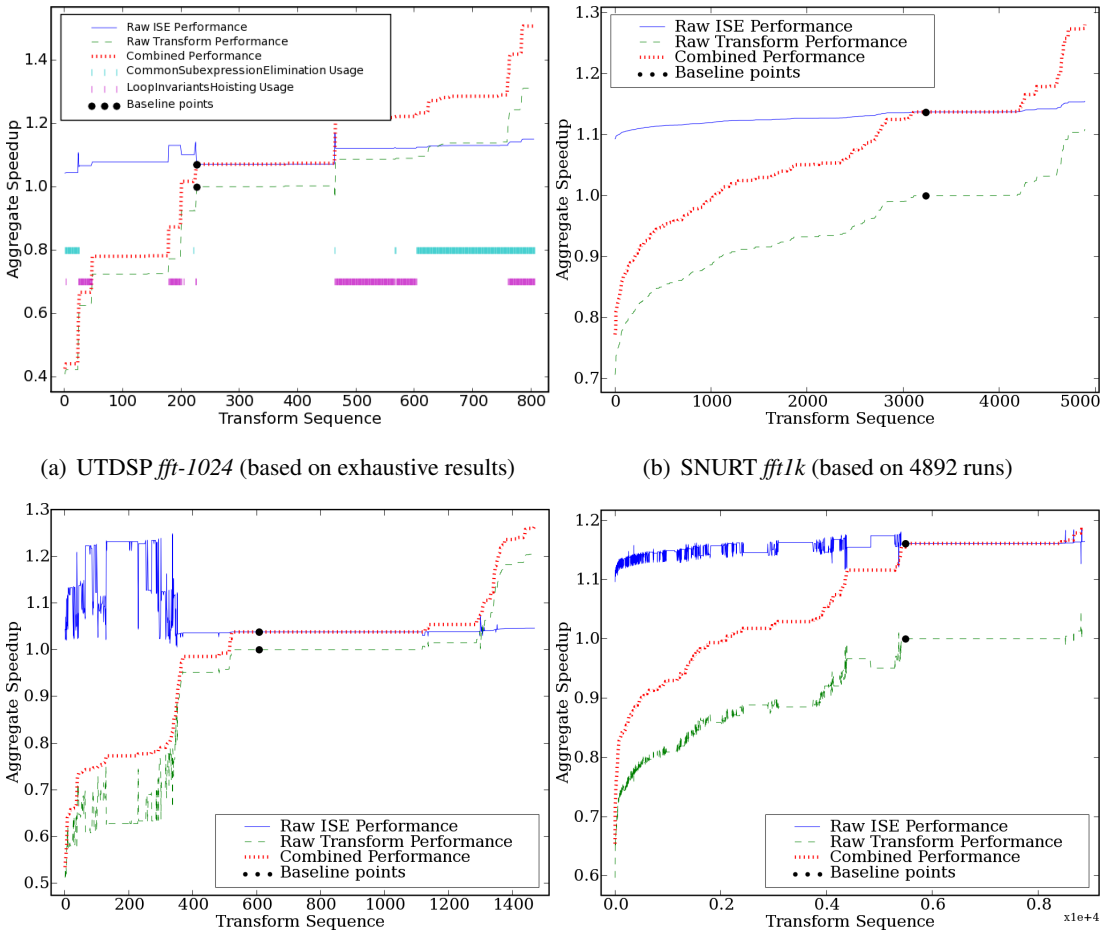
The graphs in figure 7.8 show the performance for each individual technique and the combination of the two for every sample point in the search space, for a small selection of benchmarks. The samples are sorted by the performance of combining transformations and AISE. This allows the ratio of transformation to AISE performance to be seen and also shows where there are correlations between the performance of the two individual techniques. These correlations are seen where either the performance of both individual techniques improve at the same point or where one gets better but the other gets worse.

An example of this correlation can be seen on the left side of figure 7.8(c) which shows the separated performance for sets of transformations which allow good AISE performance but perform poorly overall due to the performance decrease seen with transformations alone. A more useful example of the correlation between transformations and AISE is shown in the motivating example, UTDSP *fft-1024*, with the *common sub-expression elimination* and *loop invariant hoisting* transformations. Sequences that make use of these transformations are marked as short vertical bars in figure 7.8(a). It can be seen that all the best performing sequences make use of both of these transformations, performance improves greatly when either of them are turned on, and more so when they both are.

Figure 7.8(b) shows an almost ideal set of results (for SNURT *fft1k*), where the best set of transformation sequences when considered alone also allows the most gain from AISE. When the optimal sequences overlap in this way the combined performance is very high (e.g. going from peaks of 1.11x and 1.14x with individual techniques to a peak of 1.28x with combined techniques for SNURT *fft1k*). Figure 7.8(d) shows the results from a benchmark where almost none of the overall improvement comes from transformations but almost entirely from AISE (UTDSP *latnrm-8_1*). The graph still shows, however, that poor code shape can limit AISE.

The second, exhaustive, set of results allows for each transformation to be evaluated individually as each gets applied a fixed number of times and the shorter sequences means there is less 'piggy-backing' along-side good transformations.

For each transformation in the exhaustive results, figure 7.9 enumerates the 'good' and 'bad' outcomes achieved, where 'good' means the performance is better than the baseline. Speed-ups are shown in figures 7.9(a) and 7.9(b), code-size improvements are shown in figures 7.9(c) and 7.9(d). As with the earlier graphs code-size is presented in terms of improvement, therefore being greater than the baseline means the code indicates that smaller. Figures 7.9(a) and 7.9(c) are the improvements seen for pure transformation without AISE relative



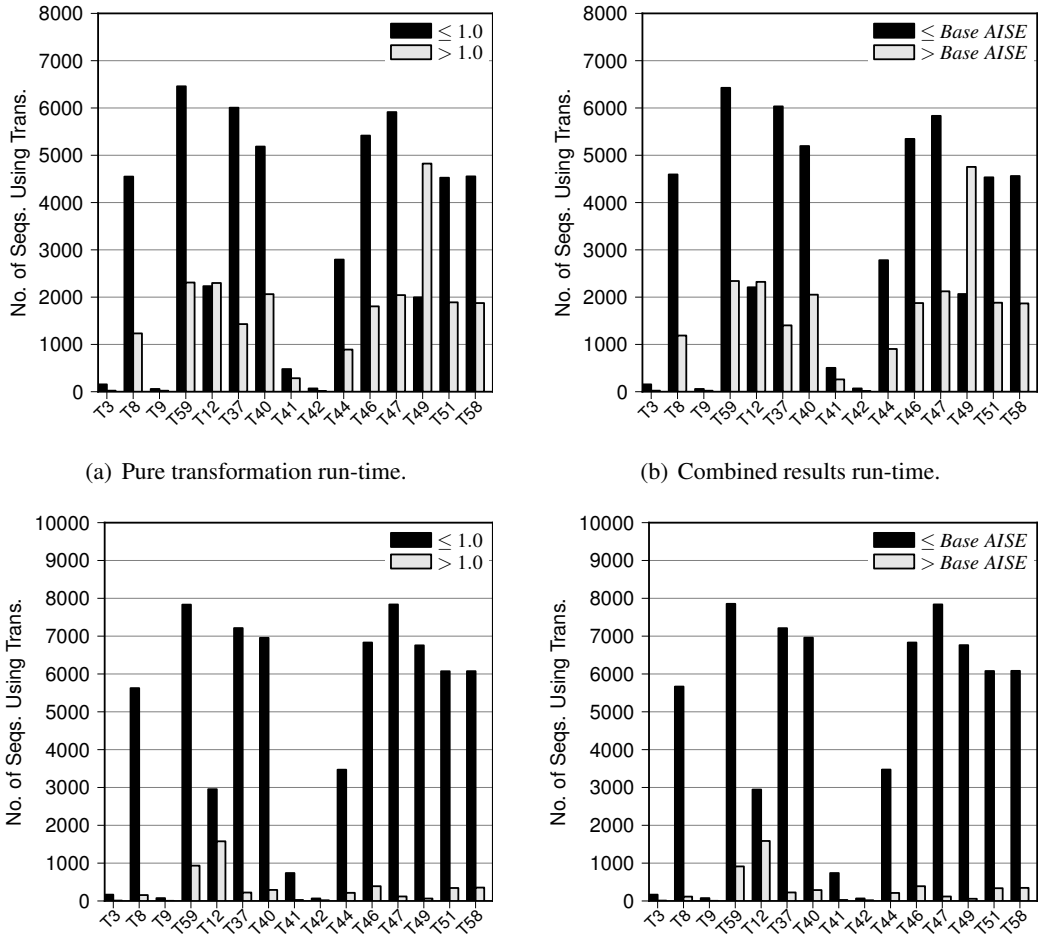
(a) UTDSP *fft-1024* (based on exhaustive results)

(b) SNURT *fft1k* (based on 4892 runs)

(c) UTDSP *edge-detect* (based on 1470 runs)

(d) UTDSP *latnrm-8_1* (based on 8882 runs)

Figure 7.8: *Speed-ups achieved for every transformation sequence in the search space for a selection of benchmarks. For each version of the program (x-axis) three speed-up values are shown: speed-up after AISE only (raw AISE performance), speed-up after source-level transformation only (raw transformation performance), and speed-up after combined AISE and source-level transformation (combined performance). The baseline points are the performance of each technique on unmodified code. The code versions are ordered by increasing combined performance along the x-axis. Figure (a) also has two horizontal bars to indicate which sequences used two key transformations:* common sub-expression elimination *and* loop-invariant hoisting*.*



(a) Pure transformation run-time.

(b) Combined results run-time.

(c) Pure transformation code-size.

(d) Combined results code-size.

Figure 7.9: *Distribution of 'good' and 'bad' sequences containing each transformation. The* greater-than *columns are 'good' and the* less-than-or-equal-to *columns are 'bad'.*

to the baseline (1.0). Figures 7.9(b) and 7.9(d) are the improvements seen for combined transformations and AISE relative to the performance of AISE on the identity code. The code-size distributions contain more 'bad' sequences than the run-time distributions because most of the transformations used are oriented towards performance rather code-size.

The per-transformation results are presented in tables 7.2 and 7.3 for speed-ups and code-size improvements respectively. The full name and description of each transformation ID may be found in appendix A. The 'usage' column refers to the number of times the transformation appeared in non-duplicate and valid sequences. The 'within 5% of best' column is the percentage of the sequences containing the given transformation that resulted in performance at least 95% as good as the very best sequence (which may or may not contain the given transformation). This percentage can be seen as a description of the risk factor of the given transformation,

Per Transformation Run-time Improvements					
Trans.	Usage	Within 5% of Best	Trans. Perf.	Combined Perf.	Expected Perf.
All	20,348	–	1.20x	1.32x	1.30x
T_3	177	7%	1.09x	1.10x	1.18x
T_8	5363	13%	1.12x	1.22x	1.21x
T_9	79	1%	1.00x	1.07x	1.09x
T_{59}	6828	48%	1.19x	1.31x	1.29x
T_{12}	6828	37%	1.19x	1.31x	1.29x
T_{37}	5983	9%	1.13x	1.24x	1.23x
T_{40}	5795	24%	1.15x	1.26x	1.25x
T_{41}	764	16%	1.15x	1.28x	1.25x
T_{42}	66	0%	1.01x	1.09x	1.09x
T_{44}	2697	8%	1.09x	1.18x	1.19x
T_{46}	5508	23%	1.15x	1.26x	1.24x
T_{47}	6224	25%	1.14x	1.25x	1.24x
$T_{49}[4]$	5174	62%	1.21x	1.33x	1.31x
T_{51}	5036	24%	1.16x	1.27x	1.26x
T_{58}	5050	21%	1.15x	1.26x	1.25x

Table 7.2: *Speed-up information for each transformation used in the exhaustive experiments.*

the higher the percentage the lower the risk that a sequence containing the transformation will perform poorly. The 'trans. perf.' column lists the best improvement achieved using only transformations while the 'combined perf.' column lists the best improvement achieved by applying transformations followed by AISE. These two numbers do not necessarily relate to the same transformation sequence. Finally, the 'expected perf.' column is an estimate of what the combined performance should be based on taking the product of the best pure transformation improvement and the performance of AISE on the identity code. The difference between this and the combined performance shows the average enabling or disabling effect of the transformations on AISE over all the sequences containing this transform. The very first row of each table shows the performance across all sequences without regard to which transformations they contain.

A number of observations are noted for these results. The combined performance is very slightly higher than expected on average for both run-time and code-size improvements across the 20,348 sequences – showing an overall additional enabling factor of AISE from the transformations. No single transformation, however, shows a significant improvement over the

Per Transformation Code-Size Improvements					
Trans.	Usage	Within 5% of Best	Trans. Perf.	Combined Perf.	Expected Perf.
All	20,348	–	1.20x	1.29x	1.28x
T_3	177	22%	1.16x	1.20x	1.24x
T_8	5363	13%	1.12x	1.20x	1.20x
T_9	79	1%	0.94x	1.02x	1.01x
T_{59}	6828	91%	1.19x	1.28x	1.27x
T_{12}	6828	93%	1.19x	1.29x	1.28x
T_{37}	5983	12%	1.17x	1.26x	1.25x
T_{40}	5795	14%	1.17x	1.26x	1.26x
T_{41}	764	7%	0.52x	0.55x	0.56x
T_{42}	66	0%	1.00x	1.07x	1.07x
T_{44}	2697	5%	0.94x	1.01x	1.01x
T_{46}	5508	19%	1.20x	1.29x	1.28x
T_{47}	6224	7%	1.17x	1.26x	1.25x
$T_{49}[4]$	5174	0%	0.82x	0.92x	0.89x
T_{51}	5036	21%	1.18x	1.27x	1.26x
T_{58}	5050	17%	1.18x	1.27x	1.26x

Table 7.3: *Code-size improvement information for each transformation used in the exhaustive experiments.*

expected performance when considering sequences across all benchmarks, though T_3 (*bounds comparison substitution*) does notably worse than expected for run-time performance. If considering the results on a per-transformation per-benchmark basis it can be seen that the numbers making up these averages have a larger amount of variance. There are too many combinations to present them all, but a few highlights are:

- T_8 (*common sub-expression elimination*) was expected to give a 1.96x speed-up on SNURT *matmul* but only a 1.75x speed-up was actually achieved. A 1.41x speed-up was expected for UTDSP *fft-1024* but a 1.51x speed-up was actually achieved.
- $T_{49}[4]$ (*loop unrolling* with a factor of 4) was expected to give a speed-up of 1.81x to SNURT *matmul* but only achieved a speed-up of 1.62x. Though for SNURT *sqrt* it achieved a run-time speed-up of 2.44x when only 2.21x was expected.

Other more general results may be observed, such as T_{59} (*forward and constant propagation*), T_{12} (*dead code elimination*) and $T_{49}[4]$ (*loop unrolling*) are mostly likely to be part of a good transformation sequence for improving run-time when considering combined transformations and AISE. To improve code-size T_{59} (*forward and constant propagation*) and T_{12} (*dead*

code elimination) are again likely to be involved in good sequences, but T_{44} (*induction variable detection*) and $T_{49}[4]$ (*loop unrolling*) are not. These observations are in line with a compiler writer's "intuition".

7.6 Summary and Conclusions

7.6.1 Critical Evaluation

The most notable omission in this performance evaluation is a cache model. The primary reasons for excluding this was to allow more accurate comparisons with previous AISE work – which does not use a cache model either, though without considering compiler transformations there is less motivation to do so – and to avoid complicating the analysis of the results. Introducing caches would mean that there would be additional interactions between the transformations and the cache configuration, which would potentially obscure the effect of the transformations on AISE performance. To try and ensure that the lack of a cache model did not affect the results, no loop-level data transformations were considered. *Loop unrolling* was used, but that mainly affects the instruction cache. It is also worth noting that a constant memory cost is assumed in this evaluation and this is equivalent to having a scratchpad memory. This is quite common for many of the digital signal processing applications as a data cache does not handle streaming data very efficiently. In fact, many of the UTDSP benchmarks used for evaluation in this chapter were used for the evaluation of chapter 6, where they operated entirely from scratchpad memories.

7.6.2 Future Work

The area of future work with the most potential is exploring loop-based transforms, such as a using a polyhedral model to be a transformation space of different loop shapes.

Related to the work in this chapter a transformation space could be specified to explore how it can enhance a complex instruction mapper (such as *MapISE*).

7.6.3 Summary

This chapter described a methodology for improved AISE that combines the exploration of high-level and generic platform-independent source transformations and low-level extension instruction identification. It has been demonstrated that source-to-source transformations are not only very effective on their own, but provide much larger scope for performance improvement through AISE than any other isolated low-level technique. Both source-level transformations and AISE have been combined in a unified framework that can efficiently optimise both hardware and software design spaces for extensible processors.

The empirical evaluation of the design space exploration framework is based on a model of the Intel XScale processor and compute-intensive kernels and applications from the SNURT and UTDSP benchmark suites. It has been successfully demonstrated that this approach is able to outperform any other existing approach and gives an average speed-up of 1.49x. Compared to previous work [Bonzini and Pozzi, 2006], a much broader array of existing transformations have been covered. This provides a more global picture of the potential for transformation in improving instruction set extension. In addition, it has been empirically demonstrated that there exists a non-trivial dependence between high-level transformations and the generated instruction set extensions justifying the co-exploration of the hardware and software design spaces.

Chapter 8

Conclusion

"Après nous, le déluge."
— Madame de Pompadour (Jeanne-Antoinette Poisson), *mistress to King Louis XV*, 1721–1764.

In this thesis an extensive evaluation of the compiler's role in ASIP use and production was undertaken. A set of compilation techniques that increased data and memory throughput were presented. Finally a design space exploration was undertaken to investigate how the compiler could enhance AISE's capabilities.

8.1 Contributions

This thesis has made contributions in four main areas, these contributions are summarised below.

8.1.1 Compiling for AISE

Commercially successful AISE systems rely on small, simple extension instructions which are easy to use during compilation. Techniques which produce large, complex extension instructions can, in theory, produce much higher speed-ups. Without compiler support, however, making use of these instructions is a difficult process. This thesis has demonstrated that by focusing solely on complex extension instructions a high-level pass can use computationally expensive algorithms while maintaining an acceptable run-time.

8.1.2 AISE for Compiling

Existing AISE techniques are fully focused on finding extension instructions which accelerate the target application as much as possible. They do not consider ease-of-use from either a compiler or a programmer's perspective. This thesis manually searched a small AISE parameter space while evaluating the AISE and was quickly able to achieve better results than when AISE was optimising purely for its internal model. This was further improved by introducing

a heuristic to AISE to represent difficulty of compilation. This was able to significantly increase performance. Finally, it was proposed that a complicated extension hardware interface be discarded and replaced with a simpler one – even if that meant smaller extension instructions. Using this interface the compiler was able to achieve a higher speed-up even though the automatically produced extension units were smaller.

8.1.3 Exploiting Dual Memory Banks

A source-level dual memory bank assignment technique was presented. Genetic programming was found to out-perform an "optimal" integer linear programming solution and could be automatically re-tuned for a different processor. Therefore, combined with its source-level design the assignment tool is able to target almost any dual memory bank system and could provide additional speed-ups over any AISE generated speed-ups.

8.1.4 Transformation-Based DSE and AISE

Finally a source-to-source compiler was used to drive AISE. A large transformation-space was explored and it was discovered that combining transformations and AISE could result in performance greater than the mere product of their individual gains. Additionally a set of key transformations for AISE are found.

8.2 Critical Evaluation

8.2.1 Integration

Several different evaluation methodologies were used in this thesis, chapters 4 and 5 used a hardware-verified cycle-accurate simulator, chapter 6 used a hardware DSP platform and chapter 7 used a performance model. Additionally, a few experiments in chapter 5 built models around the cycle-accurate simulator.

The results from chapter 6 (dual memory banks) can be applied directly to the results in the previous two chapters (complex instruction mapping and AISE). Implementing dual memory banks on *EnCore* would be possible as the ISA already supports it, but as none of the hardware, the simulator or the compiler support dual memory banks it would be a large amount of work and would not produce substantially different results from the alternative DSP processor that was used for evaluation. The models that were built on-top of the simulator were valid first steps for evaluating the proposed techniques and care was take to make them as accurate as possible. A more complete implementation to verify the existing model would still be a sensible step to take.

Finally, chapter 7 (transformations) operates on a separate model from the previous three chapters. It does not allow direct comparison with those chapters, but that is not required – it is not presenting a technique in the same sense as those chapters. Rather it is performing a design space exploration and then extracting conclusions from the results. These conclusions can be applied universally regardless of the underlying model.

8.2.2 Limits of AISE

Section 5.3 proposed a 64-bit instruction format which could access four inputs and write to four outputs. Of course, a long instruction word is only one step away from a very long instruction word, or VLIW. In many ways AISE and VLIW are similar: they are both used in the embedded domain, they both target data-level parallelism and they both require extensive compiler support. This last point being well-known for VLIW, and demonstrated by this thesis for AISE. It has been shown in this thesis that there are strong advantages to nudging AISE a little in the direction of VLIW. On the other hand, Jain et al. [2004] have shown that there are advantages to nudging VLIW a little in the direction of AISE. AISE was able to reduce VLIW issue-width and register file size by adding extension instructions to the processor. There are, therefore, elements that can be brought over to either system, and perhaps a full AISE/VLIW hybrid would be able outperform either.

While there were many AISE technologies not used in this thesis there is one that could have potentially solved many problems: state-holding extension instructions. This is an existing solution to the register bandwidth issues identified in chapter 4. The extension unit contains a scratchpad memory which is used to pass data from one instruction to the next, without ever being visible to the baseline processor. Whereas compiling for standard fully architecturally visible scratchpad memories is a well studied topic, compiling for hidden AISE scratchpads is not so well understood. Thus this thesis did not consider them at all, but instead found less esoteric solutions to the register bandwidth problem.

A global trend has been noticed with the results in this thesis. The maximum average speed-up, over 179 benchmarks, that any experiment could achieve was approximately 1.25x. “Eliminating Poor Mappings” achieved 1.20x, “Hard-Wiring Constant Values” also reached 1.20x, “Register Load Cost” found a speed-up of 1.24x, combining these last two techniques earned a speed-up of 1.26x. “Wide Instructions” managed to achieve a speed-up as high as 1.28x, but only which an unfeasibly complicated architecture, a more realistic architecture reached a 1.26x speed-up. Finally, although *ISEGen* estimates a 1.54x speed-up, section 4.9.1 briefly demonstrates that this is generally off by a factor of at least 2x, which reduces *ISEGen's* speed-up estimate to 1.27x. The fact that all these numbers are around the same value could be coincidence, or it could represent the limit of available performance for the AISE tool (*ISEGen*) on this suite of benchmarks. If this does represent the limit of performance then the work

presented in chapter 5 was able to reach the limit with several separate proposals.

8.3 Insights

This section briefly summarises some insights which can be gained from the results presented in this thesis. Some of these insights imply future work, but they are not included in that section as it is not clear how that work should proceed.

Section 4.5 showed that it is sometimes beneficial to ignore extension instructions and many of the results in chapters 4 and 5 showed that it is rare for every extension instruction to get used. This means that the AISE tool is generating instructions that the compiler is never using, which is an obvious waste of expensive hardware resources. A simple post-processing pass could be used to eliminate these instructions before hardware synthesis but it would be better to avoid generating them in the first place. The creation of the unused instructions may be blocking the generation of other, useful, instructions. Section 5.1.3 already describes one additional heuristic for *ISEGen* that greatly improved the usefulness of the generated instructions by including one compiler issue in the cost model. It seems likely that further changes could be made to the algorithm so that the AISE tool is trying to maximise the expected gains of instructions when used by compilers rather than their expected gains in an idealised processor model.

A good place to start trying to solve this problem would be to directly tackle the single biggest performance blocker: fitting irregular instructions onto a regular datapath. *ISEGen* generates very unusual instructions with irregular dataflow, but then data is fed in and out of the instructions using regular vector registers. This mismatch results in many extra cycles spent moving data into the correct position. This thesis presented several methods of minimising this problem: skipping instructions in the compiler if they are likely to require too many extra cycles moving data (section 4.5); adding a heuristic to *ISEGen* to try and avoid generating instructions where this is likely to be an issue (section 5.1.3); or most drastically, eliminating the vector registers completely (section 5.3). An interesting alternative would be to have the AISE tool target vectors directly. *ISEGen* operates on an IR derived from a scalar SSA IR, an AISE tool that operates on an IR derived from the vector descriptions of an auto-vectorising compiler may be able to improve performance by producing large instructions which fit perfectly onto the vector register model.

Another area where AISE tools could benefit from further work is in the area of domain-based instruction set extension. A small experiment described in section 4.8.4 showed that *ISEGen* is not able to perform domain-based analysis. Even large benchmarks suffer in performance, partially because the use of many algorithms in the single application means that *ISEGen* struggles to find a good representative set of extension, as it is effectively treating the

large benchmark as if it is many smaller benchmarks glued together. A domain-based AISE tool would be able to target situations where a single processor must perform many tasks – this is becoming increasingly common due to device consolidation.

Section 8.2.2 has already said that AISE and VLIW are targeting the same set of performance gains and this entire thesis has been about creating a basis for compilation on AISE generated processors. An area that is ripe for inspiring further work on compilation for AISE processors is VLIW complilation: most of the transformations designed for VLIW compilers aim to increase the amount of data parallelism available. These are exactly the sort of transformations that would benefit AISE as well.

Compiler optimisations in general are crucial for effective AISE, chapter 7 showed that badly formed code results in AISE tools targeting these pieces of "low hanging fruit". So using the compiler middle-end to eliminate redundant or inefficient code is critical, otherwise AISE does not find good extension instructions. A further generalisation of this is that running a standard compiler middle-end (e.g. *GCC* `-O2`) helps to normalise the code into more standard patterns. This should increase the usability of extension instructions across different applications as even if the C is different between two implementations of a related concept the dataflow might end up the same.

8.4 Future Work

The previous chapters have already presented key ideas for future work. The most important possibilities are to extended the complex instruction mapper (*MapISE*) with the ability to partially map extension instructions based on arithmetic identities. Adapting to the restrictions of a polynomial-time graph-subgraph isomorphism library would provide significant run-time benefits. The existing machine learning work could be extended with more advanced techniques. Finally the transformation space that has already been explored could be extended through the use of polyhedral transformations.

An overall design space exploration experiment is essential for learning more about how the compiler should affect AISE. This would involve searching *ISEGen's* heuristic parameter space but instead performing evaluation using its internal model, *MapISE* should use the extensions and the resulting binary be run on the cycle accurate simulator. A simpler prototyping task would be to take the knowledge gained from chapter 5 and encode it in *ISEGen's* performance model.

The complex instruction mapper presented in this thesis can target any form of graph-shaped instruction. With a little retargeting effort it could target vector-based instructions such as MMX or SSE. It would be for an interesting evaluation to see if the technique described here could compete with a dedicated vectoriser.

Finally, the memory bank techniques from chapter 6 could be ported to target XY memory on *EnCore*, so as a true evaluation of using dual memory banks to increase memory bandwidth could be performed.

Appendix A

SUIF Transformation List

"Keep Ithaka always in your mind.
Arriving there is what you are destined for.
But do not hurry the journey at all.
Better if it lasts for years,
so you are old by the time you reach the island,
wealthy with all you have gained on the way,
not expecting Ithaka to make you rich.

Ithaka gave you the marvelous journey.
Without her you would not have set out.
She has nothing left to give you now."
— Constantine P. Cavafy, *Greek poet*, 1863–1933. Extract from "Ithaka" (1911), translated by Edmund Keeley and Philip Sherrard.

This appendix provides a complete list of SUIF1 based source-level transformations used in chapter 7 together with a short description of their function. There were two experiments in chapter 7, experiment two only made use of the reduced set of transformations in section A.1, experiment one used all the transformations in this appendix. As the aim of experiment one was to consider everything and see what had an effect there are many transformations in section A.2 that are not pure optimisations, e.g. the "dismantle" instructions perform lowerings. Some also did not ever apply to any of the benchmarks considered as they targeted code features that aren't found in those benchmarks, but as they were considered in the first experiment they are listed here. For a more detailed description of each transformation please refer to the SUIF documentation [Stanford Compiler Group, 1996].

A.1 Most Important Transformations

T_3**: Bounds Comparison Substitution.** This replaces comparisons of upper and lower bounds of a loop inside the loop body with the known result of that comparison.

T_8**: Common Sub-expression Elimination.** Simple common sub-expression elimination.

T_9: **Control Simplification.** Simplify `if` statements for which one branch or the other always executes. Remove `for` loops that are never executed.

T_{12}: **Dead Code Elimination.** Simple dead code eliminations.

T_{37}: **For Loop Normalisation.** Normalise all `for` loops to have lower bound of zero, step size of one, and less than or equal to test.

T_{40}: **Guard For.** This adds `if` nodes around some TREE_FOR nodes to ensure that whenever any TREE_FOR node is executed, the landing pad and first iteration will always be executed.

T_{41}: **If Hoisting.** This moves certain `if` nodes up in the code under some circumstances that can allow the test for the `if` to be eliminated.

T_{42}: **Imperfectly Nested Loop Conversion.** Turn imperfectly nested loop nests into perfectly nested loop nests by pulling conditionals as far out as possible.

T_{44}: **Induction Variable Detection.** This does simple induction variable detection. It replaces the uses of the induction variable within the loop by expressions of the loop index and moves the incrementing of the induction variable outside the loop.

T_{46}: **Lift Call Expression.** This takes any calls that are within expression trees out of the expression trees.

T_{47}: **Loop Invariant Hoisting.** This moves the calculation of loop-invariant expressions outside of loop bodies.

T_{49}: **Loop Unrolling:*x*.** Unroll loop x times. Values for x are limited to 2, 4, 5 or 7.

T_{51}: **Move Loop Invariant Conditionals.** Move all loop-invariant conditionals that are inside a loop outside the outermost loop.

T_{58}: **Unstructured Control Flow Optimisation.** Simple optimisations on unstructured control flow. Remove unreachable code. Remove labels that are not the target of any branch. Remove labels that are followed by unconditional jumps. Remove branches to same target as "natural" control flow. Invert condition of branch if it simplifies control flow.

T_{59}: **Full Copy, Forward and Const Propagation.** Composite transformation to perform copy propagation, forward propagation, constant folding, local constant propagation and constant extraction in this order iteratively until they stop changing the IR. These transformations were not considered individually in experiment two, this composite was used instead.

A.2 Additional Transformations

T_1: **Array Delinearisation.** Turn 1-dimensional arrays into multi-dimensional arrays.

T_2: **Bit Packing.** Combine local variables that are used only as single bits, packing them together into variables of type `int`.

T_4: **Break Load Constant Instruction.** This breaks all load constant instructions of a symbol and non-zero offset into an explicit addition of the offset.

T_5: **Call By Reference Replacement.** Replace call-by-reference scalar variables with copy-in, copy-out.

T_6: **Chain Array References.** Attempt to chain together multiple array reference instructions in series into a single array reference instruction.

T_7: **Constant Propagation.** Propagate forward the definition of constants.

T_{10}: **Constant Folding.** Fold constants wherever possible.

T_{11}: **Copy Propagation.** Propagate forward copy operations on simple local variables.

T_{13}: **Default SUIF Transformations.** This performs the default passes designed to be used immediately after the front end, to turn some non-standard SUIF that the front end produces into standard SUIF.

T_{14}: **Dismantle Array Instruction.** Dismantle array instructions (into explicit base plus offset).

T_{15}: **Dismantle Div Ceil/Floor Instruction.** Dismantle all ceil/floor instructions.

T_{16}: **Dismantle Div Mod Instruction.** Dismantle all mod instructions.

T_{17}: **Dismantle Empty Tree For.** This dismantles TREE_FORs with empty bodies.

T_{18}: **Dismantle Int Abs/Max/Min Instruction.** Dismantle all integer abs/max/min instructions.

T_{19}: **Dismantle Abs/Min/Max Instruction.** Dismantle all min/max instructions.

T_{20}: **Dismantle Multiway Branch.** Dismantles all multiway branch instructions.

T_{21}: **Dismantle Non Constant For.** This dismantles TREE_FORs unless the upper bound and step are both loop constants.

T_{22}: **Dismantle Tree Block.** This dismantles all TREE_BLOCKs.

T_{23}: **Dismantle Tree Block Without Symbol Table.** This dismantles all TREE_BLOCKs that have empty symbol tables.

T_{24}: **Dismantle Tree For.** This dismantles all TREE_FORs.

T_{25}: **Dismantle Tree For With Modified Index Variable.** This dismantles TREE_FORs for which the index variable might be modified by the TREE_FOR body.

T_{26}: **Dismantle Tree For With Spilled Index Variable.** This dismantles TREE_FORs with a spilled index variable.

T_{27}: **Dismantle Tree Loop.** This dismantles all TREE_LOOPs.

T_{28}: **Eliminate Enumeration Types.** This replaces all uses of enumerated types with a corresponding plain integer type.

T_{29}: **Eliminate Struct Copies.** This gets rid of all structure copies, whether through copy instructions, load-store pairs, or memcopy instructions.

T_{30}: **Eliminate Sub Variables.** This removes all sub-variables and replaces uses of them with uses of their root ancestors, with the appropriate offsets.

T_{31}: **Explicit Load Store.** This puts in explicit loads and stores for access to all variables that are not local, non-static, non-volatile variables without the addr_taken flag set.

T_{32}: **Extract Upper Array Bounds.** Attempt to extract upper bound information for array reference instructions from the variables for the arrays being referenced.

T_{33}: **Find For.** This builds TREE_FOR nodes out of TREE_LOOP nodes for which a suitable index variable and bounds can be found.

T_{34}: **Fix Address Taken.** Set the is_addr_taken flag of each variable to TRUE or FALSE depending on whether or not its address is actually taken.

T_{35}: **(Strictly) Fix Bad Nodes.** This fixes bad nodes. This is used as part of the default expansion after the front end.

T_{36}: **Fix LDC Types.** This puts the correct types on all ldc (load constant) instructions that load symbol addresses.

T_{38}: **Global Variable Privatisation.** Do some code transformations to help with privatisation of global variables across calls.

T_{39}: **Globalise Local Static Variables.** This changes all static local variables into global variables in the file symbol table.

T_{43}: **Improve Array Bounds.** Try to improve the array bound information by replacing variables used in array bounds with constants by looking for constant assignments to those variables at the start of the scope of each such array type.

T_{45}: **Kill Redundant Line Marks.** This removes all mark instructions that contain nothing but line information.

T_{48}: **Loop Flattening.** Same as loop unrolling, but unrolls the loop completely if it is small enough and does not contain too many iterations.

T_{50}: **Mod Ref Annotations.** This puts mod/ref annotations on TREE_FORs.

T_{52}: **Pointer Conversion.** Add array reference instructions in place of pointer arithmetic where possible.

T_{53}: **Privatisation.** This privatises every variable listed in the annotation "privatisable" on each TREE_FOR.

T_{54}: **Reduction Detection.** This finds simple instances of reduction. It moves the summation out of the loop.

T_{55}: **Replace Constant Variables.** Replace uses of variables with "is constant" annotations with constants based on the static initialisation information.

T_{56}: **Scalarisation.** This turns local array variables into collections of element variables when all uses of the array are loads or stores of known elements.

T_{57}: **(Aggressively) Scalarise Constant Array References.** Overlap array references with constant indexes with scalar variable.

Appendix B

Full Results

"Science is a way of trying not to fool yourself. The first principle is that you must not fool yourself, and you are the easiest person to fool."
— Richard Feynman, *Physicist*, 1918–1988.

The charts in this appendix are more complete versions of charts found in earlier chapters. The charts found in earlier chapters only contained a subset of the benchmarks that were used for evaluation, the charts in this chapter contain every benchmark. In some cases there are also additional charts that cover a wider range of parameters than are shown in the main text.

This appendix is not intended to be read directly, the charts lack any context except for their captions. They are intended to be referenced from the main text.

Some benchmark suite names are abbreviated in these charts to help them fit on the page. These abbreviations are:

- **Dint:** DSPstone fixed-point
- **Dfp:** DSPstone floating-point
- **E1A:** EEMBC-1 Automotive
- **E1C:** EEMBC-1 Consumer
- **E1N:** EEMBC-1 Networking
- **E1O:** EEMBC-1 Office
- **E1T:** EEMBC-1 Telecom
- **E2C:** EEMBC-2 Consumer
- **E2N:** EEMBC-2 Networking
- **U:** UTDSP
- **UU:** UTDSP unrolled

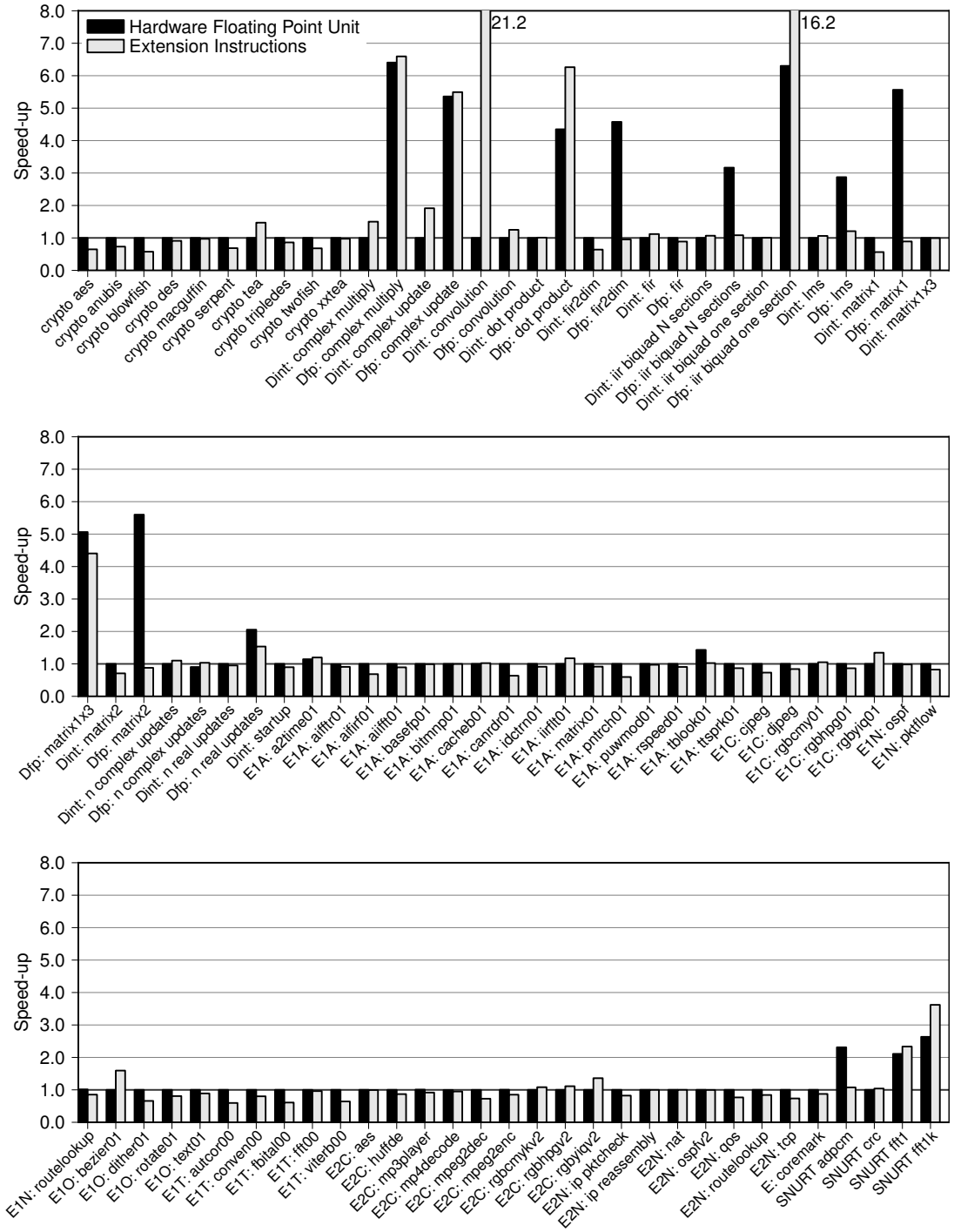


Figure B.1: Note: this is the full version of figure 4.6 on page 48. *A comparison of a hardware floating point unit with extension instructions. In this chart the baseline processor, that speed-ups are calculated from, has no floating point hardware.*
(Continued on the next page.)

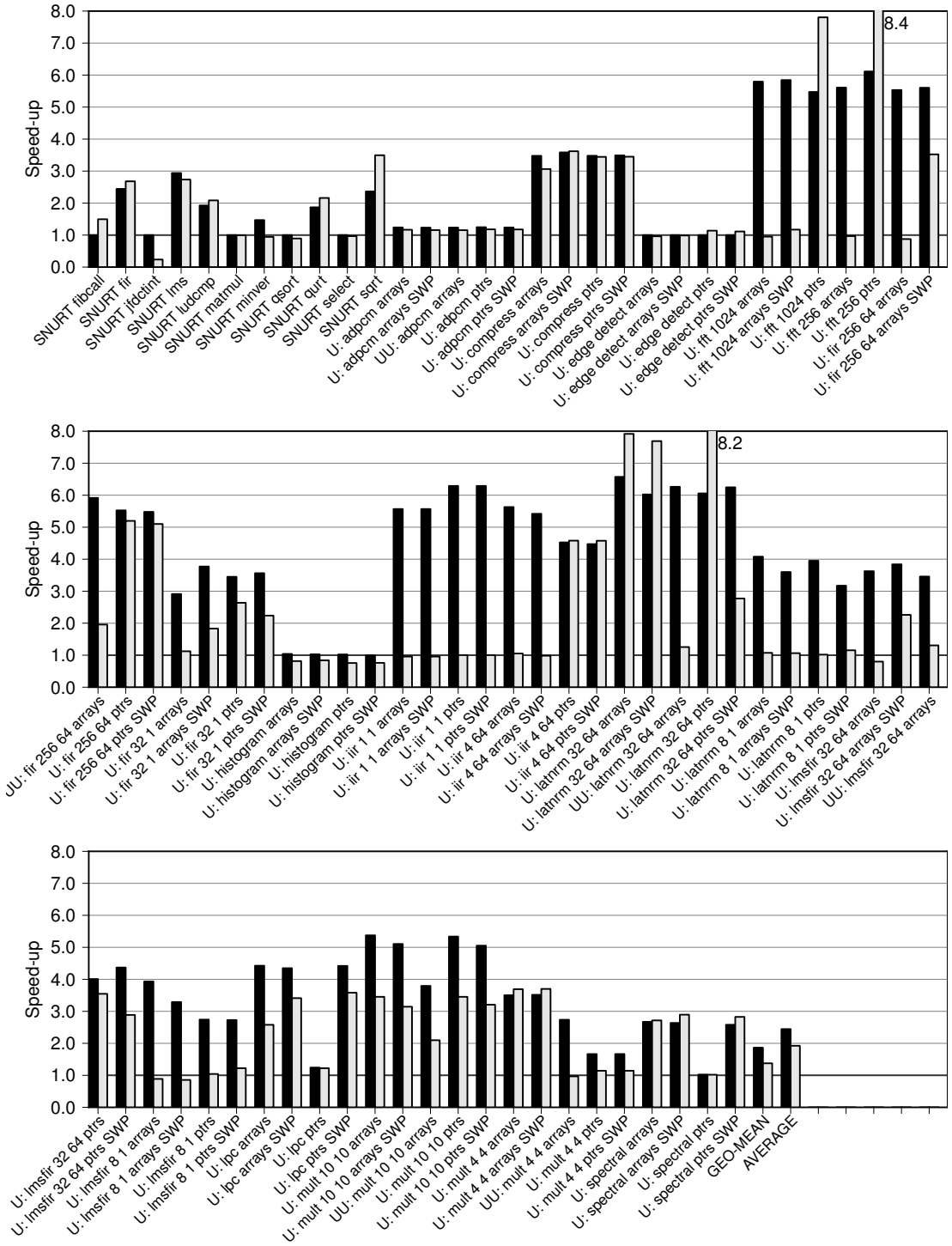


Figure B.1 (continued): *The left bar of each benchmark is the speed-up obtained by adding a floating point unit to the baseline processor, the right bars are the speed-ups obtained by adding extension instructions.*

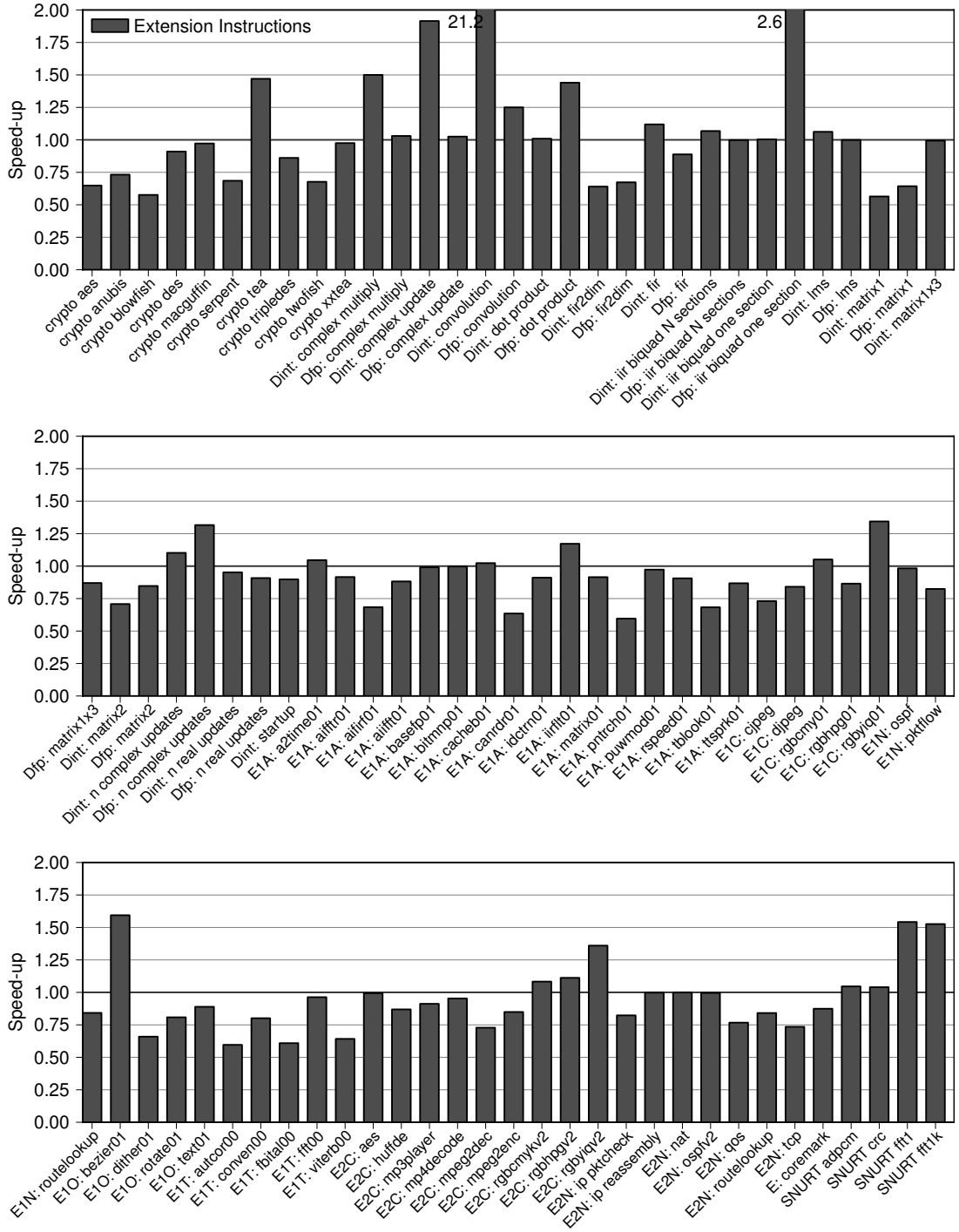


Figure B.2: Note: this is the full version of figure 4.7(a) on page 50. *The performance of* MapISE *with default settings. Each benchmark is using its own extension unit specialised for it by the AISE tools.* (Continued on the next page.)

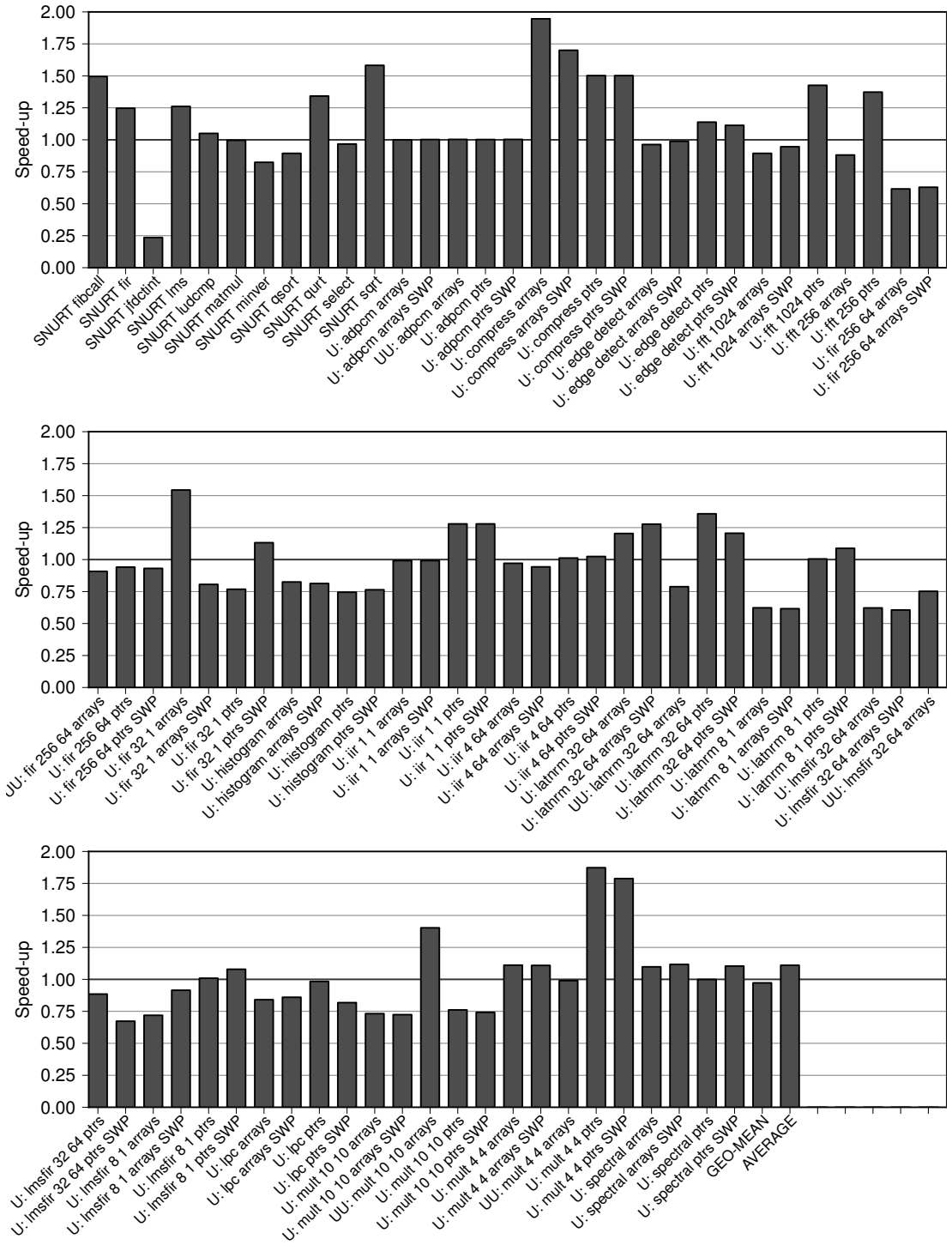


Figure B.2 (continued): *The speed-ups provided by adding extension instructions specialised to each benchmark. Note that unlike figure 4.6 the baseline processor used to calculate speed-up from in this (and subsequent) graphs* does *have a floating point unit.*

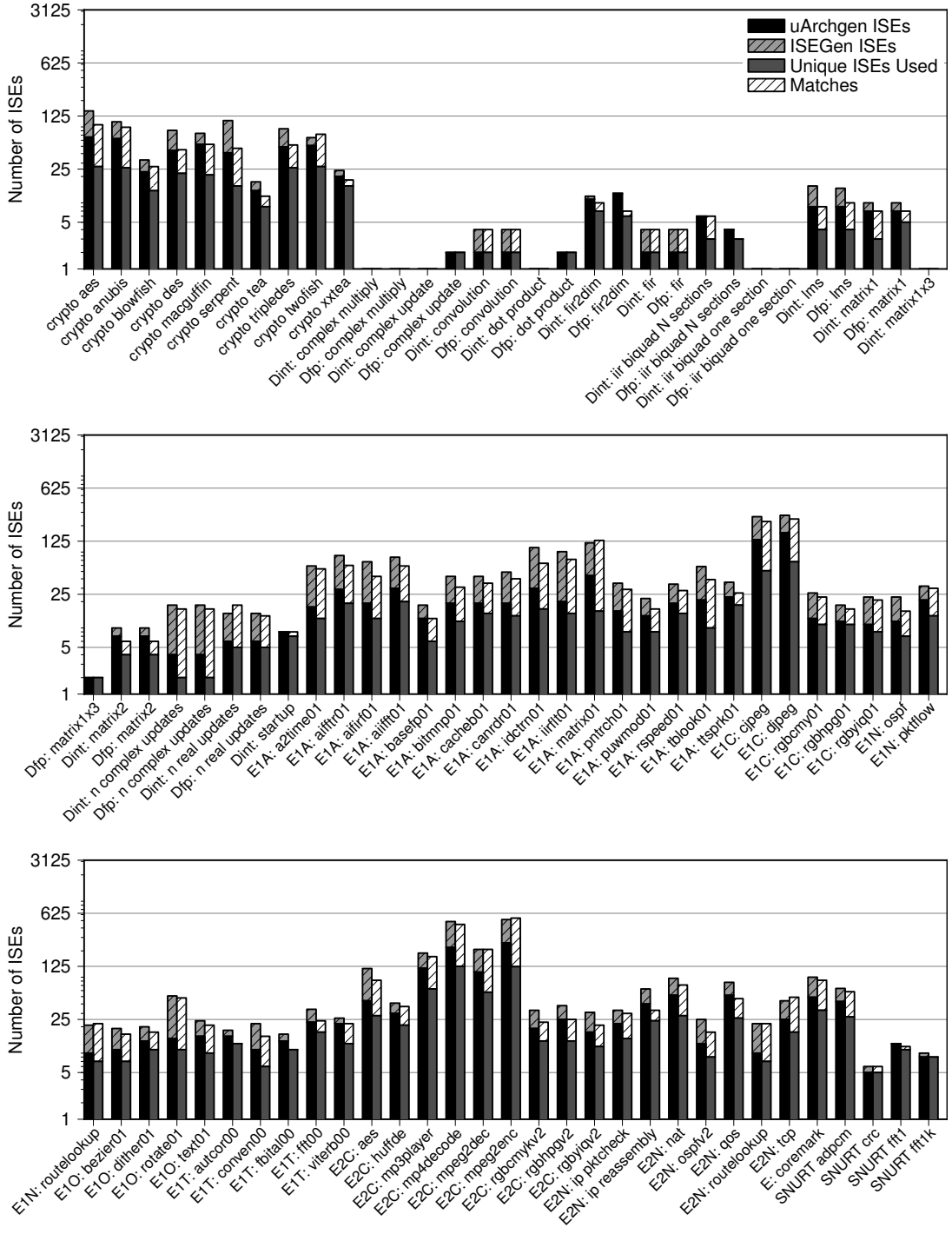


Figure B.3: Note: this is the full version of figure 4.7(b) on page 50. *The performance of* MapISE *with default settings. Each benchmark is using its own extension unit specialised for it by the AISE tools. Number of extension instructions found and used. Note that this chart has a logarithmic scale. The stacked bars on the left are the number of extension instructions* (Continued on the next page.)

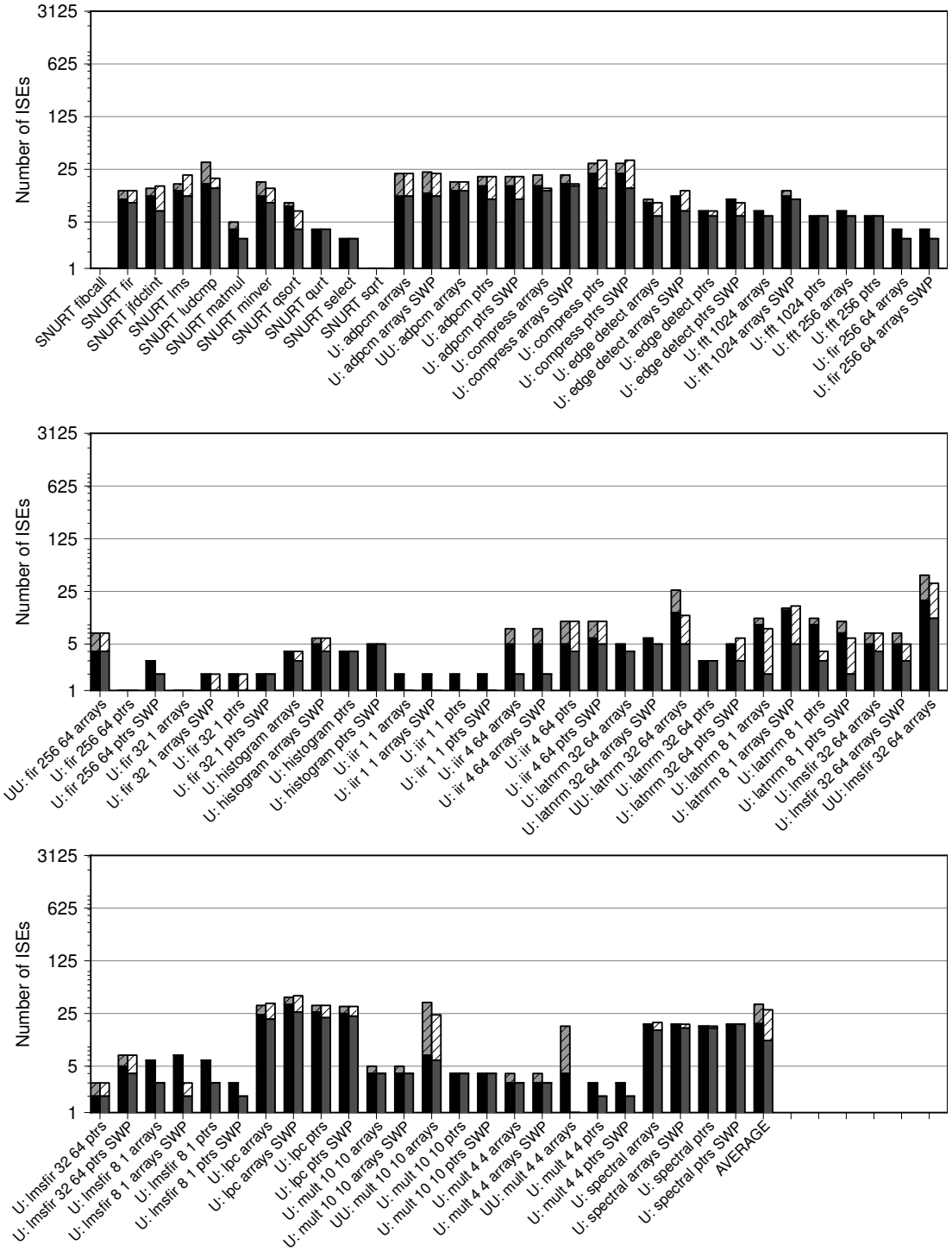


Figure B.3 (continued): *found by the AISE tools. The lower bar is the number of unique extension instructions, the upper bar is the number of non-unique extension instructions found. The stacked bars on the right describe the equivalent for* MapISE*. The lower bar is the number of unique extension instructions exploited, the upper bar is the total number of sites where extension instructions were used.*

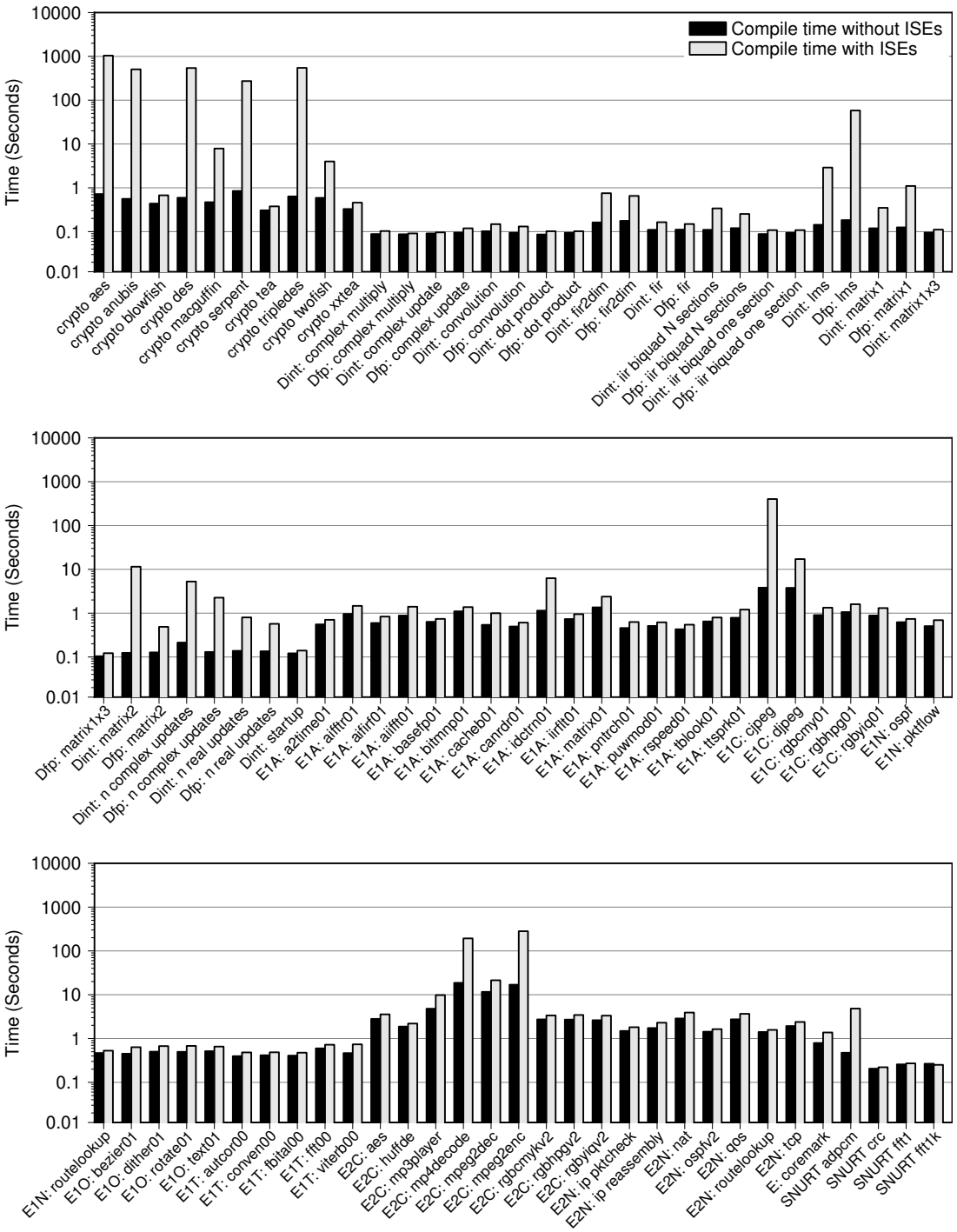


Figure B.4: Note: this is the full version of figure 4.8(a) on page 51. *Run-time of* MapISE *compared to other tools. Note that both charts have a logarithmic scale.*
(Continued on the next page.)

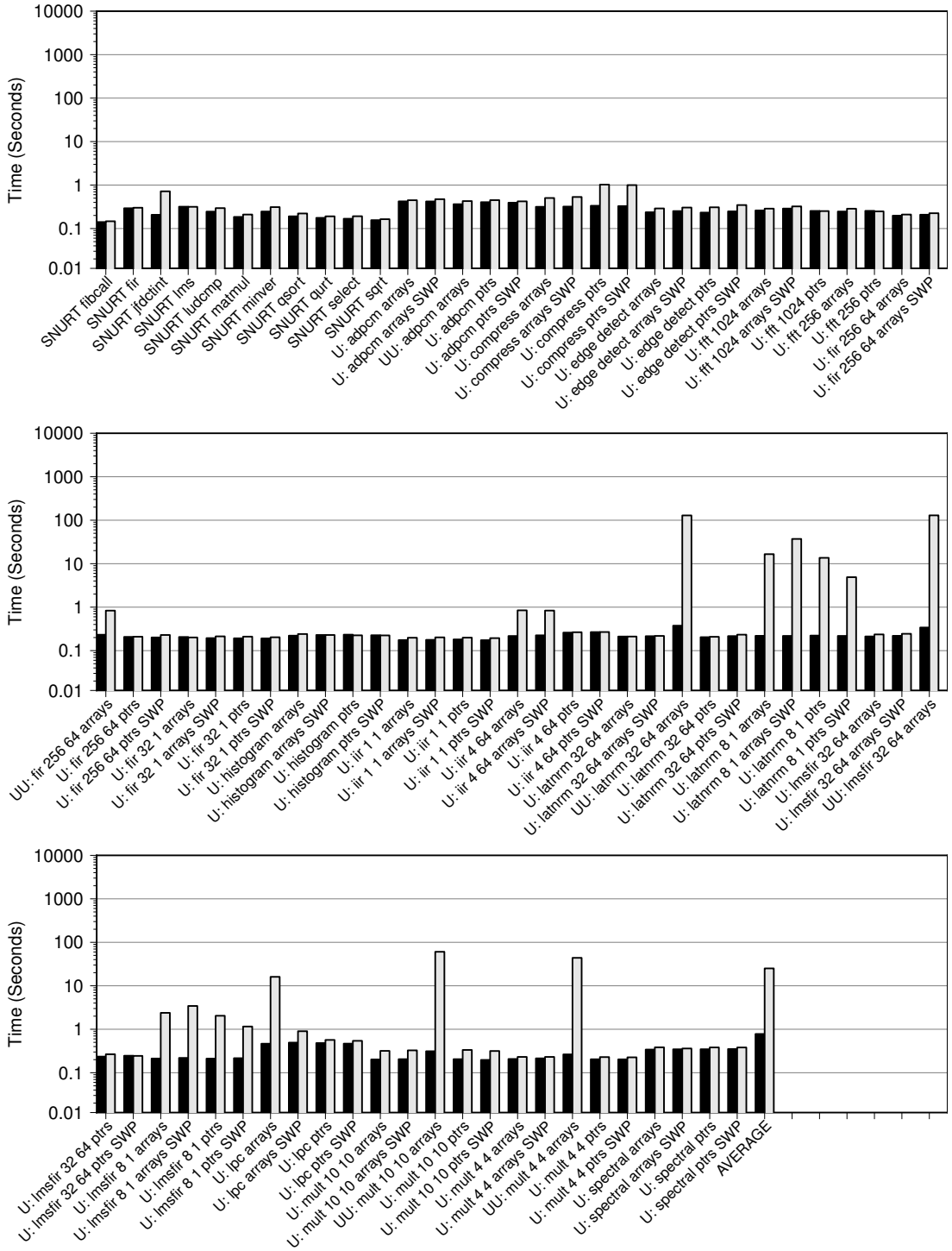


Figure B.4 (continued): *The time taken to compile each benchmark with the standard compiler (left-hand bar) or with* MapISE *(right-hand bar).*

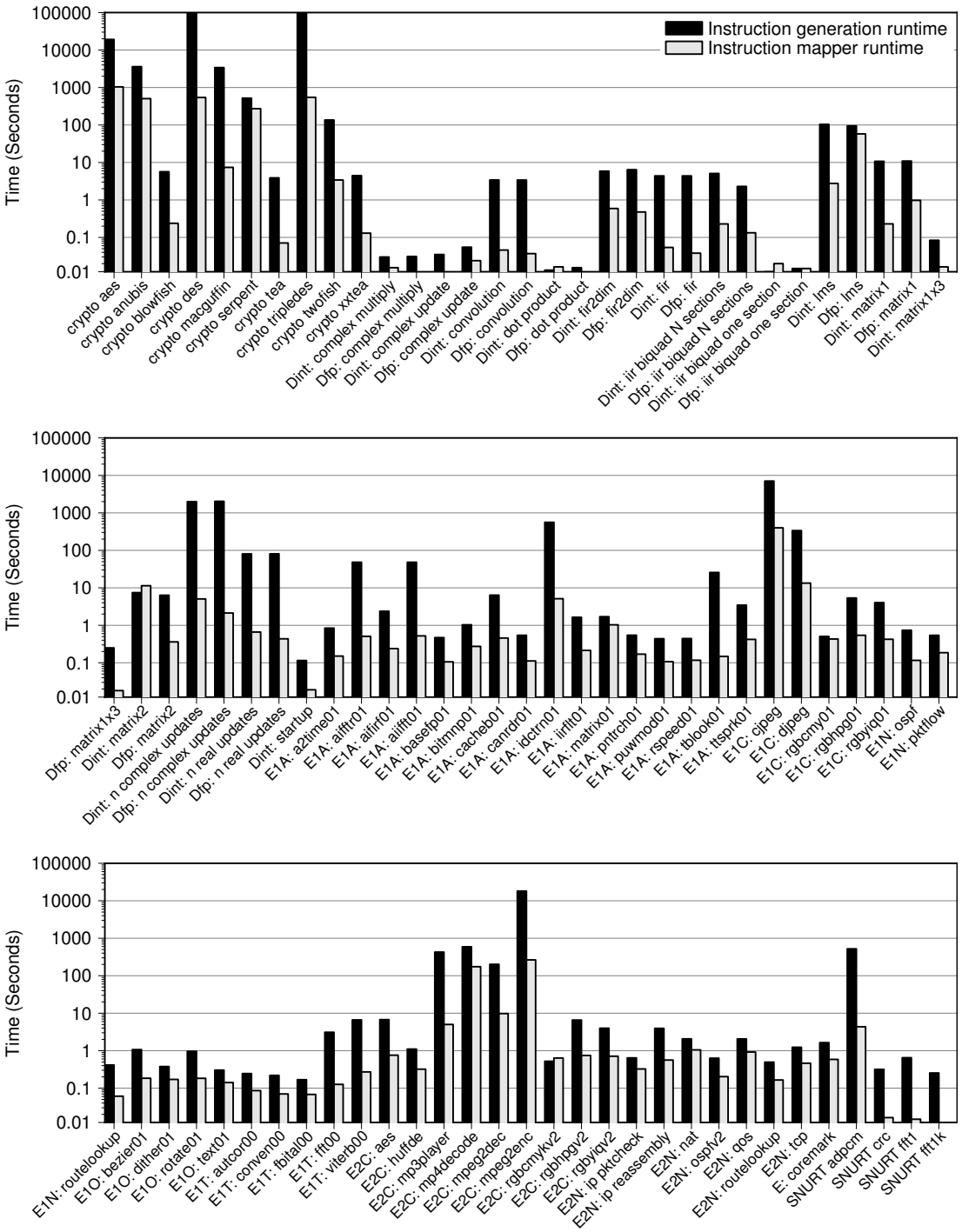


Figure B.5: Note: this is the full version of figure 4.8(b) on page 51. *Run-time of* MapISE *compared to other tools. Note that both charts have a logarithmic scale.*
(Continued on the next page.)

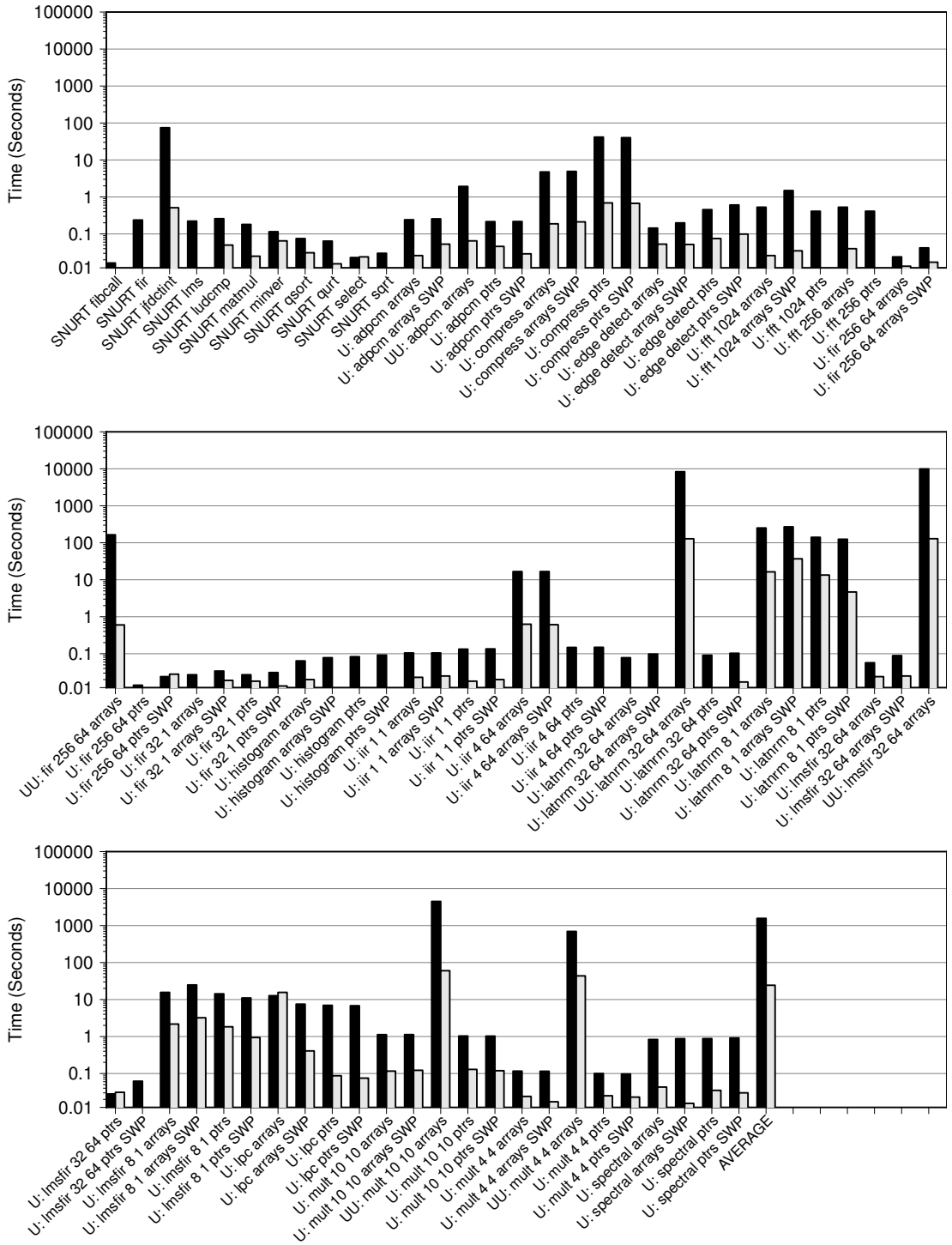


Figure B.5 (continued): *The left bar for each benchmark is the time taken to generate extension instructions (the run time of* ISEGen*, time to construct the hardware of extension unit is not included). The right bars are the length of time that* MapISE *adds to the compile time (over* GCC *alone).*

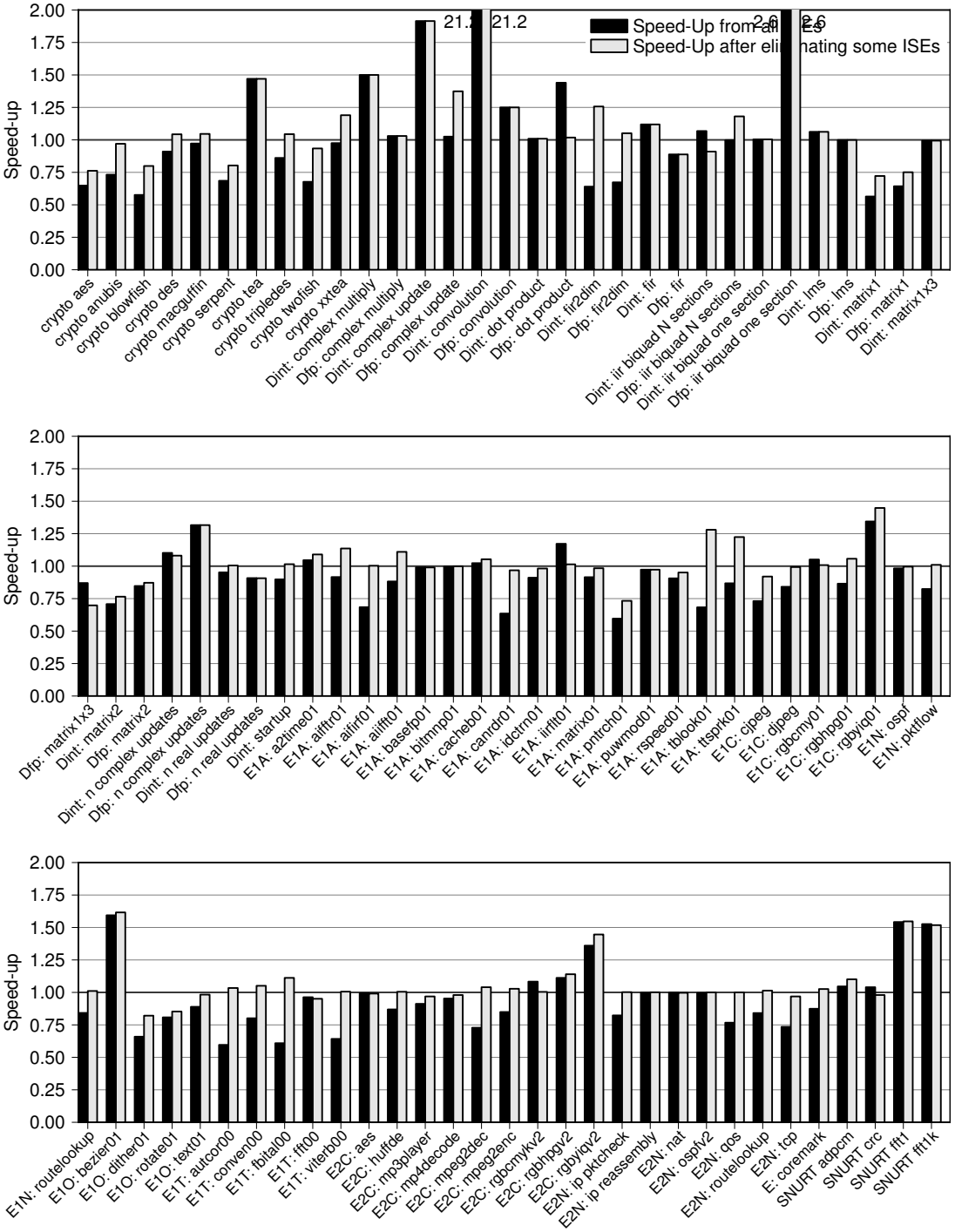


Figure B.6: Note: this is the full version of figure 4.9(a) on page 53. *An evaluation of eliminating poor mappings in* MapISE. (Continued on the next page.)

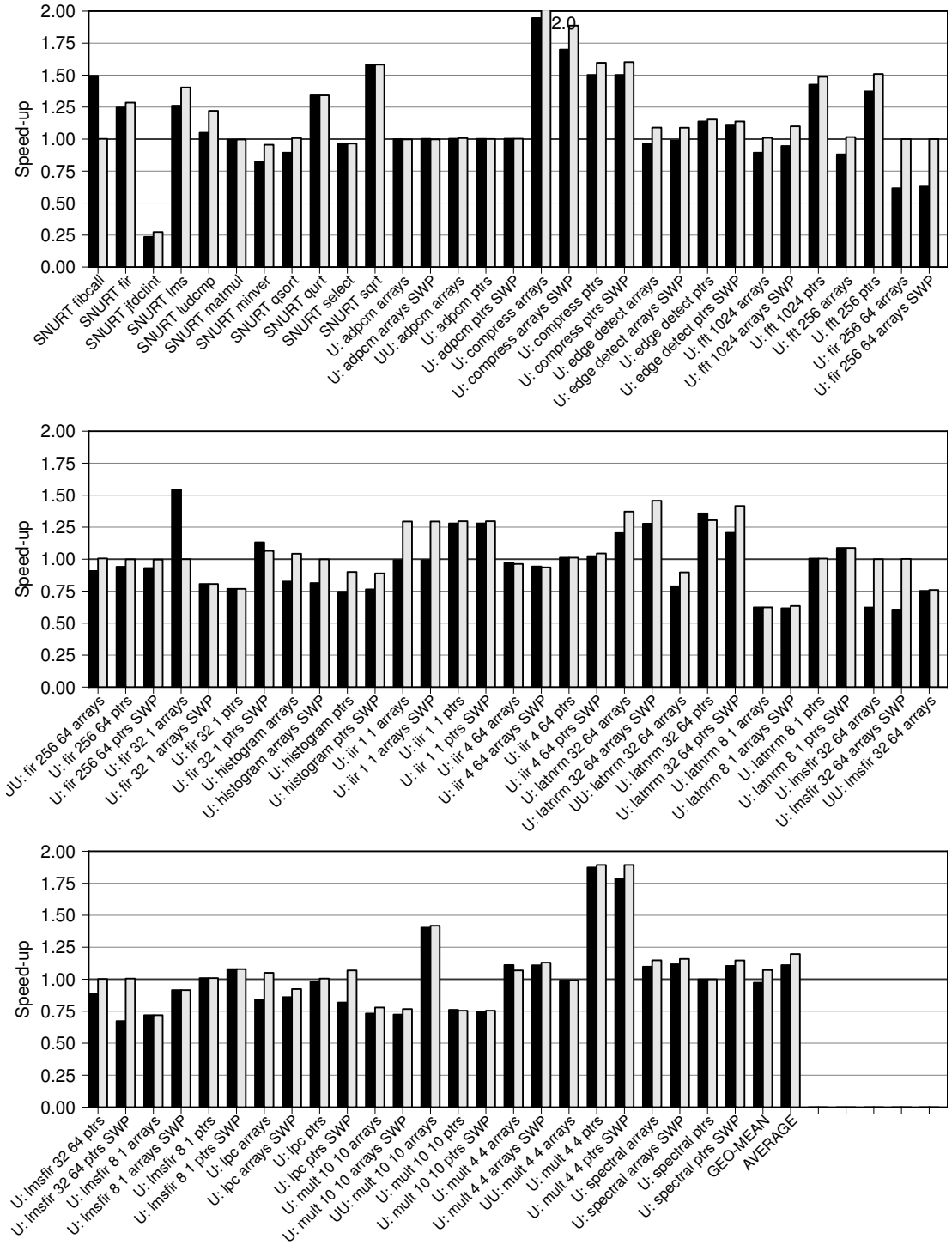


Figure B.6 (continued): *Speed-up obtainable with default mapping (left-hand bar) or when eliminating poor mappings (right-hand bar).*

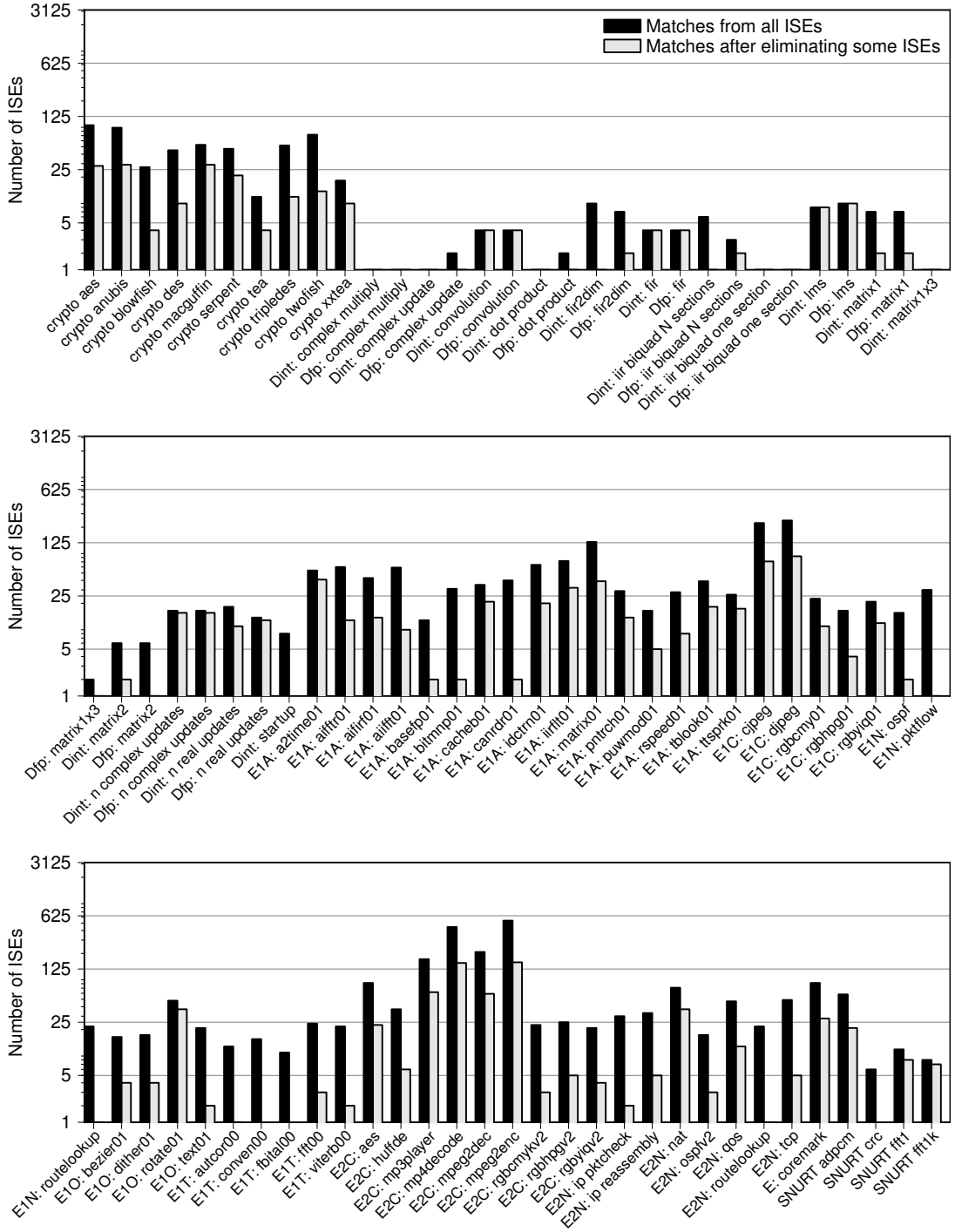


Figure B.7: Note: this is the full version of figure 4.9(b) on page 53. *An evaluation of eliminating poor mappings in* MapISE. (Continued on the next page.)

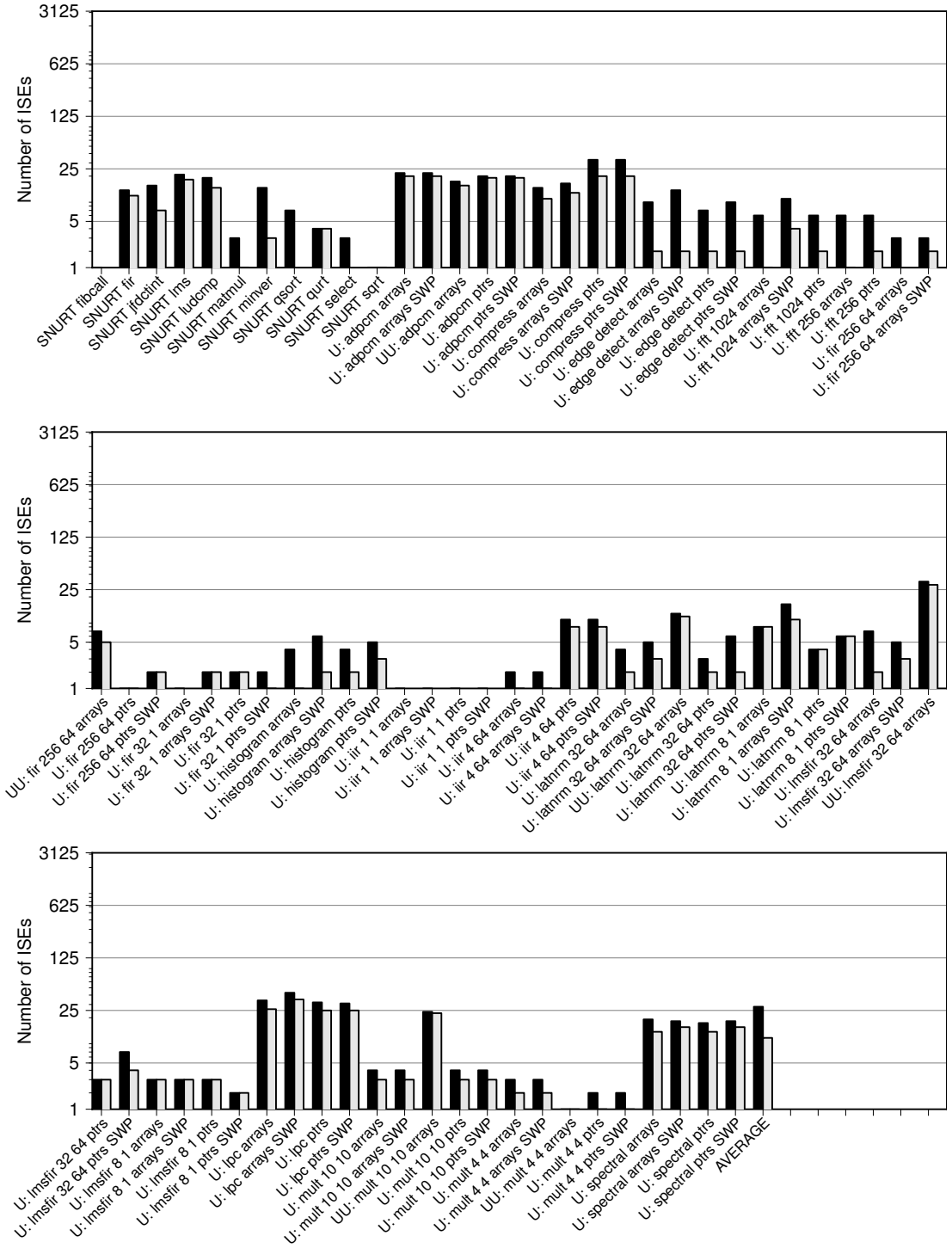


Figure B.7 (continued): *The number of sites where extension instructions are found for default mapping (left-hand bar) or when eliminating poor mappings (right-hand bar).*

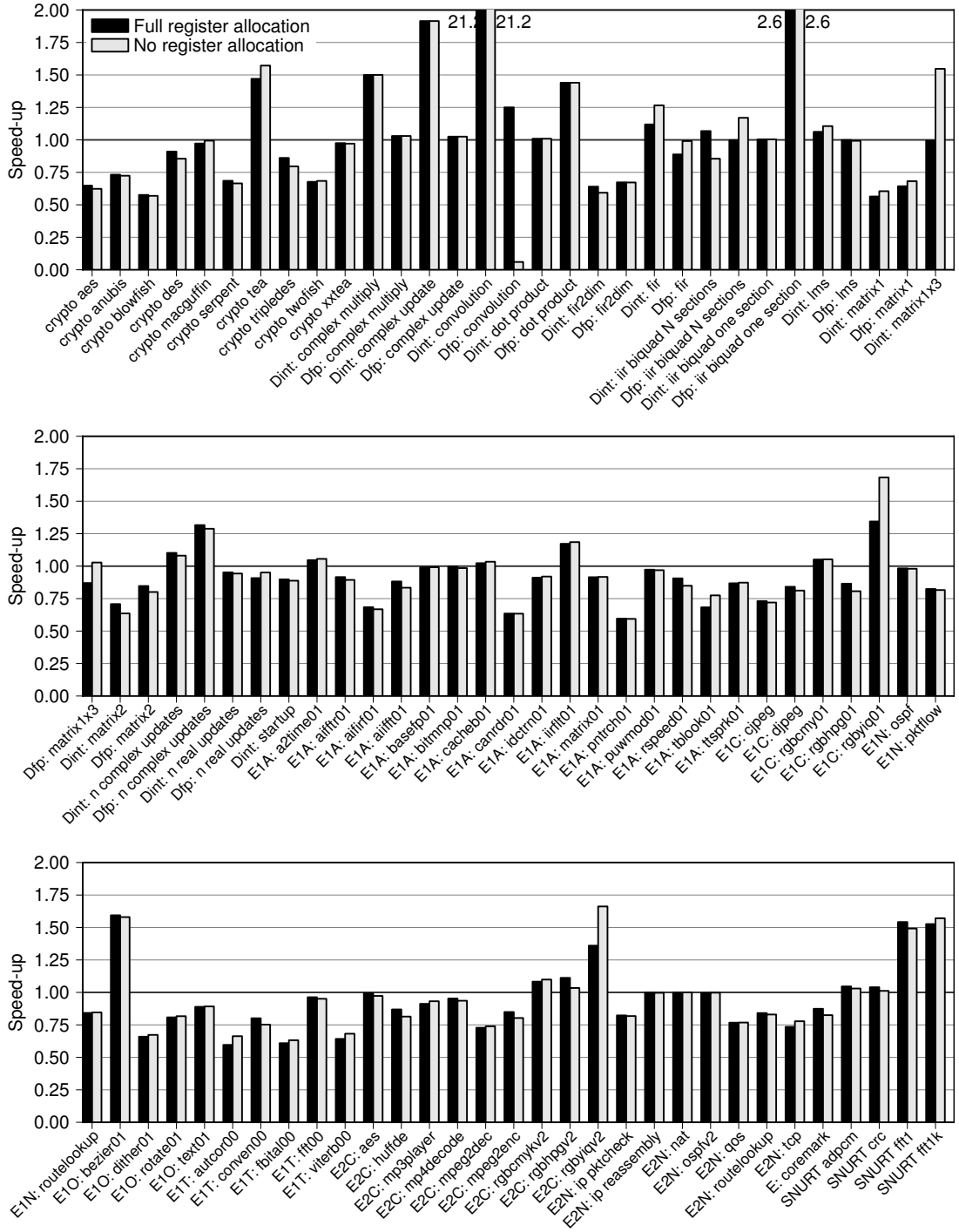


Figure B.8: Note: this is the full version of figure 4.10(a) on page 54. *A comparison of different register allocation modes.* (Continued on the next page.)

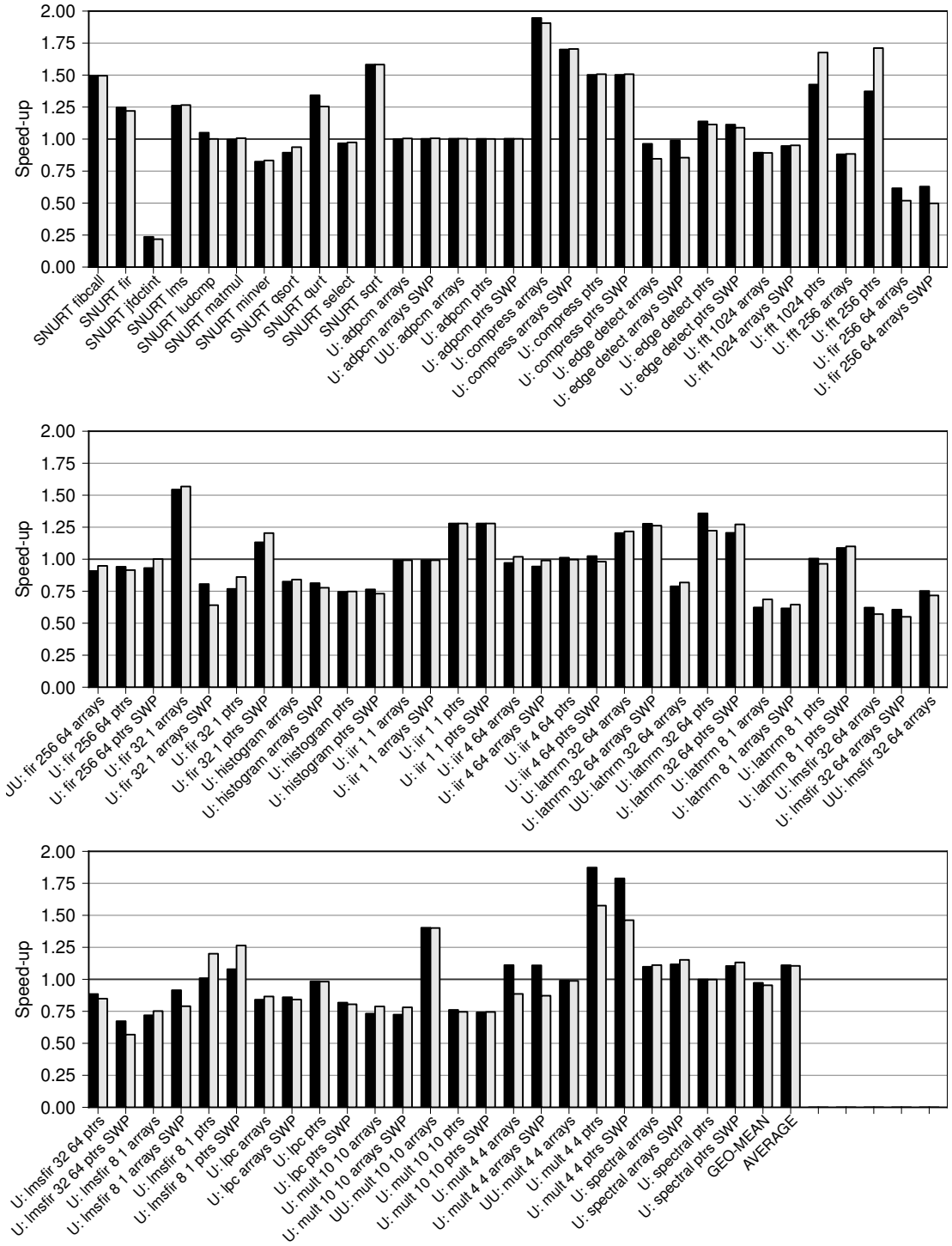


Figure B.8 (continued): *The speed-ups obtained when using the default register allocation settings (left-hand bar) and when a single static allocation is used for every extension instruction (right-hand bar).*

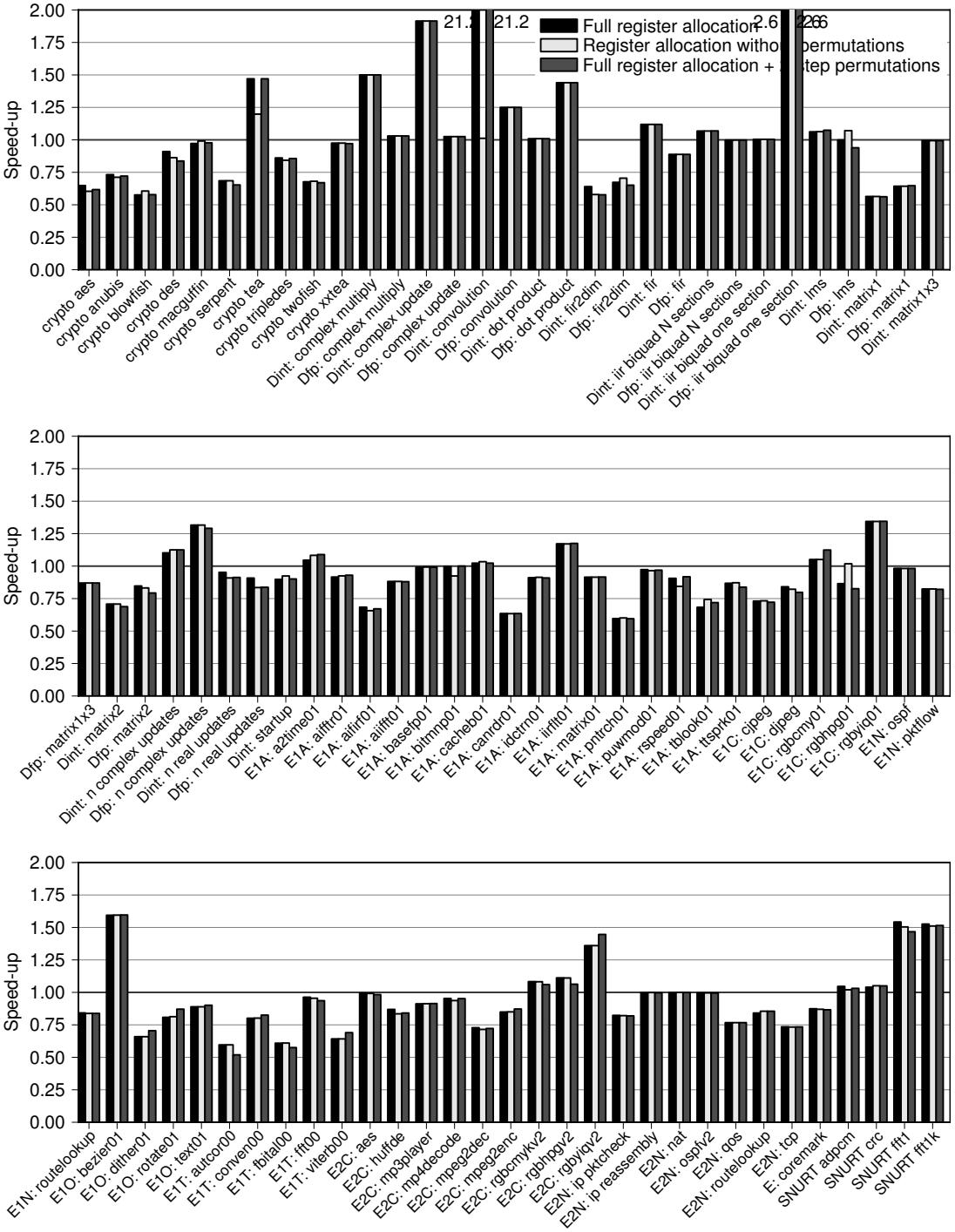


Figure B.9: Note: this is the full version of figure 4.10(b) on page 54. *A comparison of different register allocation modes. This chart compares the speed-ups possible when using the permutation units.* (Continued on the next page.)

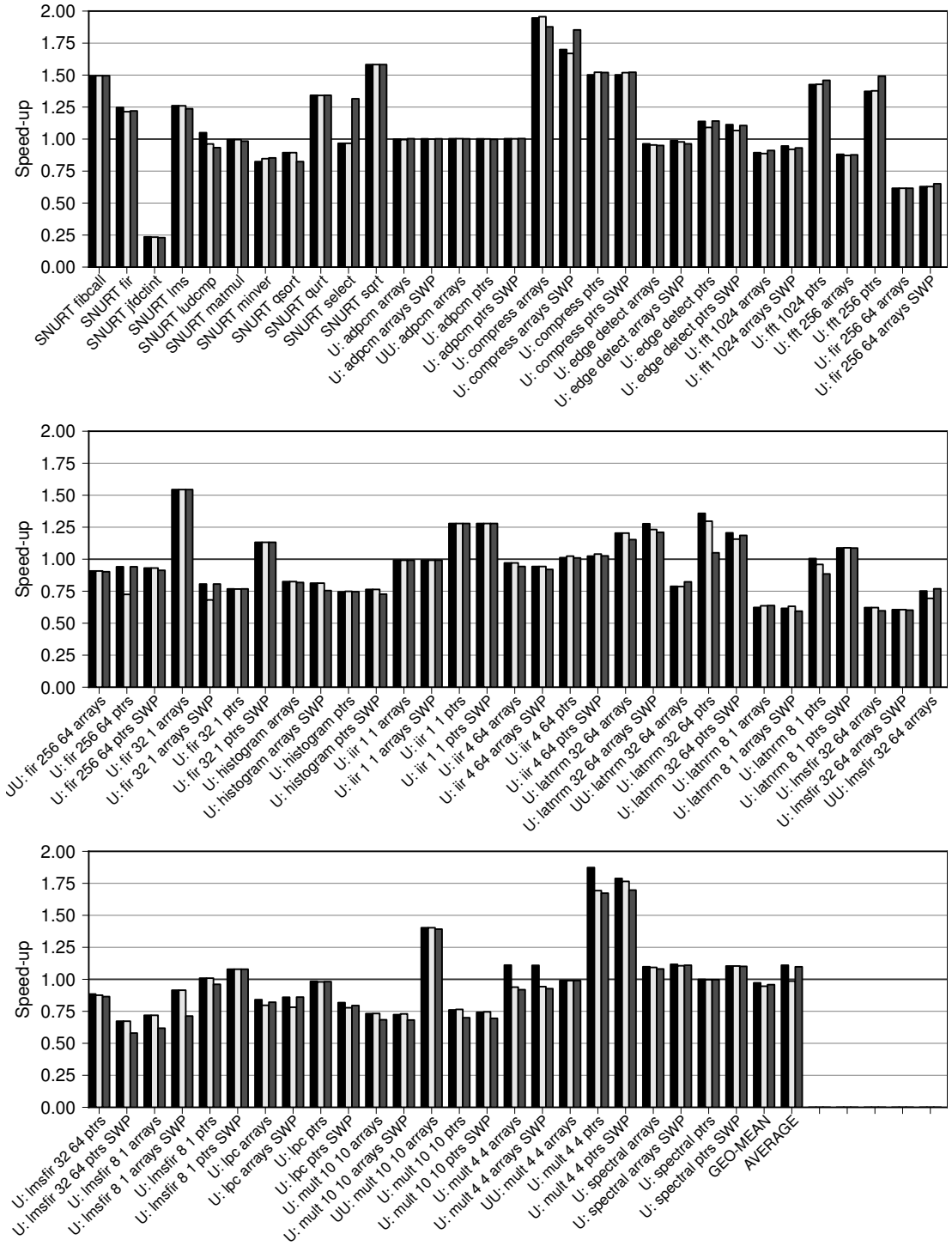


Figure B.9 (continued): *The default setting is to use one-step permutations (left bar), this is compared with never using the permutation unit (middle bar) or using two-step permutations (right-hand bar).*

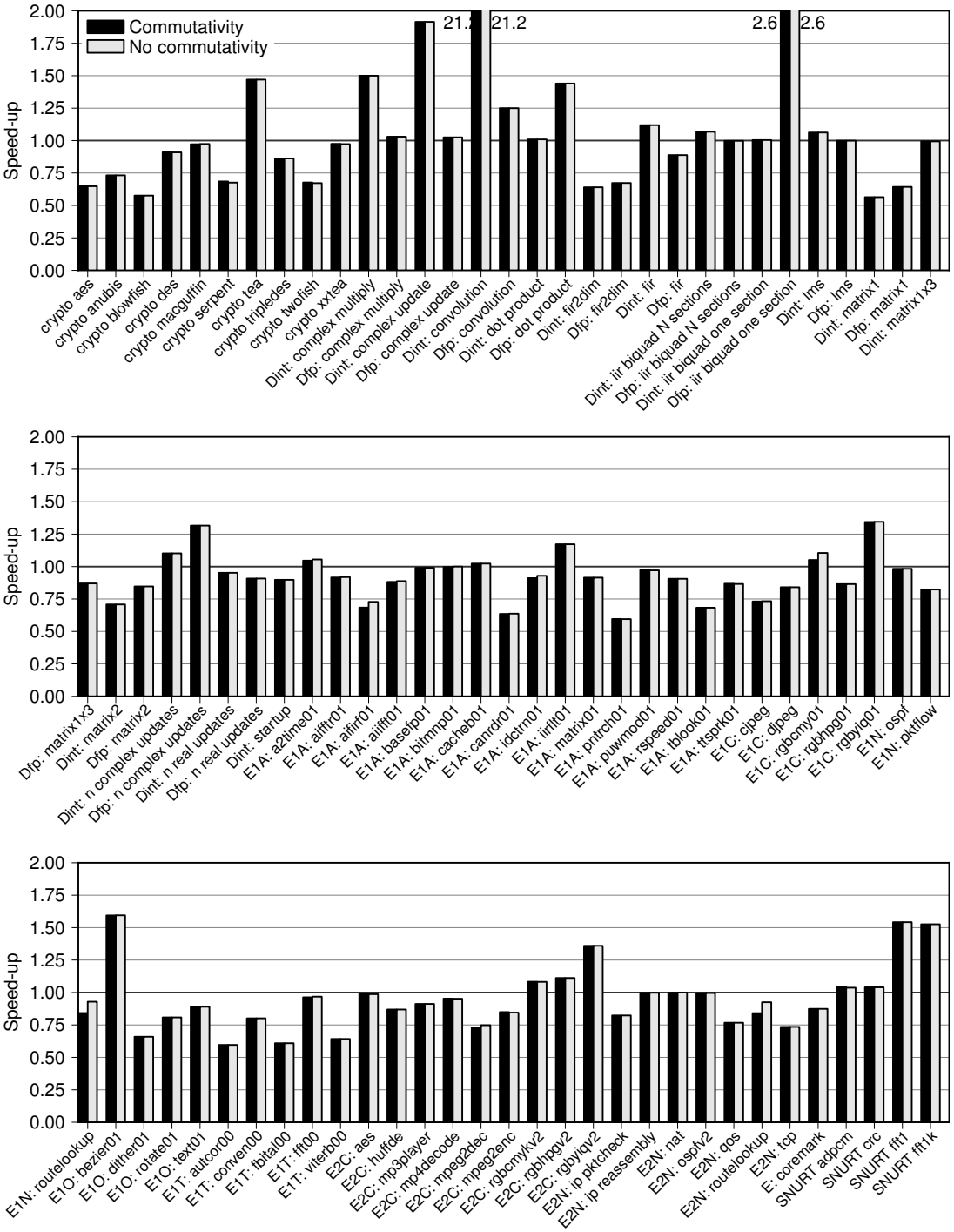


Figure B.10: Note: this is the full version of figure 4.11(a) on page 57. *A comparison of different commutativity options. This chart shows the speed-ups obtained when exploiting commutativity in* MapISE*. The left bar allows commutativity in both integer and* (Continued on the next page.)

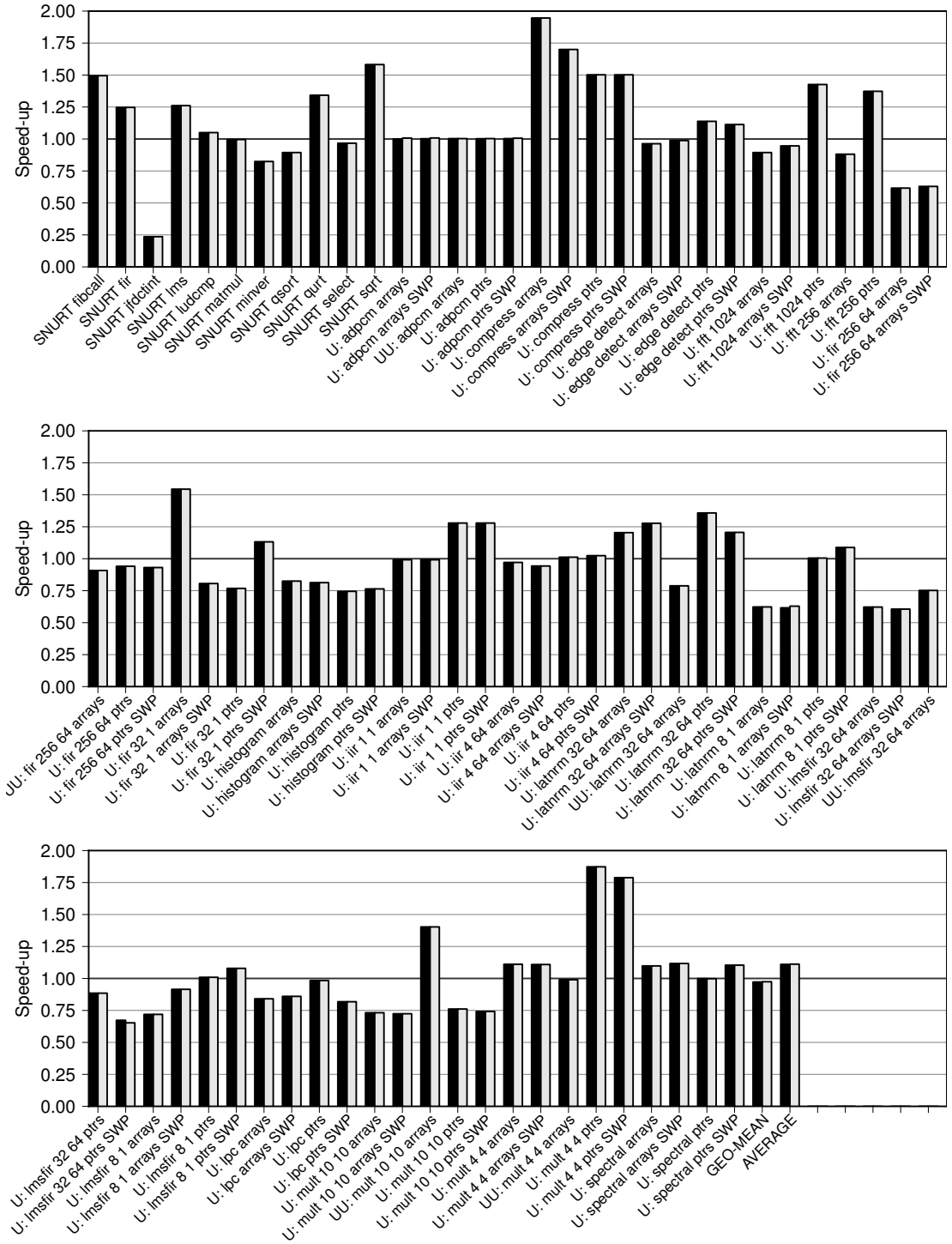


Figure B.10 (continued): *floating point nodes for every arithmetic operation that is naturally commutative. The right bar does not consider commutativity at all.*

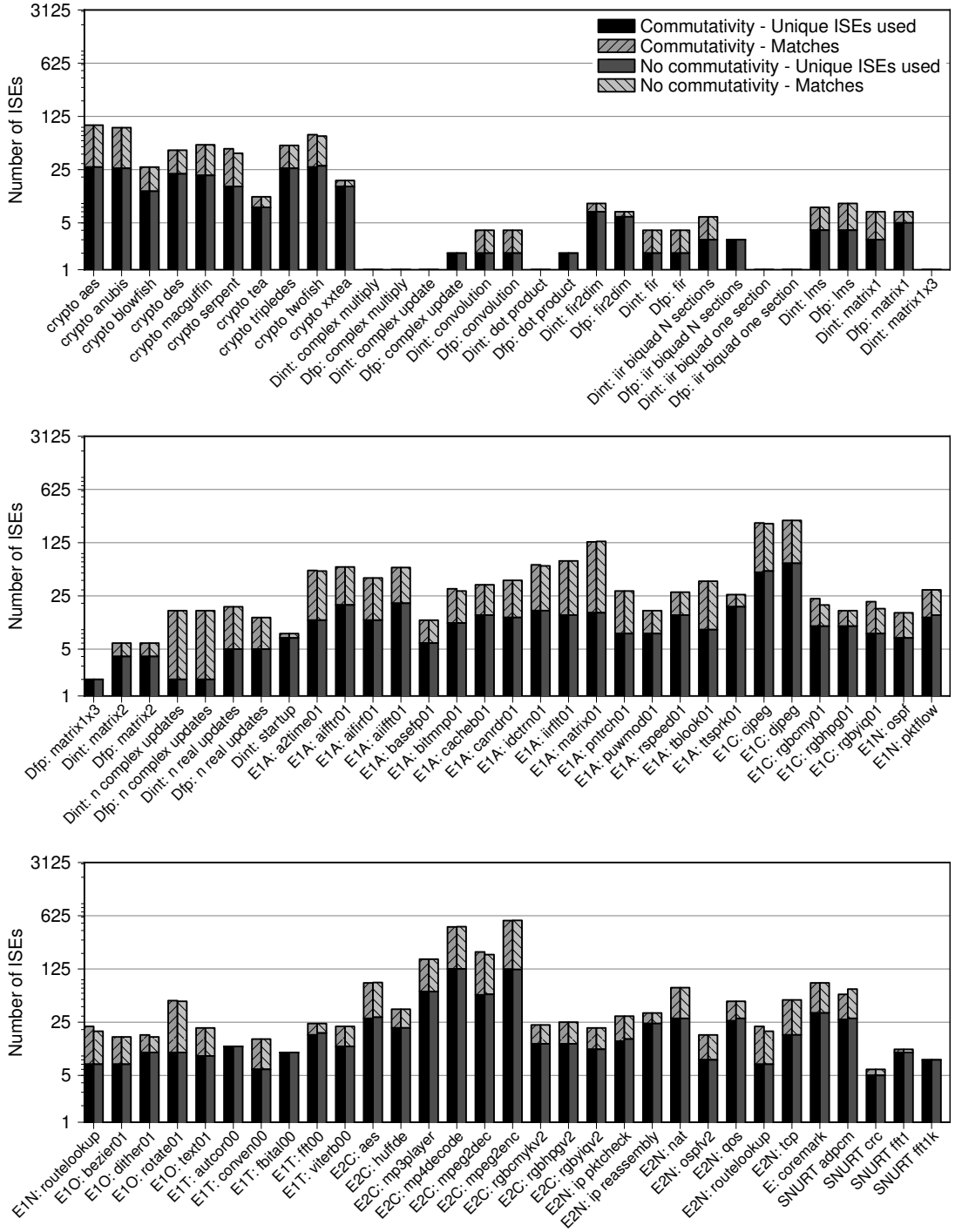


Figure B.11: Note: this is the full version of figure 4.11(b) on page 57. *A comparison of different commutativity options. This chart is similar to figure 4.7(b) except that every* (Continued on the next page.)

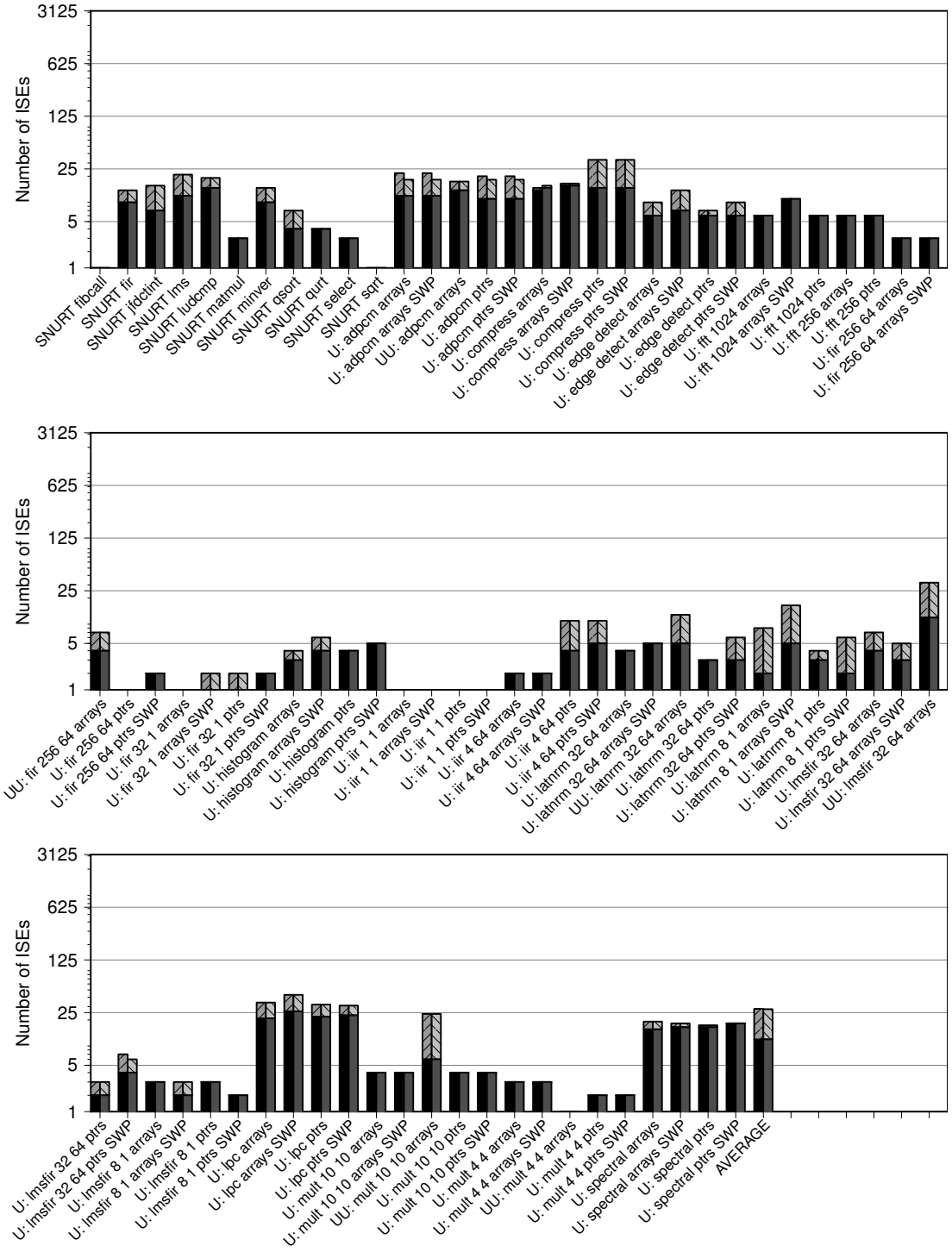


Figure B.11 (continued): *stacked bar relates to the number of extension instructions that* MapISE *can use. The lower bars are the number of unique extension instructions used, the upper bars are the total number of sites that extension instructions could be used.*

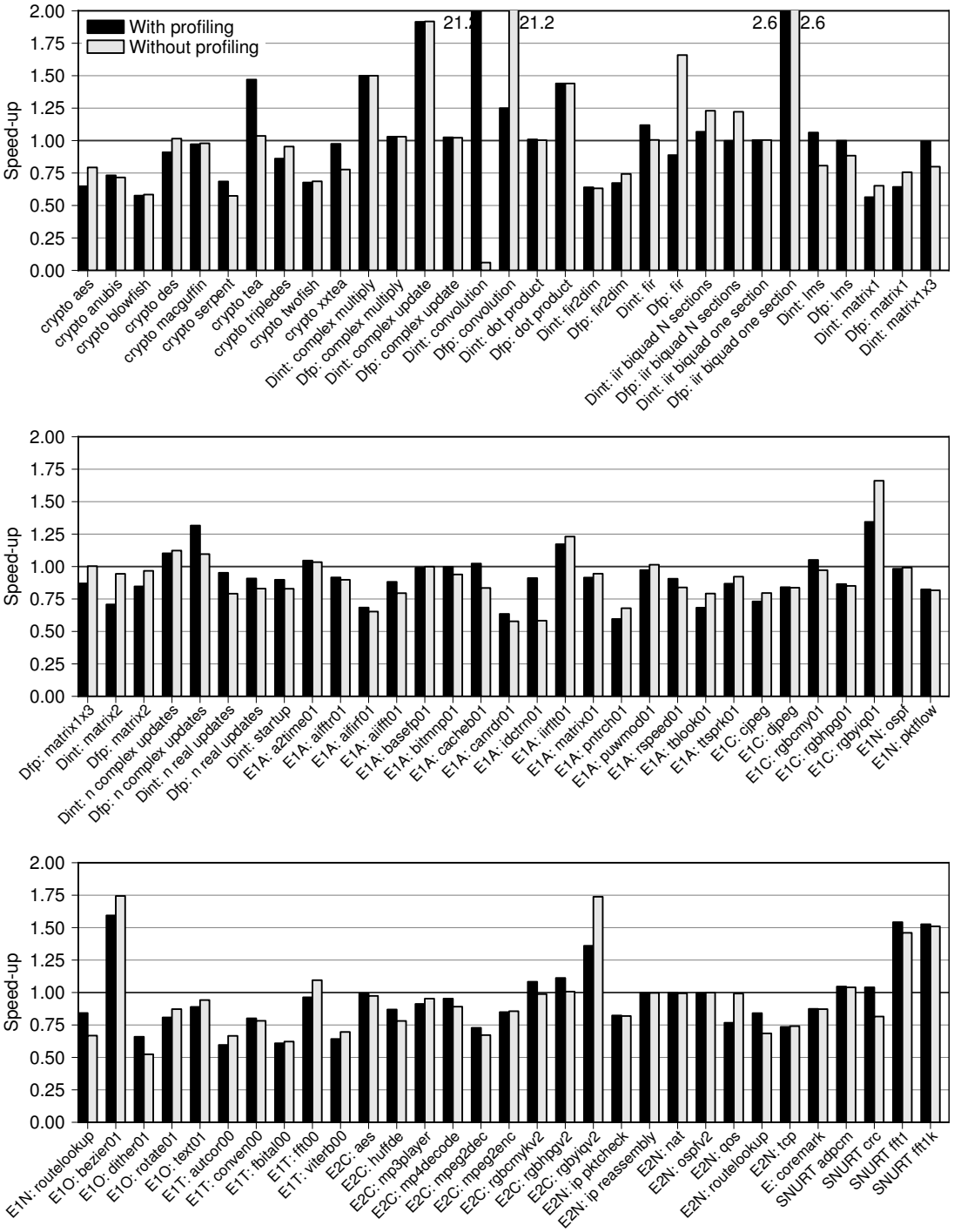


Figure B.12: Note: this is the full version of figure 4.12(a) on page 58. *An evaluation of* MapISE *when the compiler mode changes between extension instruction generation and exploitation. Withholding profiling data affects the decisions made by* GCC's *optimiser.* (Continued on the next page.)

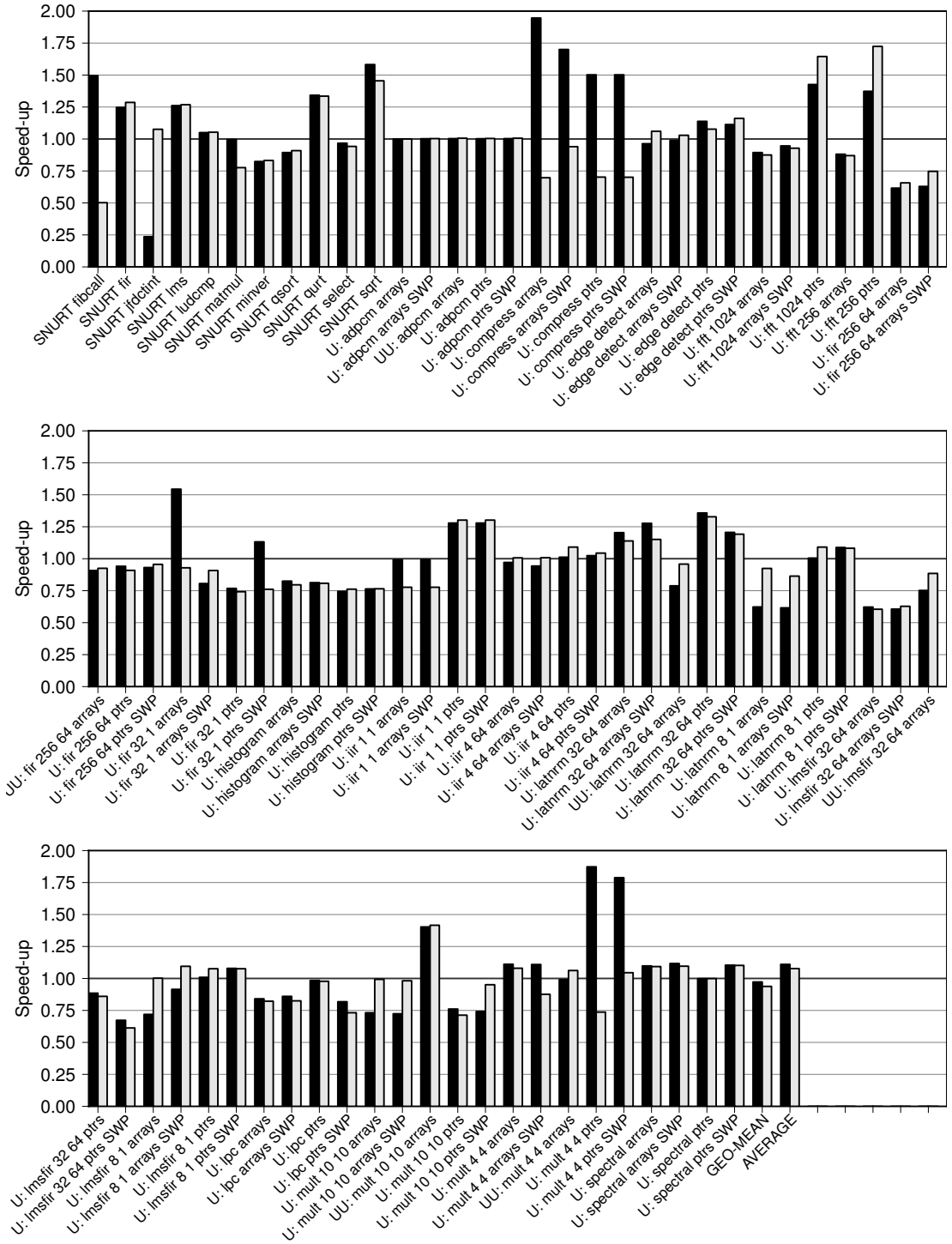


Figure B.12 (continued): *Speed-ups obtained from the default (left-hand bar) and the alternative modes (right-hand bar).*

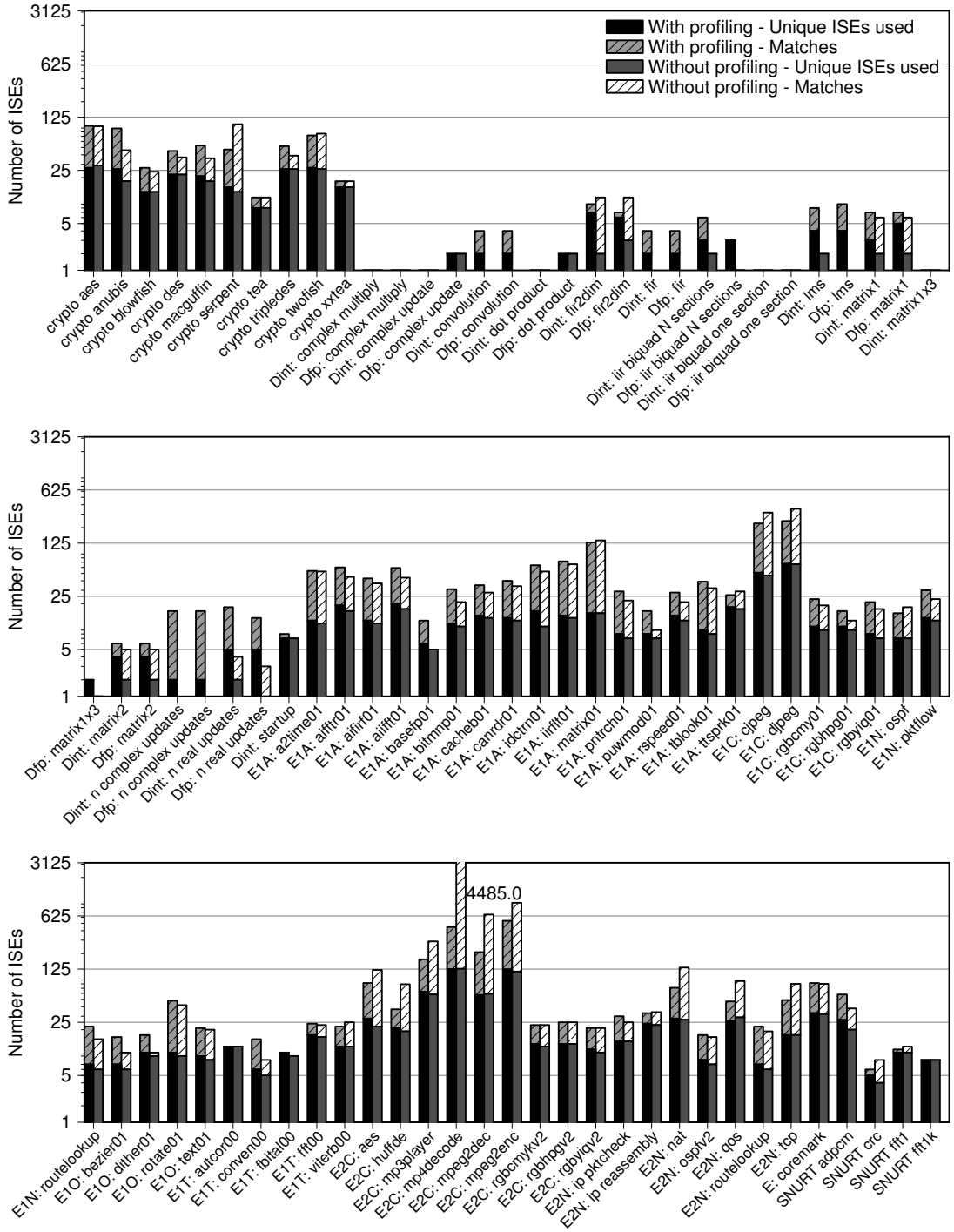


Figure B.13: Note: this is the full version of figure 4.12(b) on page 58. *An evaluation of* MapISE *when the compiler mode changes between extension instruction generation and exploitation. Withholding profiling data affects the decisions made by* GCC's *optimiser.* (Continued on the next page.)

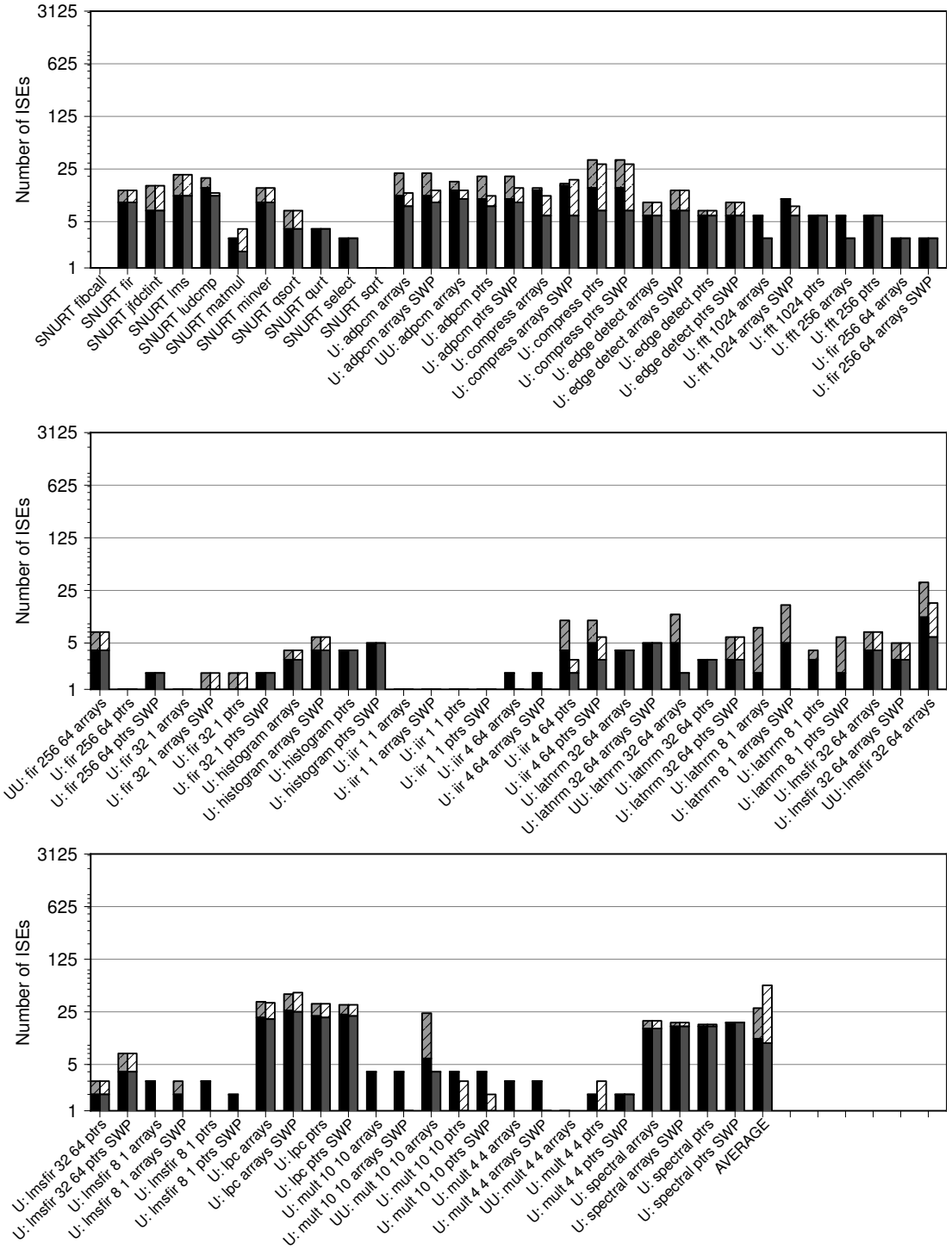


Figure B.13 (continued): *Mapping quality information for default (left-hand bar) and alternative modes (right-hand bar).*

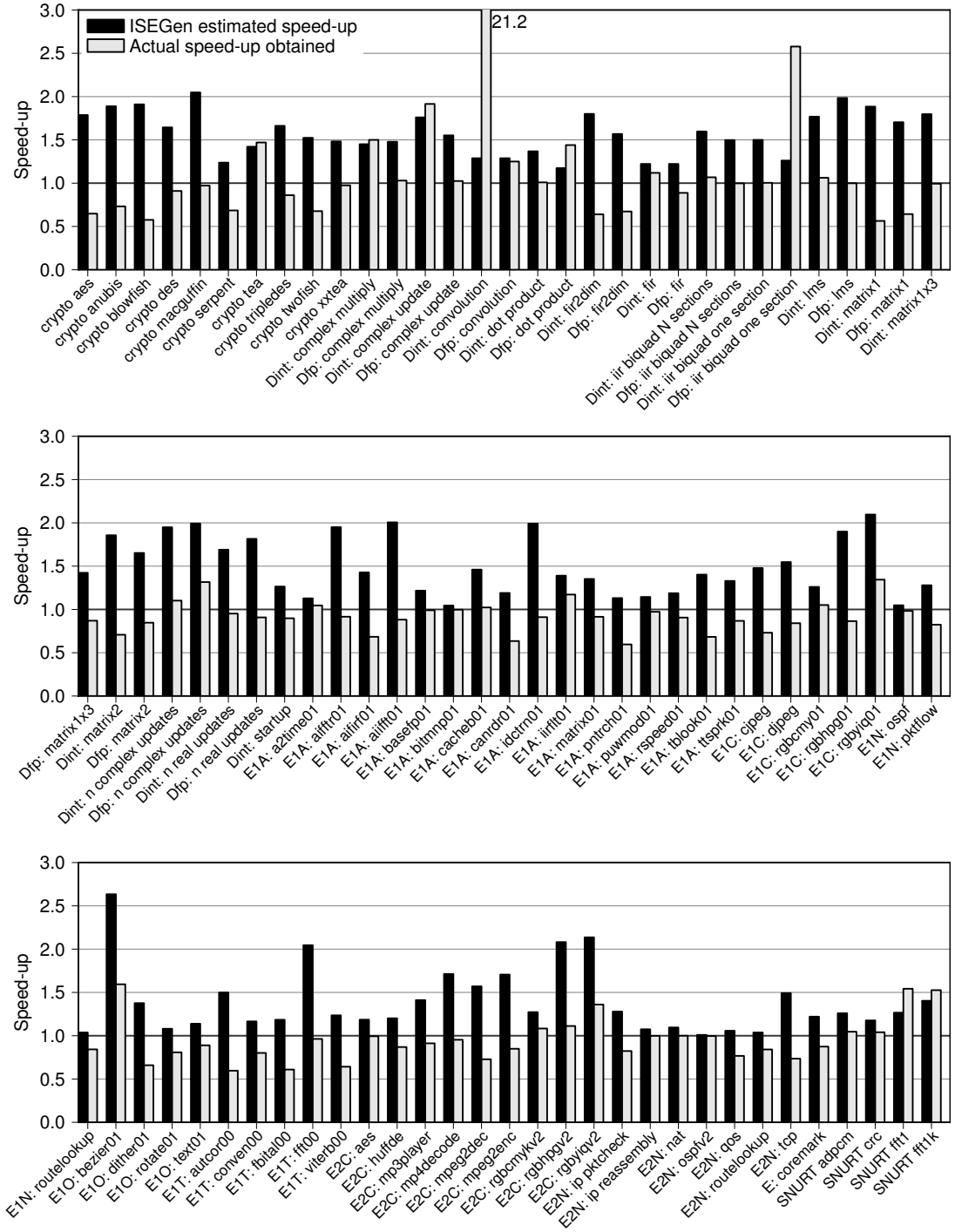


Figure B.14: Note: this is the full version of figure 4.18 on page 65. (Continued on the next page.)

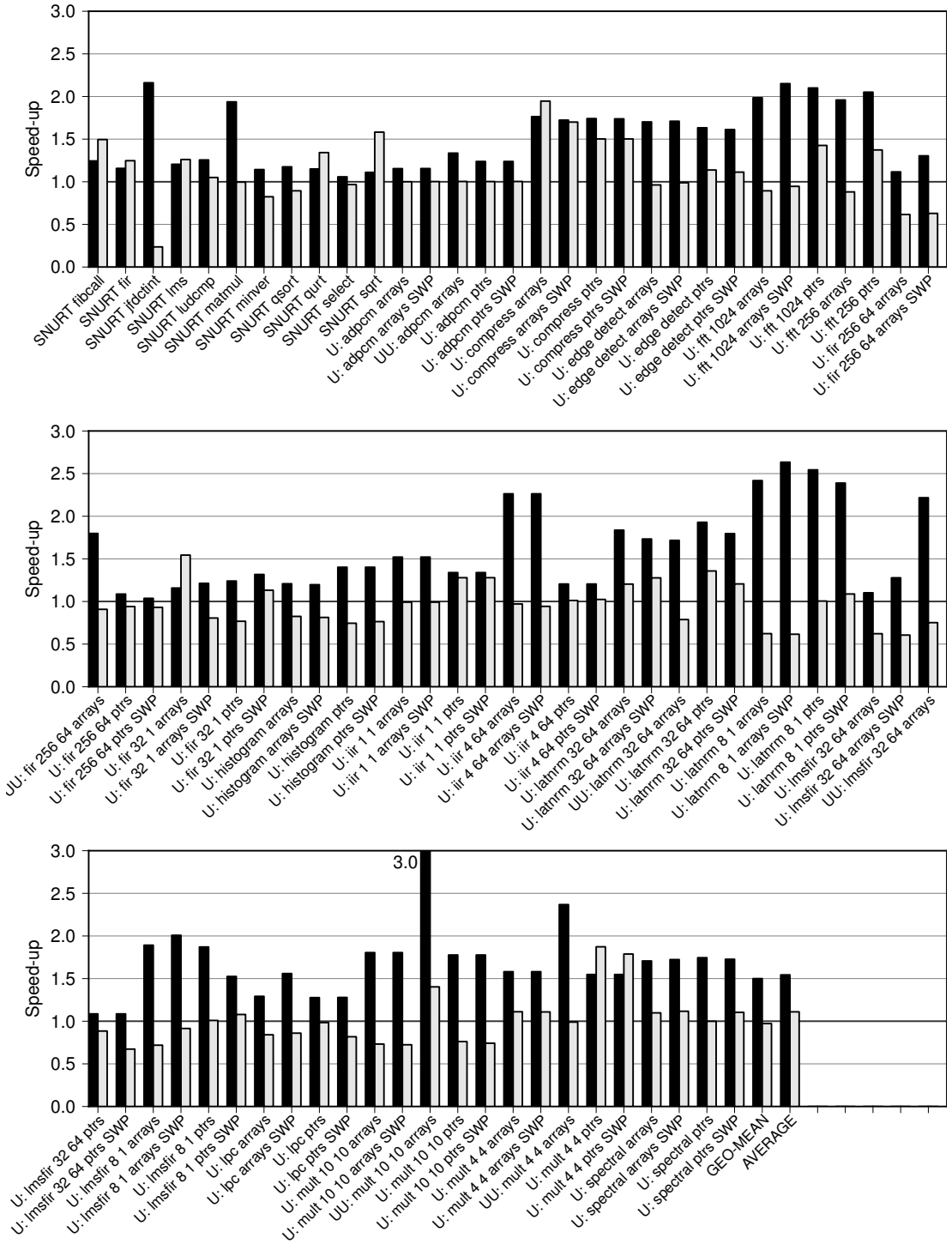


Figure B.14 (continued): *A comparison of* ISEGen's *estimated benefit from extension instructions and the benefit actually achieved by* MapISE.

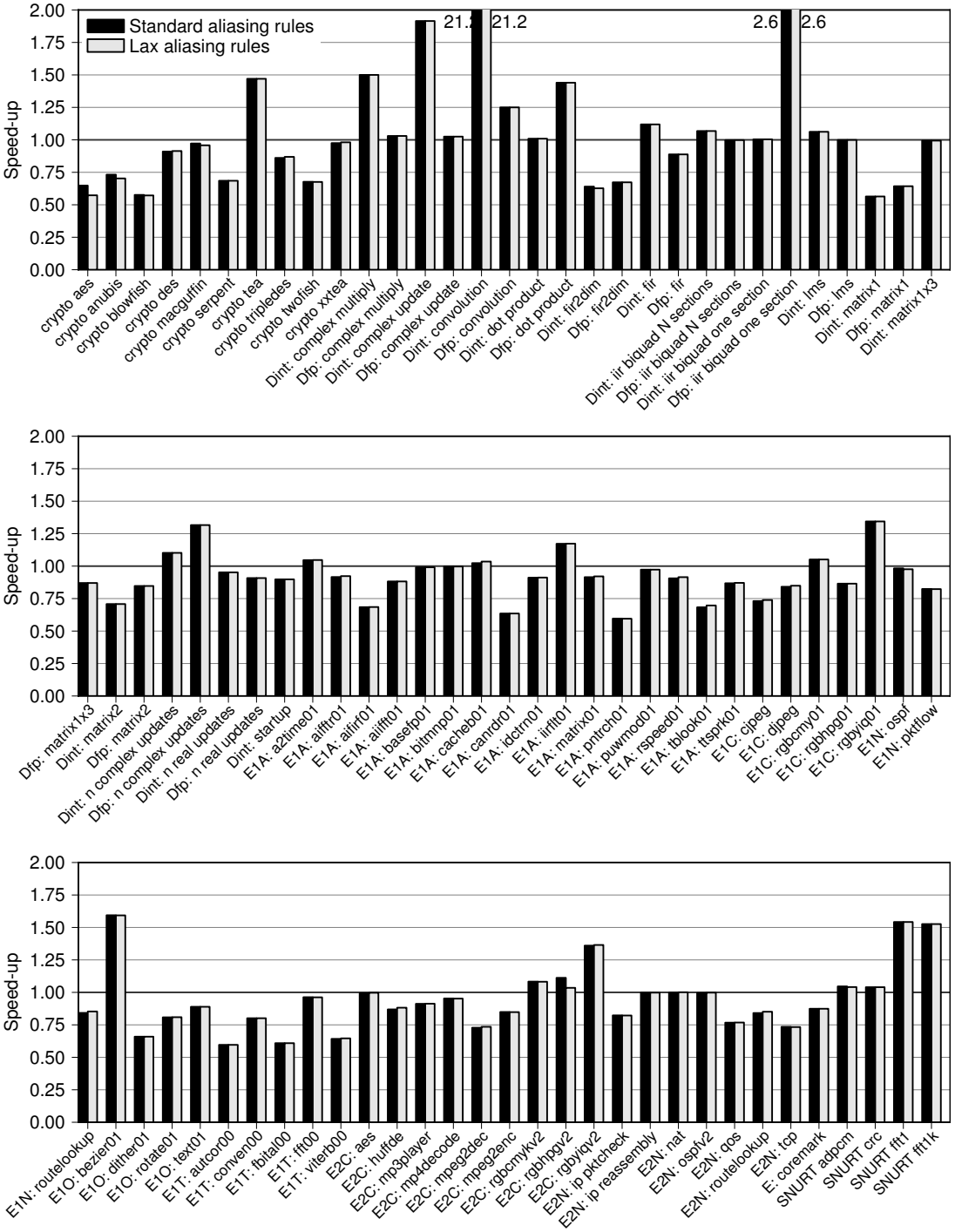


Figure B.15: Note: this is the full version of figure 4.19(a) on page 67. *An investigation of the effect of weakening the aliasing rules in* GCC *with regards to* MapISE. (Continued on the next page.)

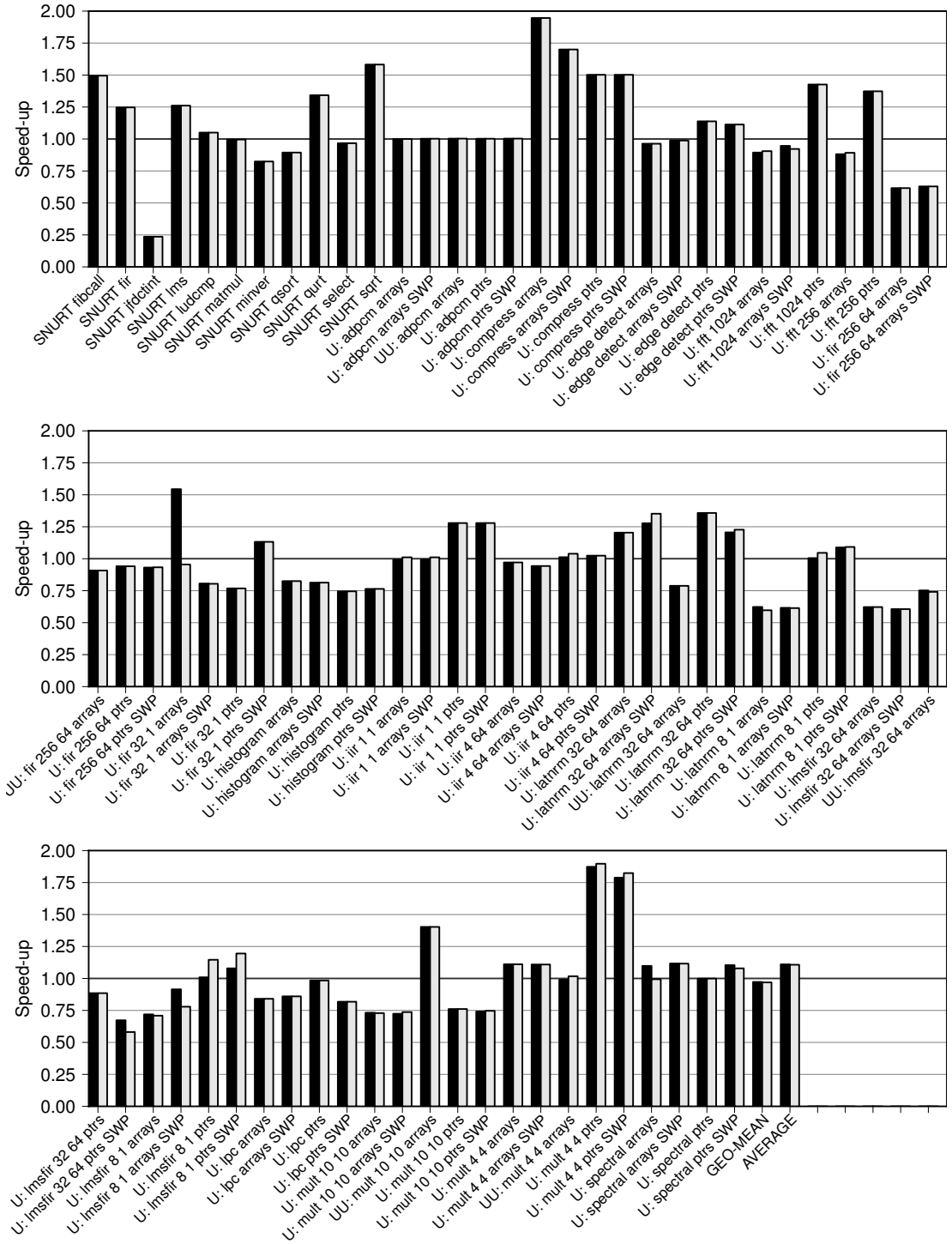


Figure B.15 (continued): *Speed-ups obtained with standard C aliasing rules (left-hand bars) or less strict rules (right-hand bars).*

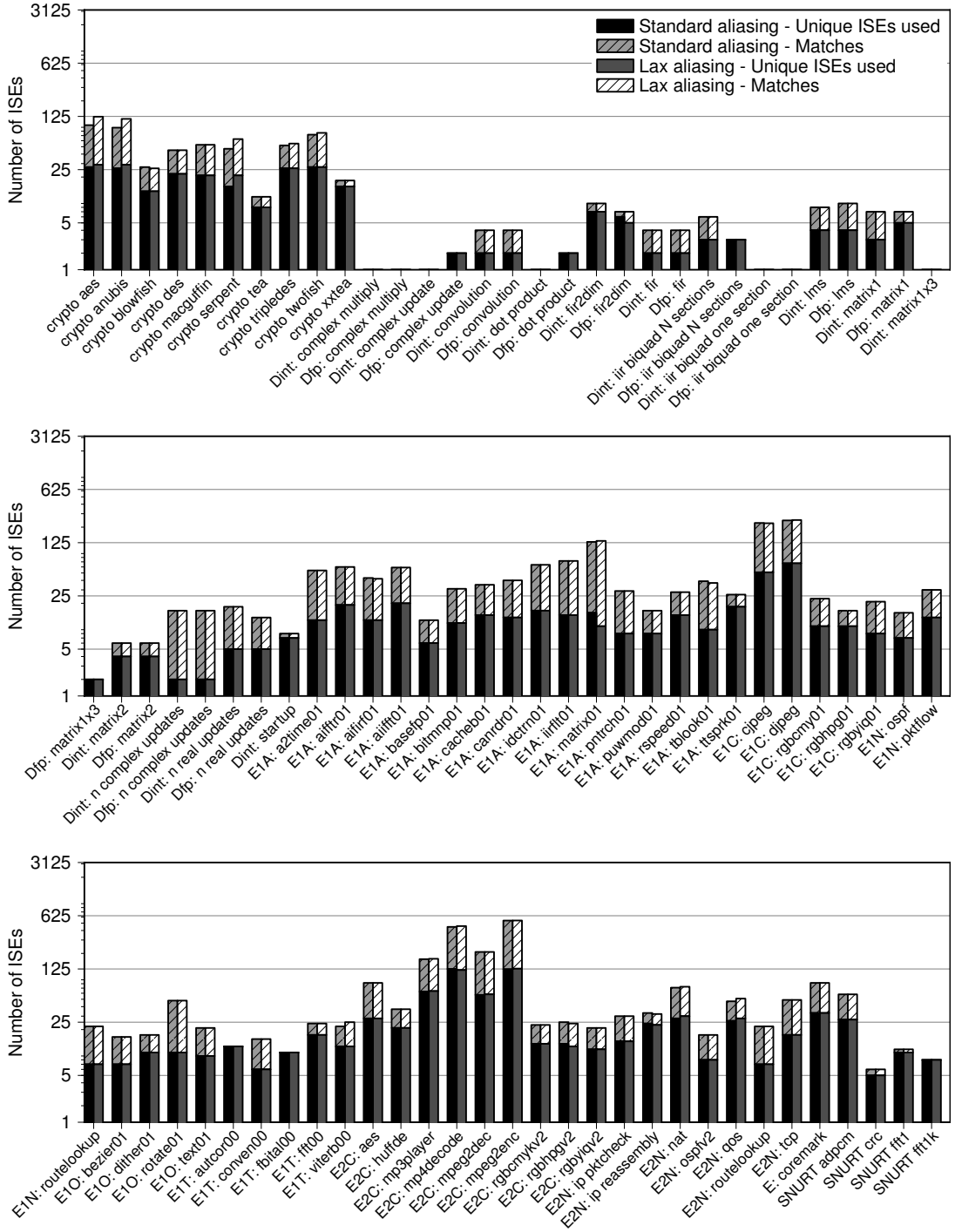


Figure B.16: Note: this is the full version of figure 4.19(b) on page 67. *An investigation of the effect of weakening the aliasing rules in* GCC *with regards to* MapISE. (Continued on the next page.)

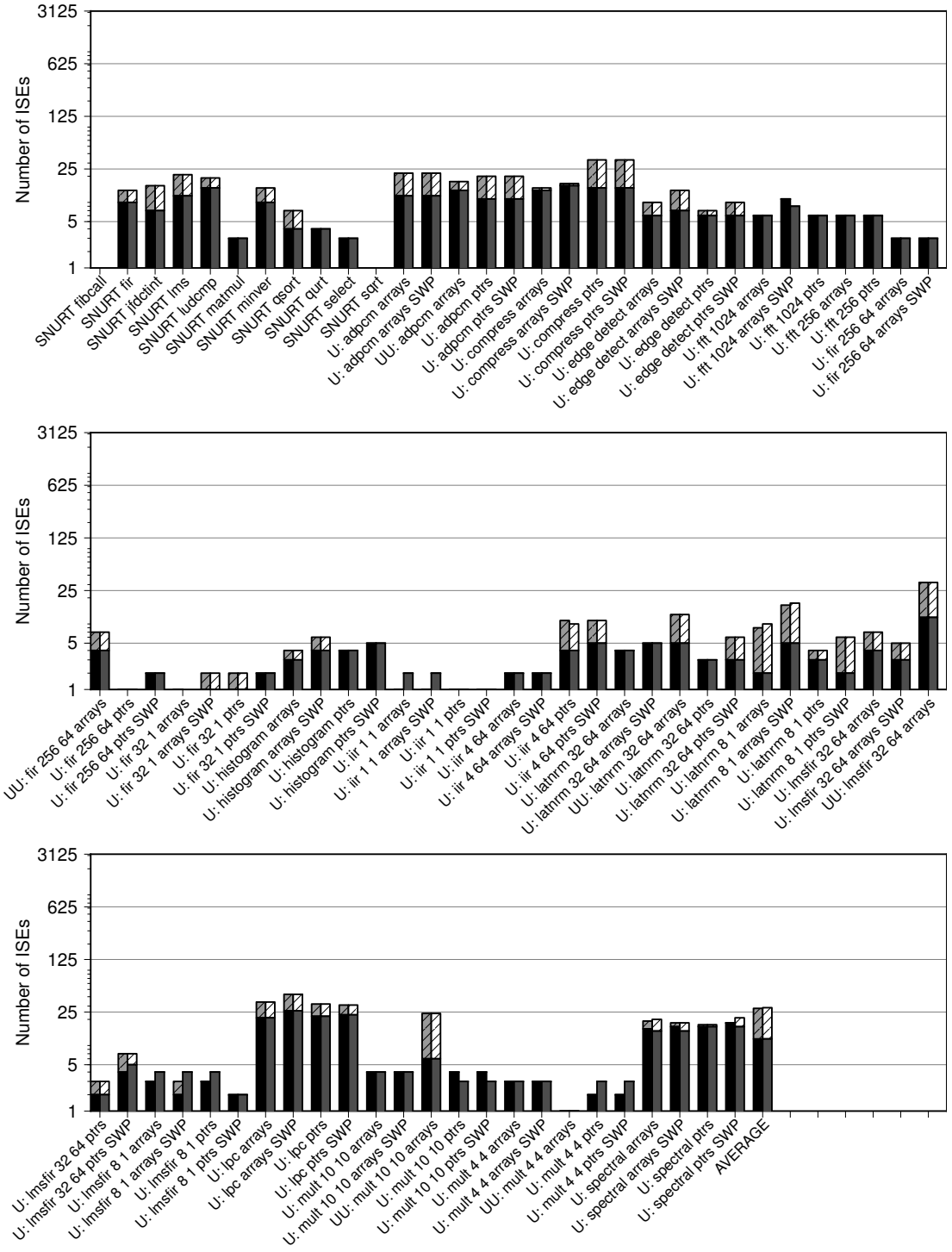


Figure B.16 (continued): *Mapping quality information.*

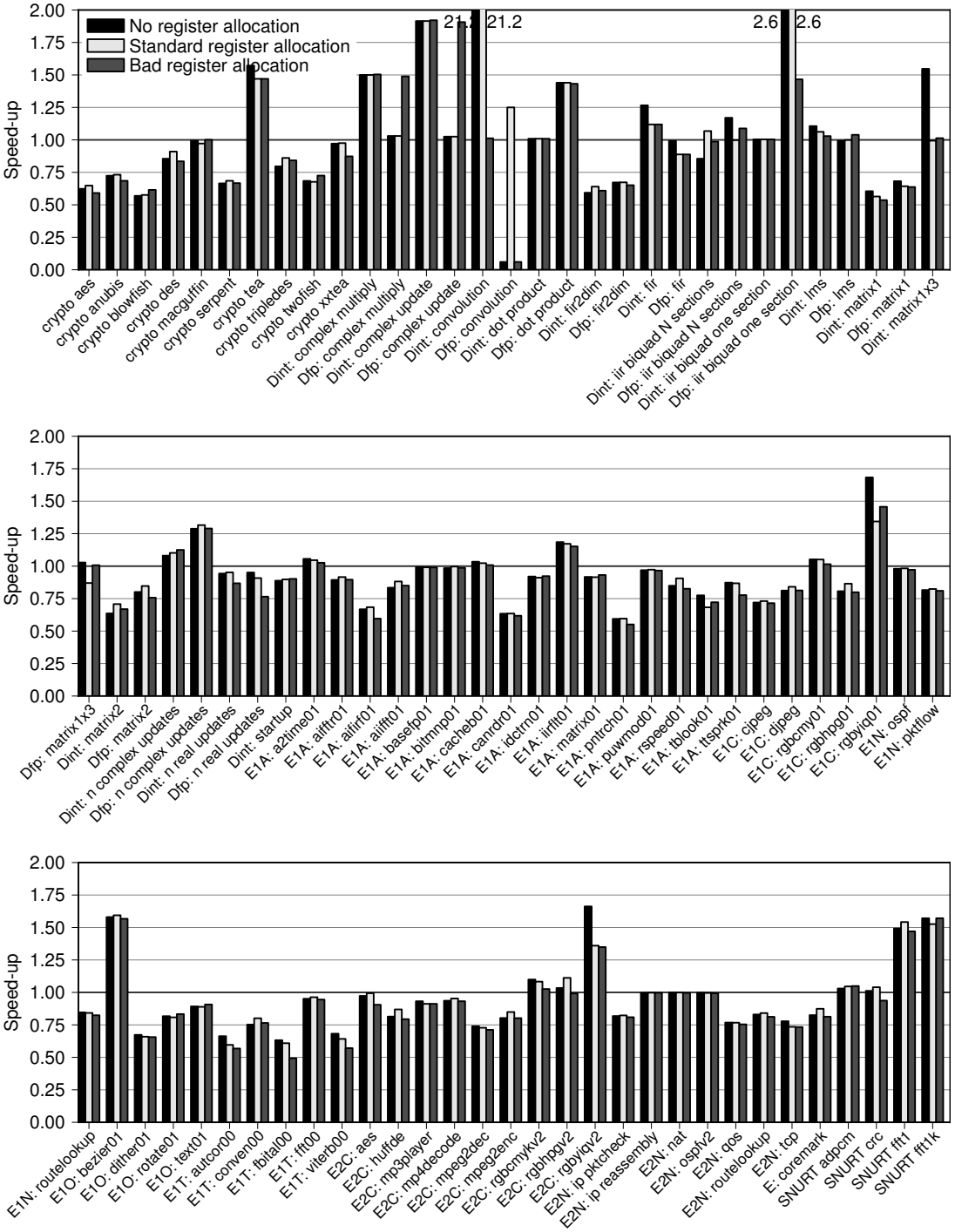


Figure B.17: Note: this is the full version of figure 4.20 on page 69. (Continued on the next page.)

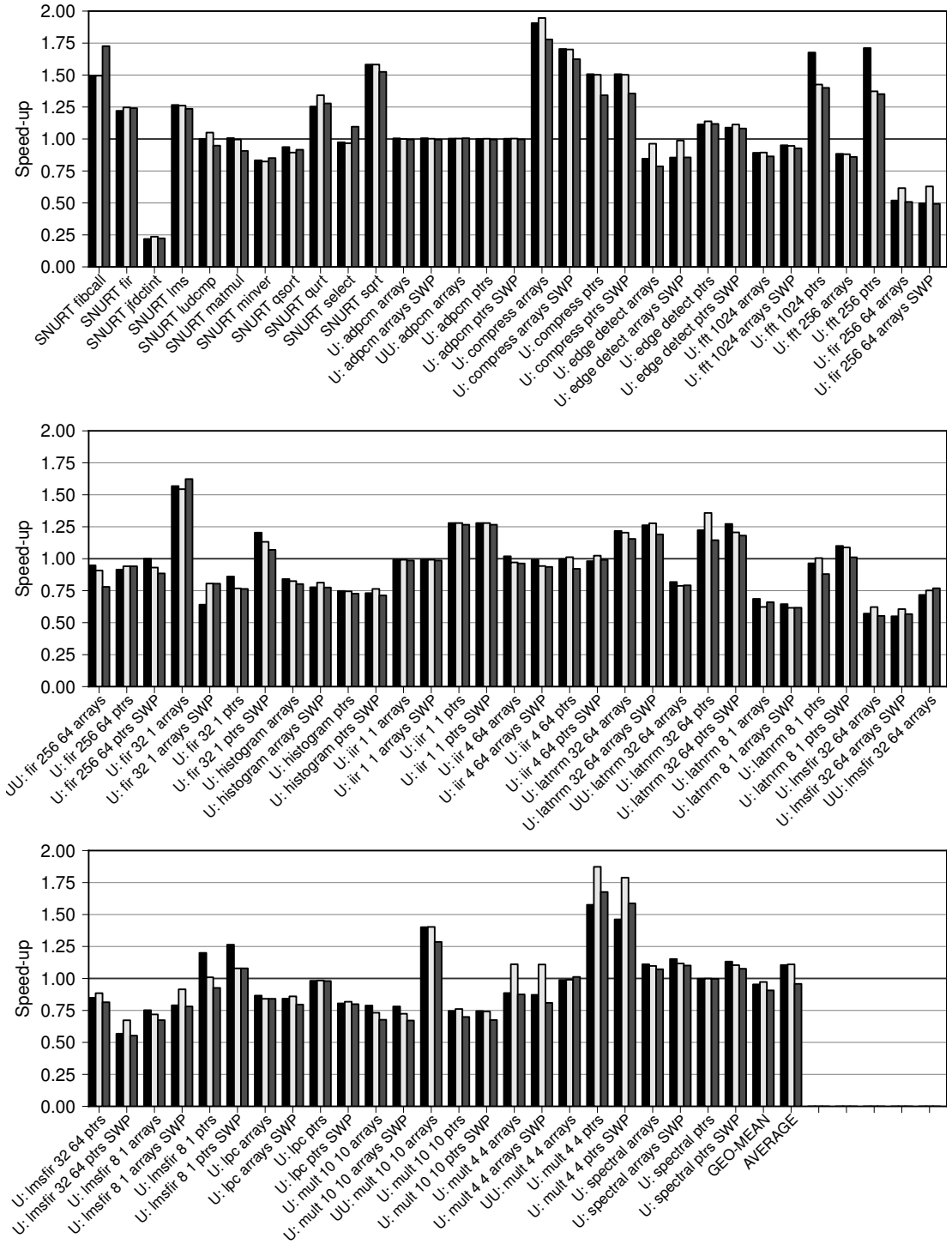


Figure B.17 (continued): *The speed-ups obtained when using deliberately bad register allocation heuristics.*

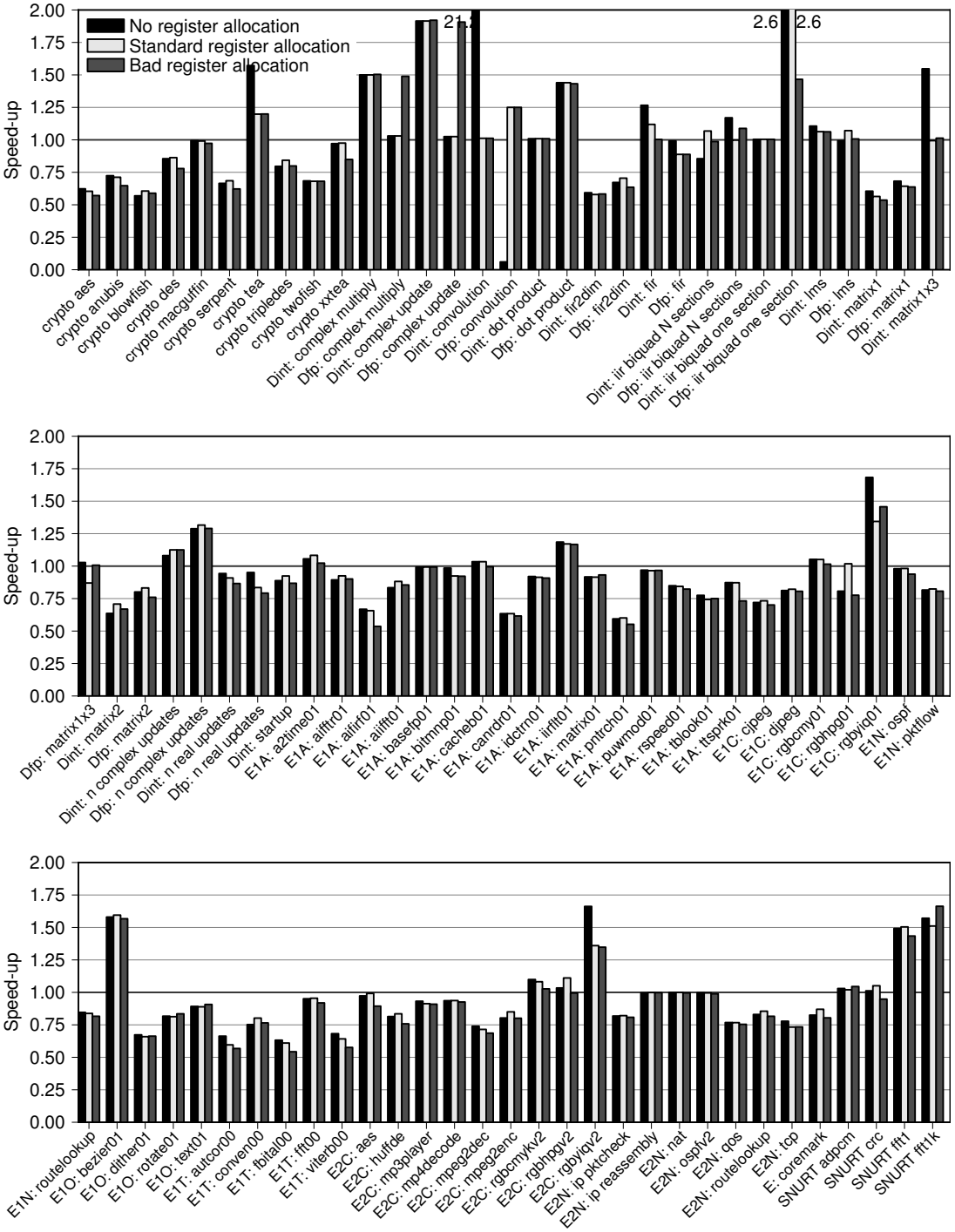


Figure B.18: Note: this is the full version of figure 4.21(a) on page 70. *The speed-ups obtained when using deliberately bad register allocation heuristics with permutation variations.* (Continued on the next page.)

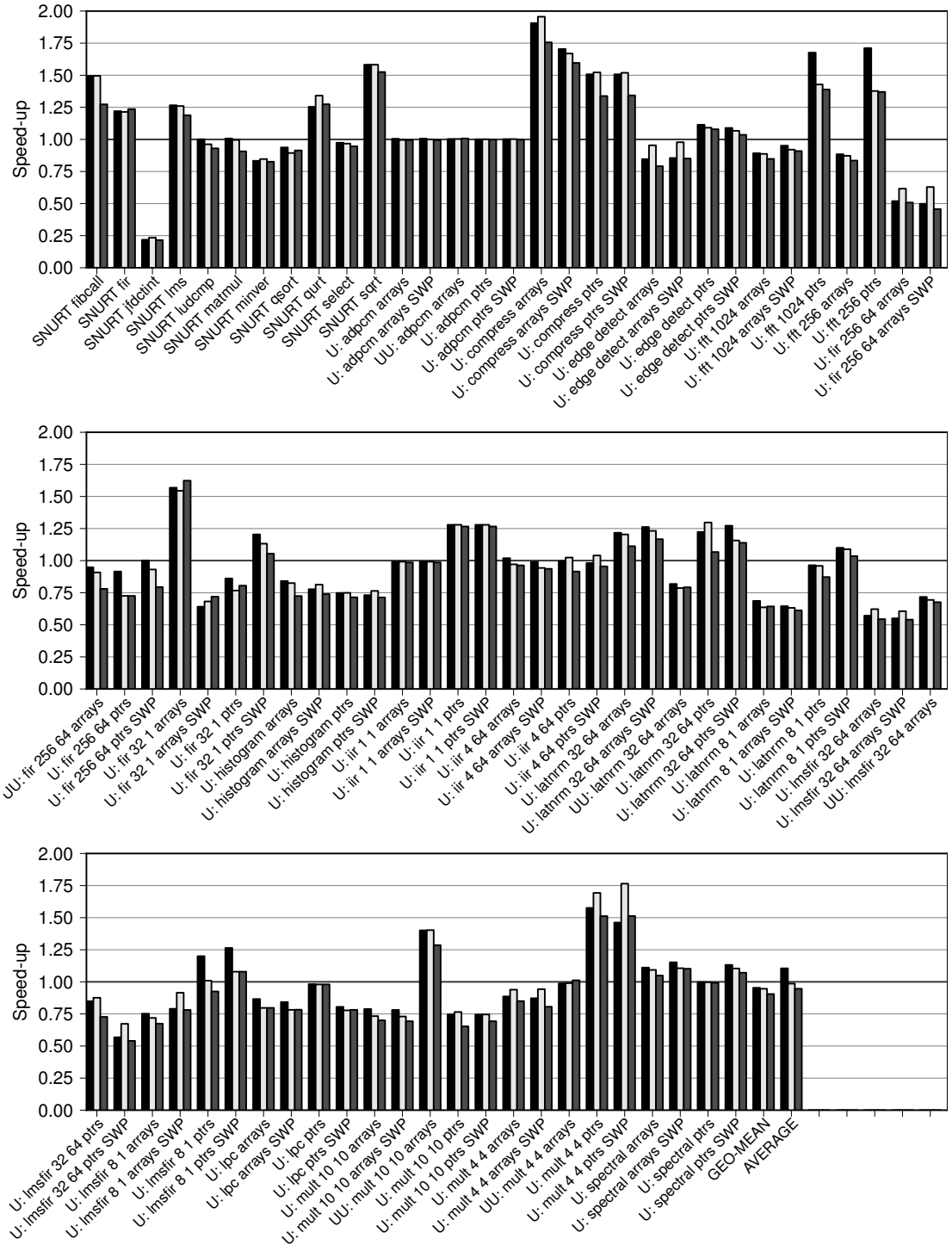


Figure B.18 (continued): *No permutations used.*

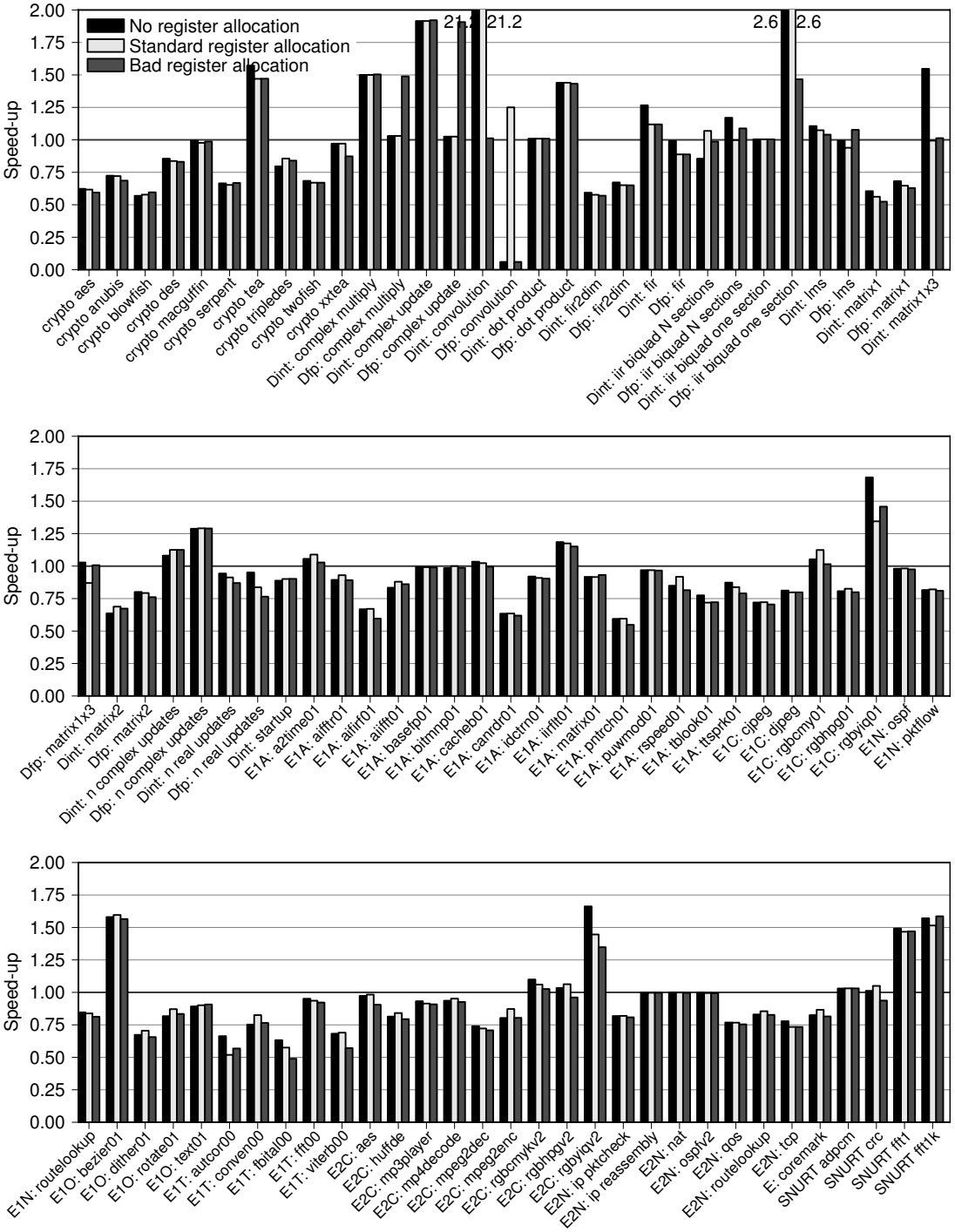


Figure B.19: Note: this is the full version of figure 4.21(b) on page 70. *The speed-ups obtained when using deliberately bad register allocation heuristics with permutation variations.* (Continued on the next page.)

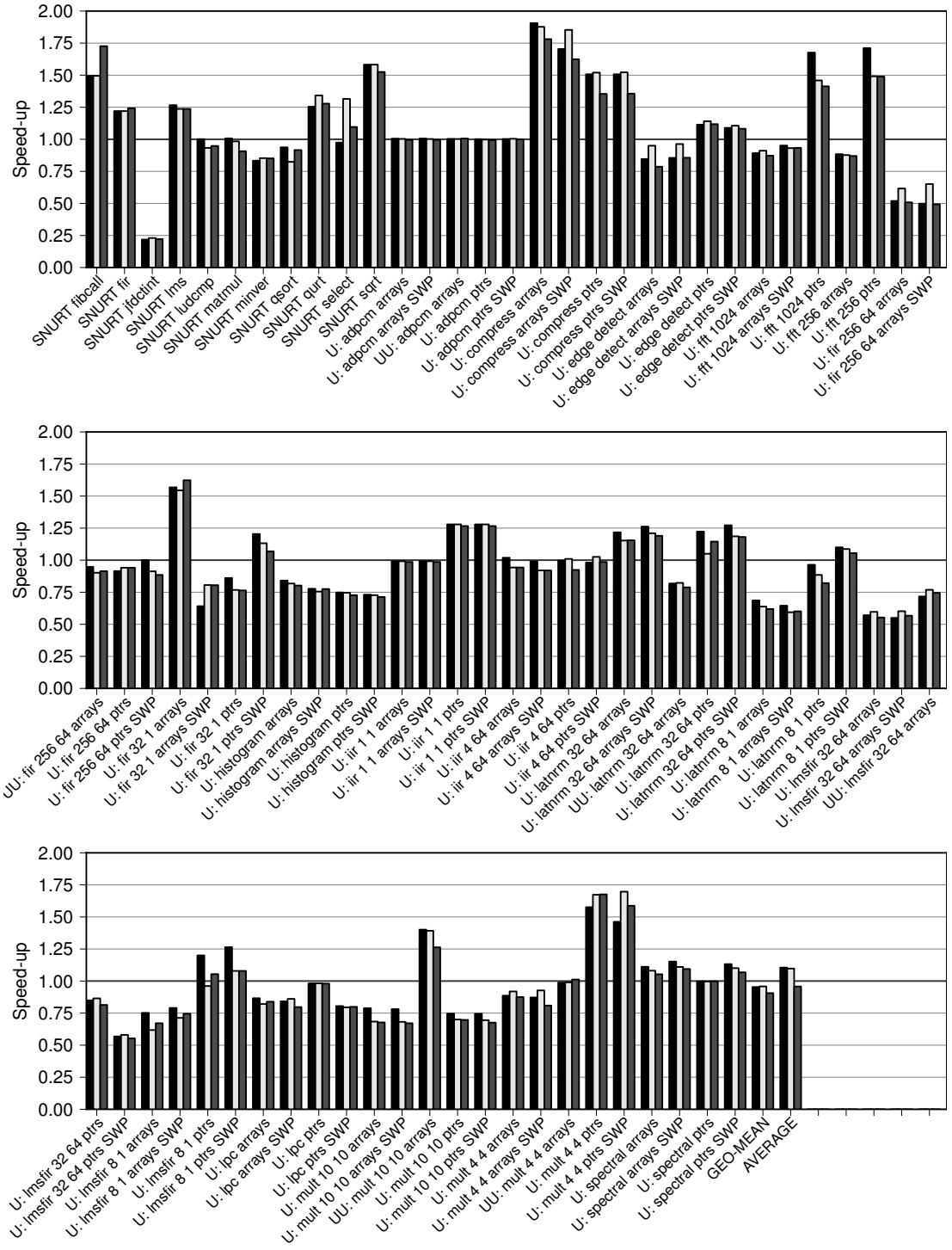


Figure B.19 (continued): *Two-step permutations used.*

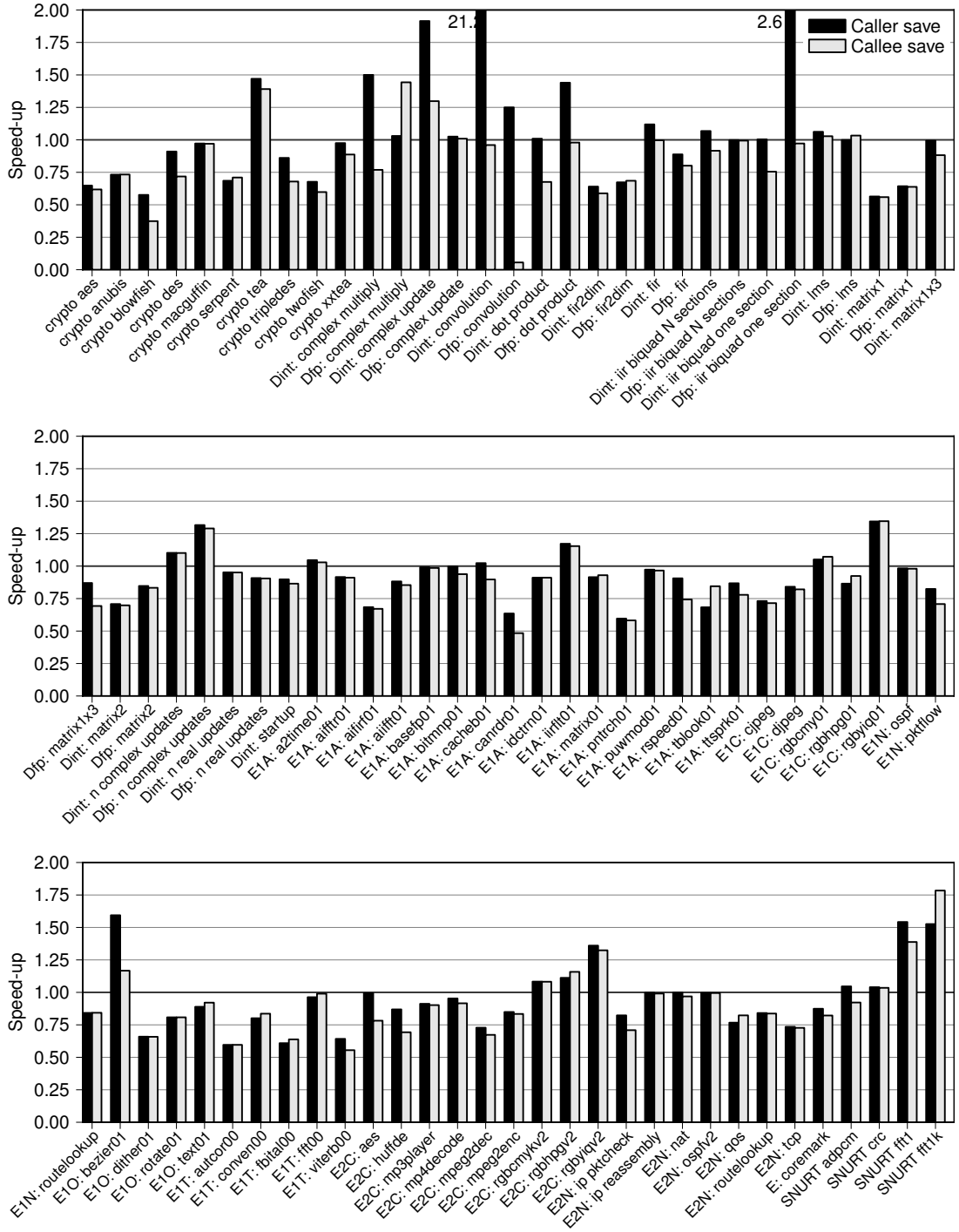


Figure B.20: Note: this is the full version of figure 4.22(a) on page 71. *An evaluation of callee-saved registers against caller-save.* (Continued on the next page.)

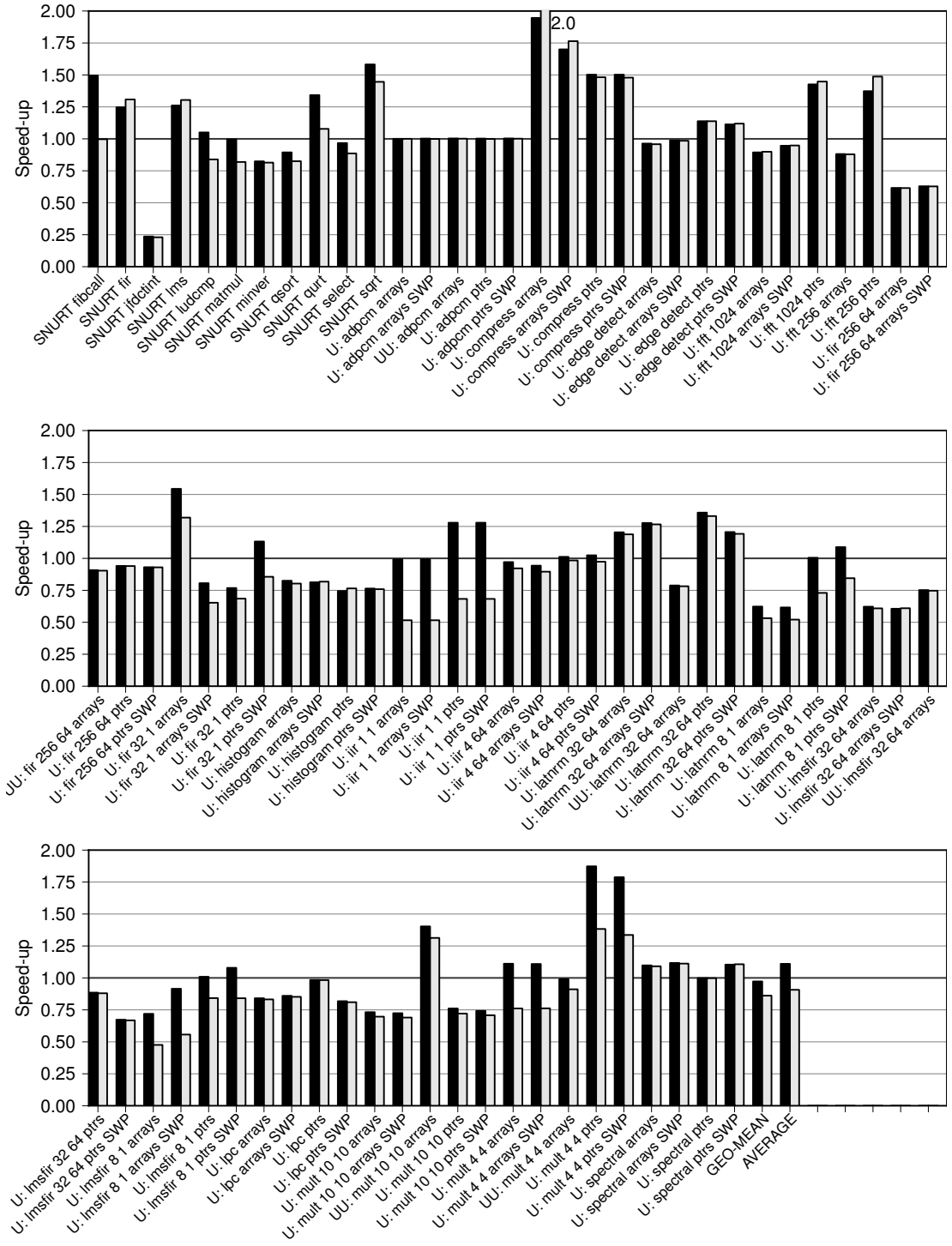


Figure B.20 (continued): *Default register allocator.*

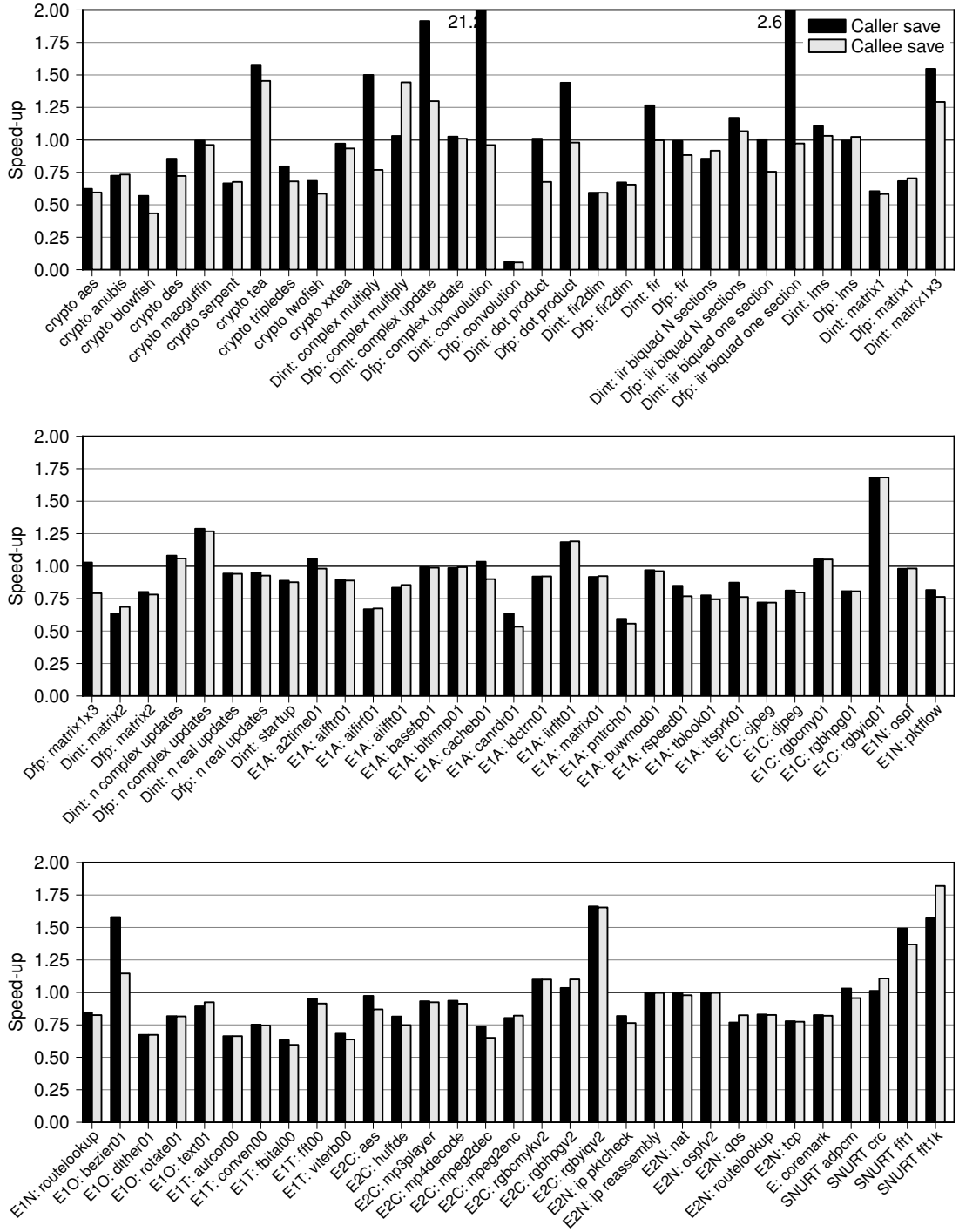


Figure B.21: Note: this is the full version of figure 4.22(b) on page 71. *An evaluation of callee-saved registers against caller-save.* (Continued on the next page.)

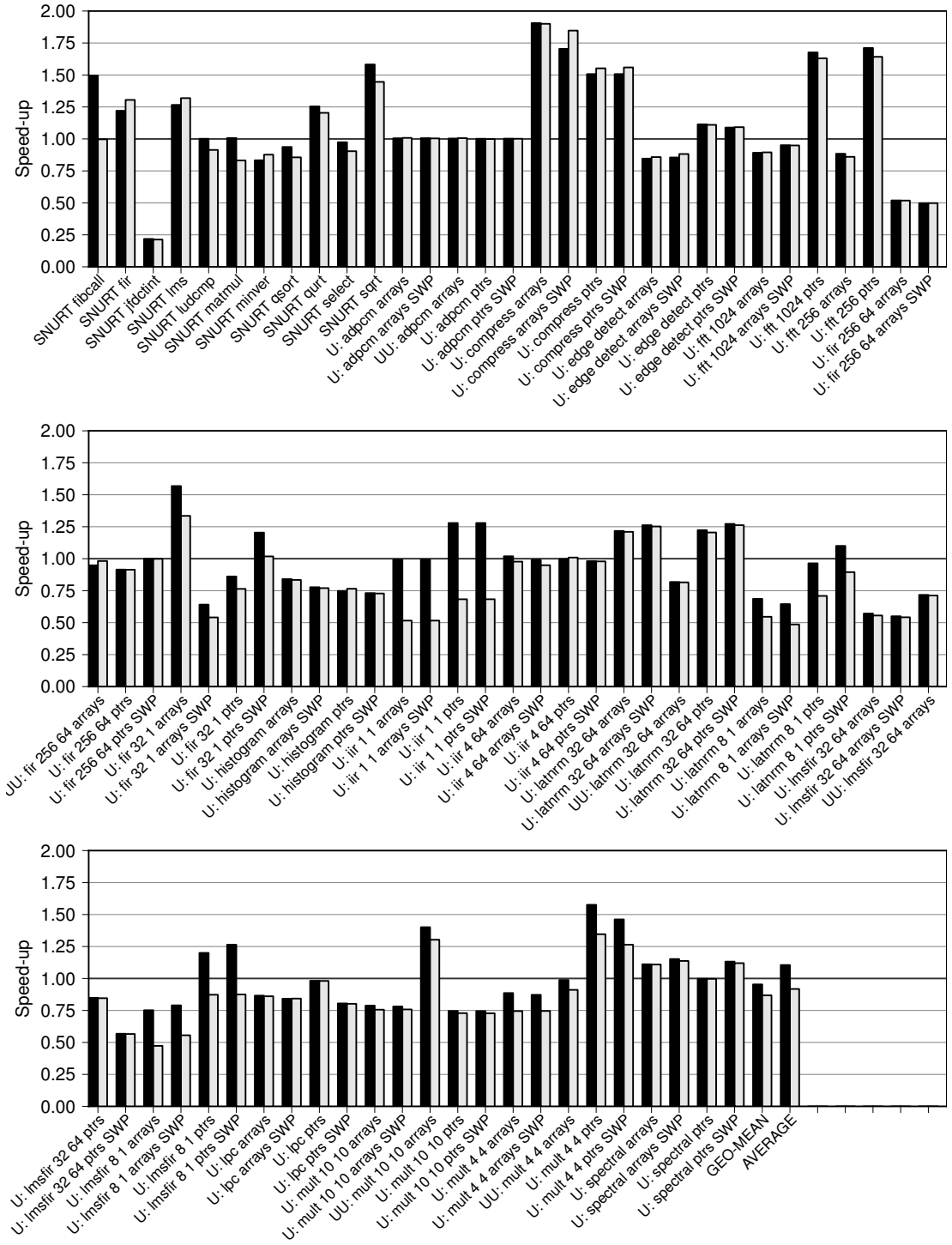


Figure B.21 (continued): *No register allocator.*

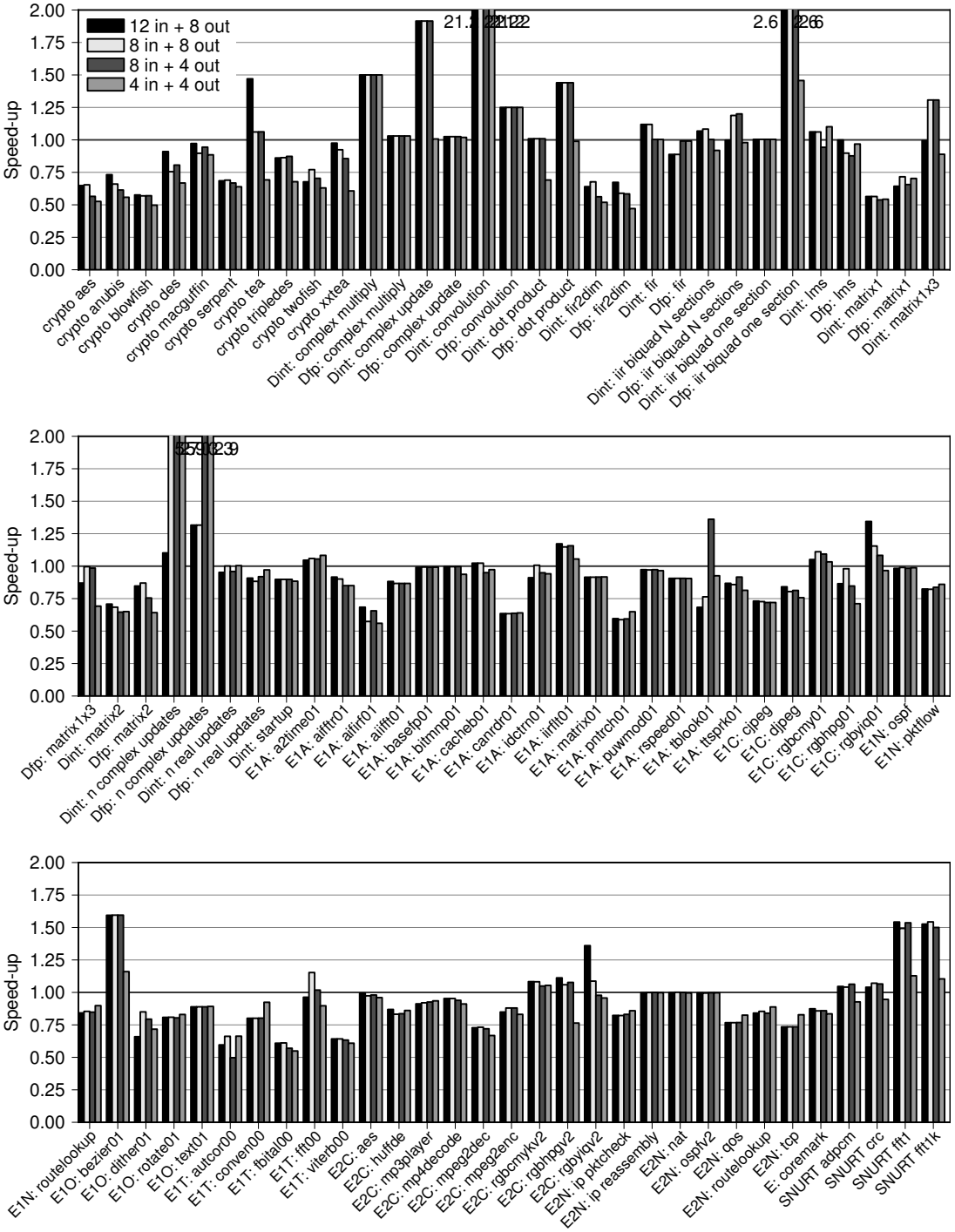


Figure B.22: Note: this is the full version of figure 5.1(a) on page 80. *A comparison of different register port constraints.* (Continued on the next page.)

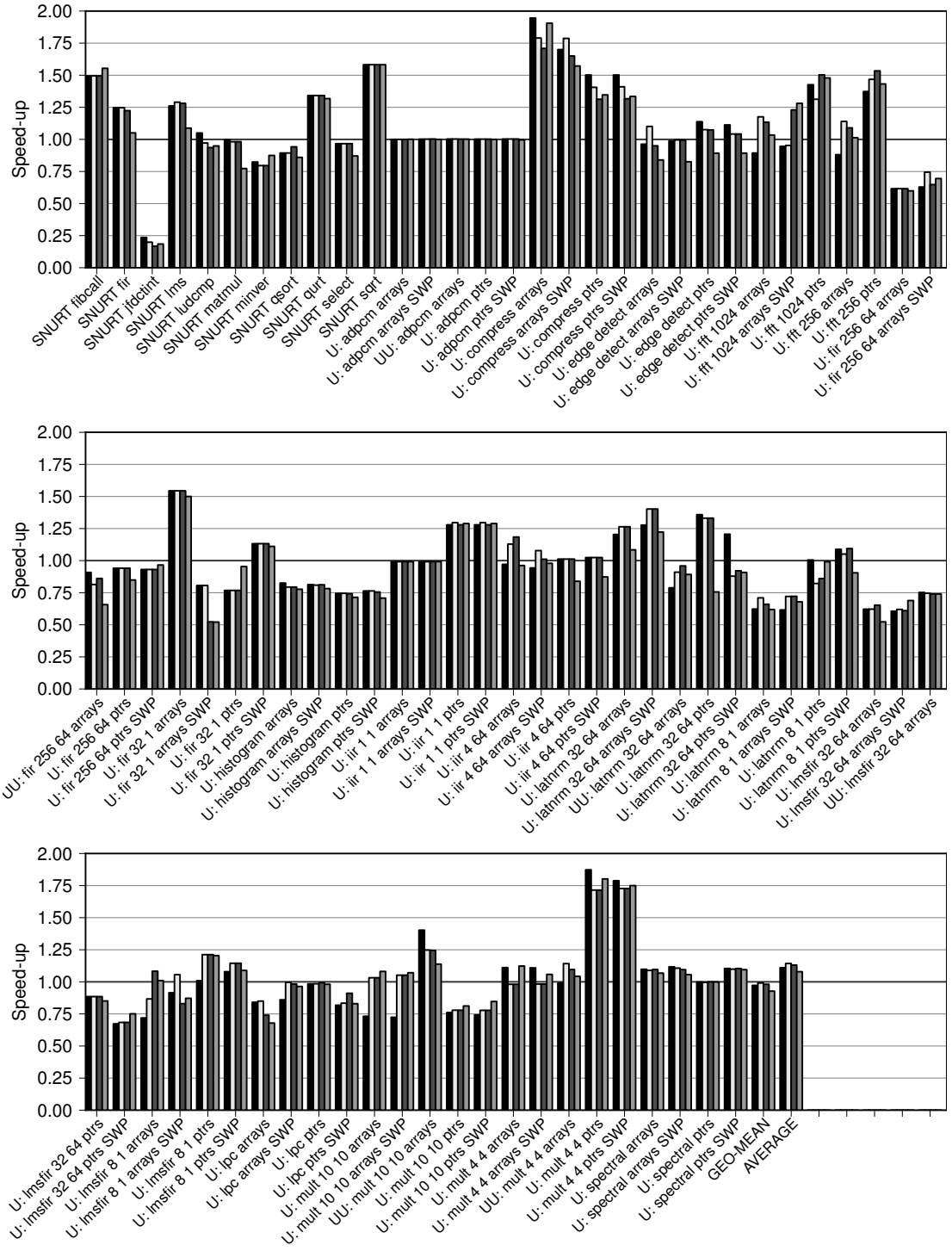


Figure B.22 (continued): *Speed-ups.*

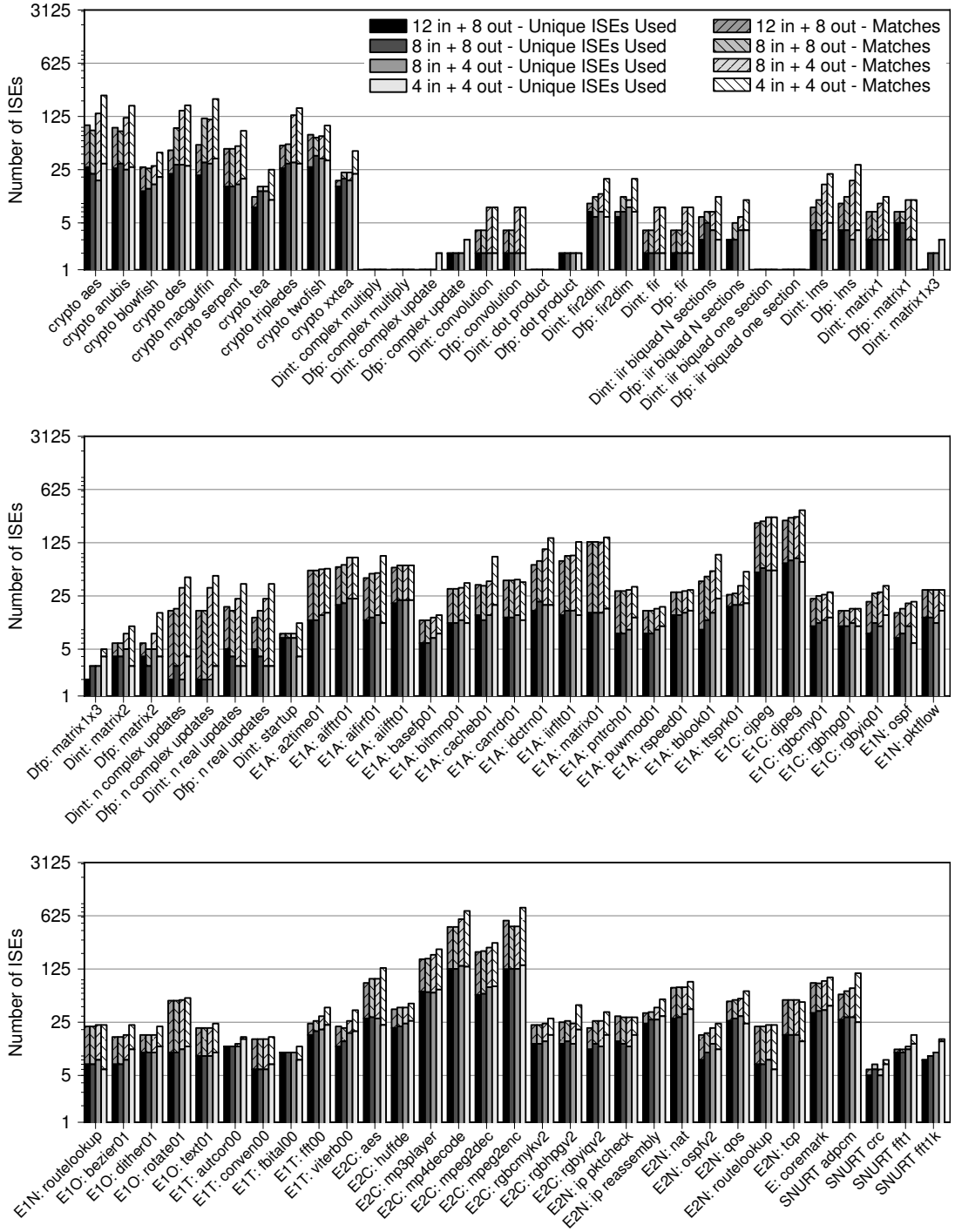


Figure B.23: Note: this is the full version of figure 5.1(b) on page 80. *A comparison of different register port constraints.* (Continued on the next page.)

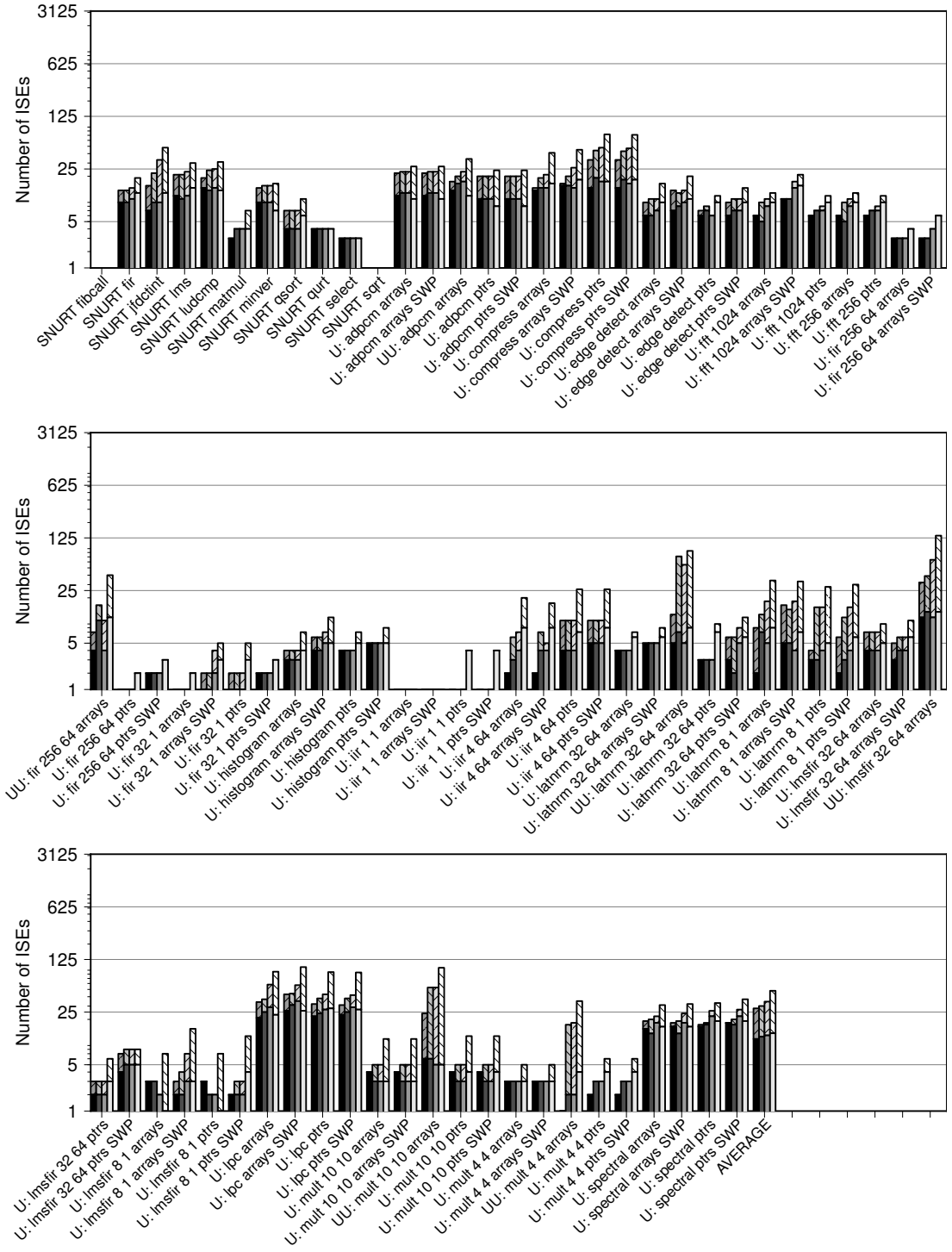


Figure B.23 (continued): *Mapping quality information.*

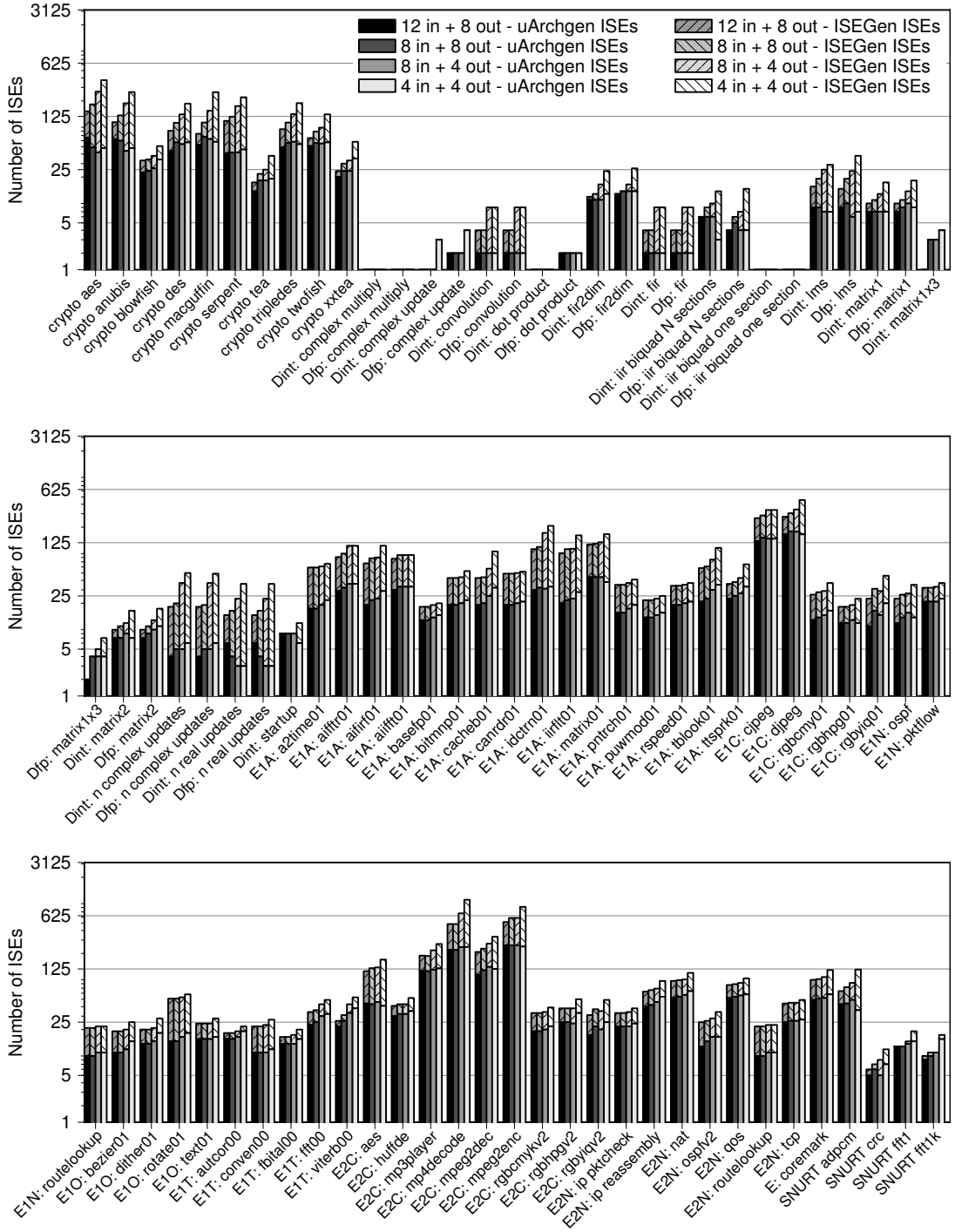


Figure B.24: Note: this is the full version of figure 5.2 on page 81. *The effect that different register port constraints have on the number of* (Continued on the next page.)

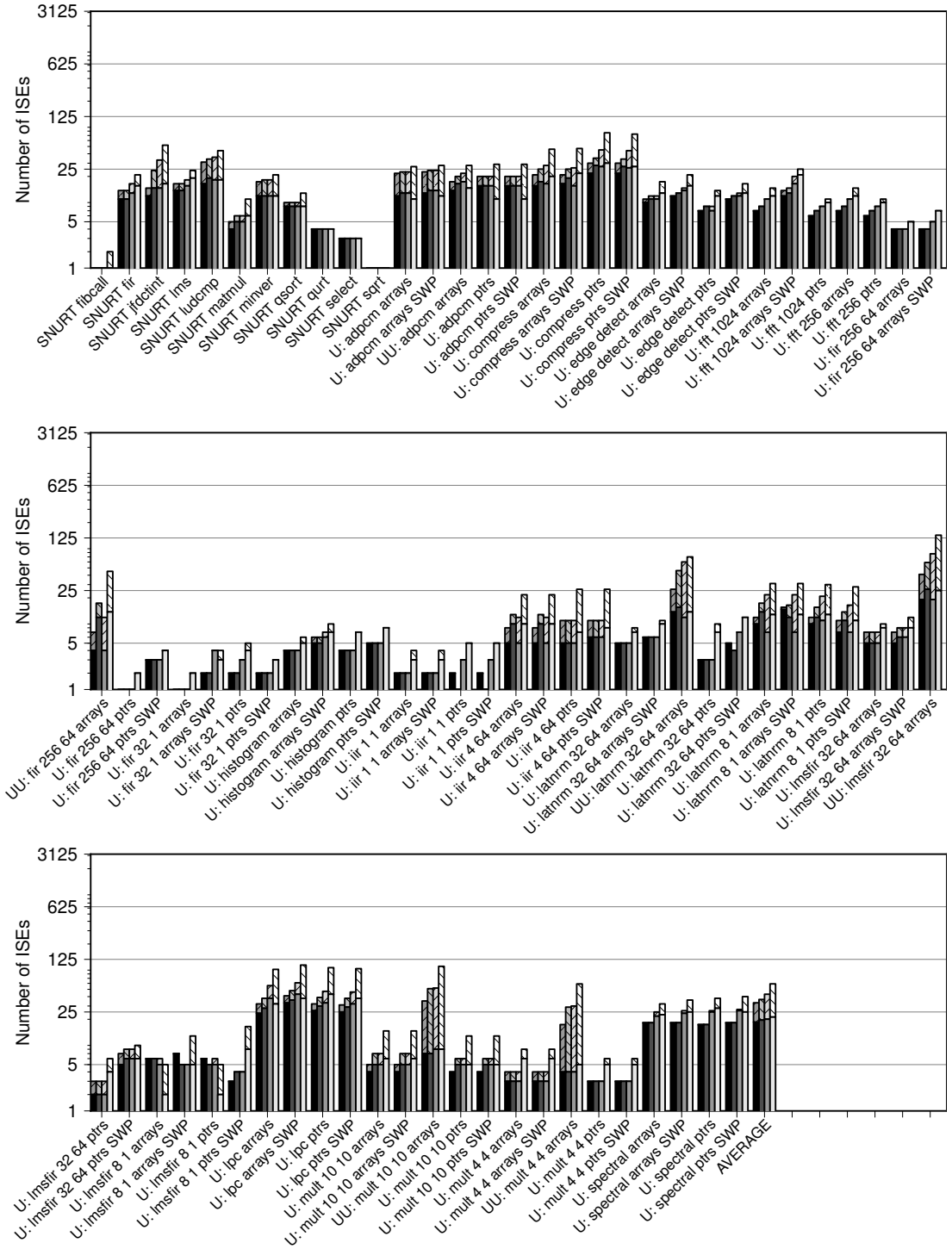


Figure B.24 (continued): *extension instructions that* ISEGen *and* uArchGen *will produce.*

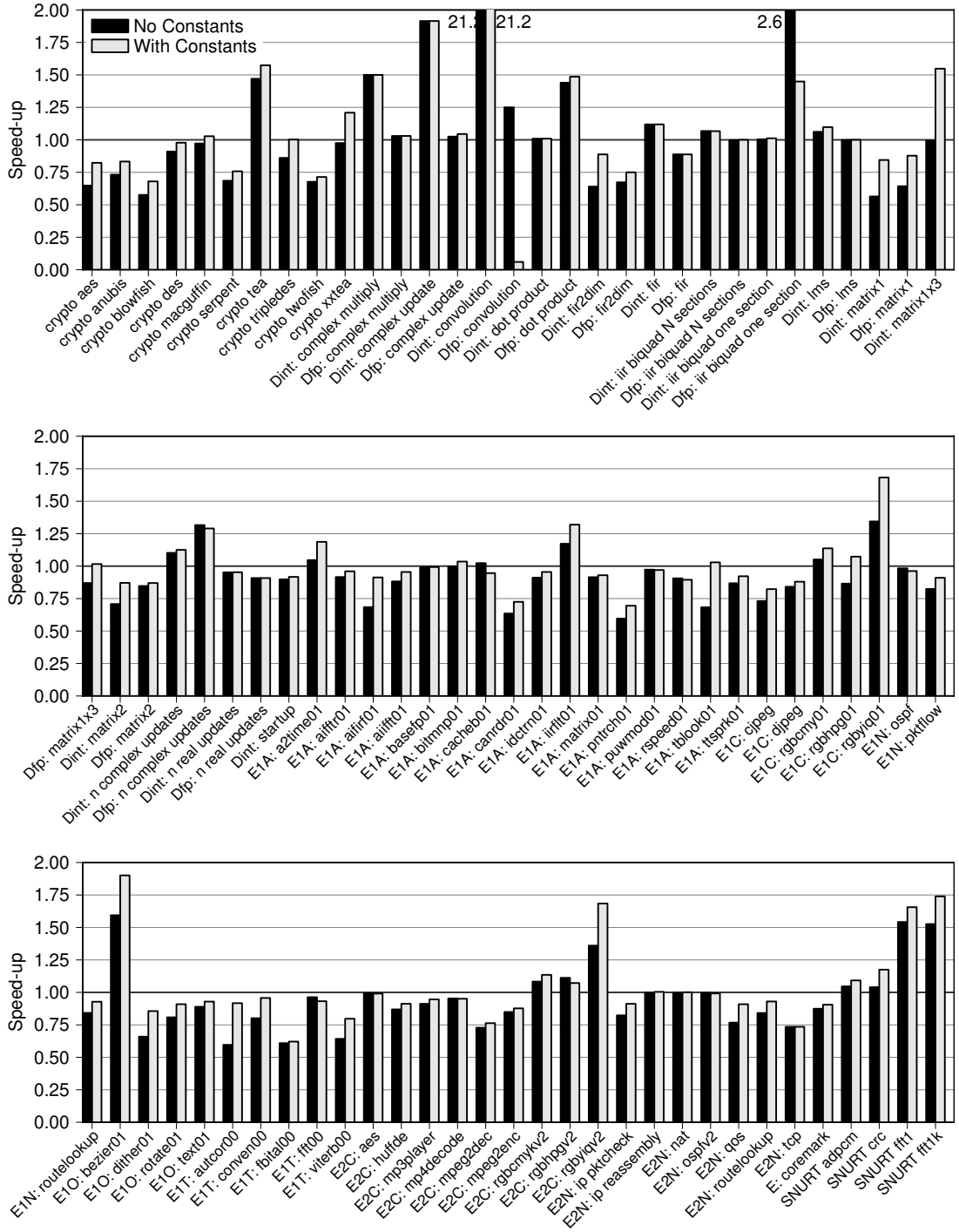


Figure B.25: Note: this is the full version of figure 5.3(a) on page 82. *The effect that introducing hard-wired constant values into extension instructions has on the end results and* MapISE's *ability to use* (Continued on the next page.)

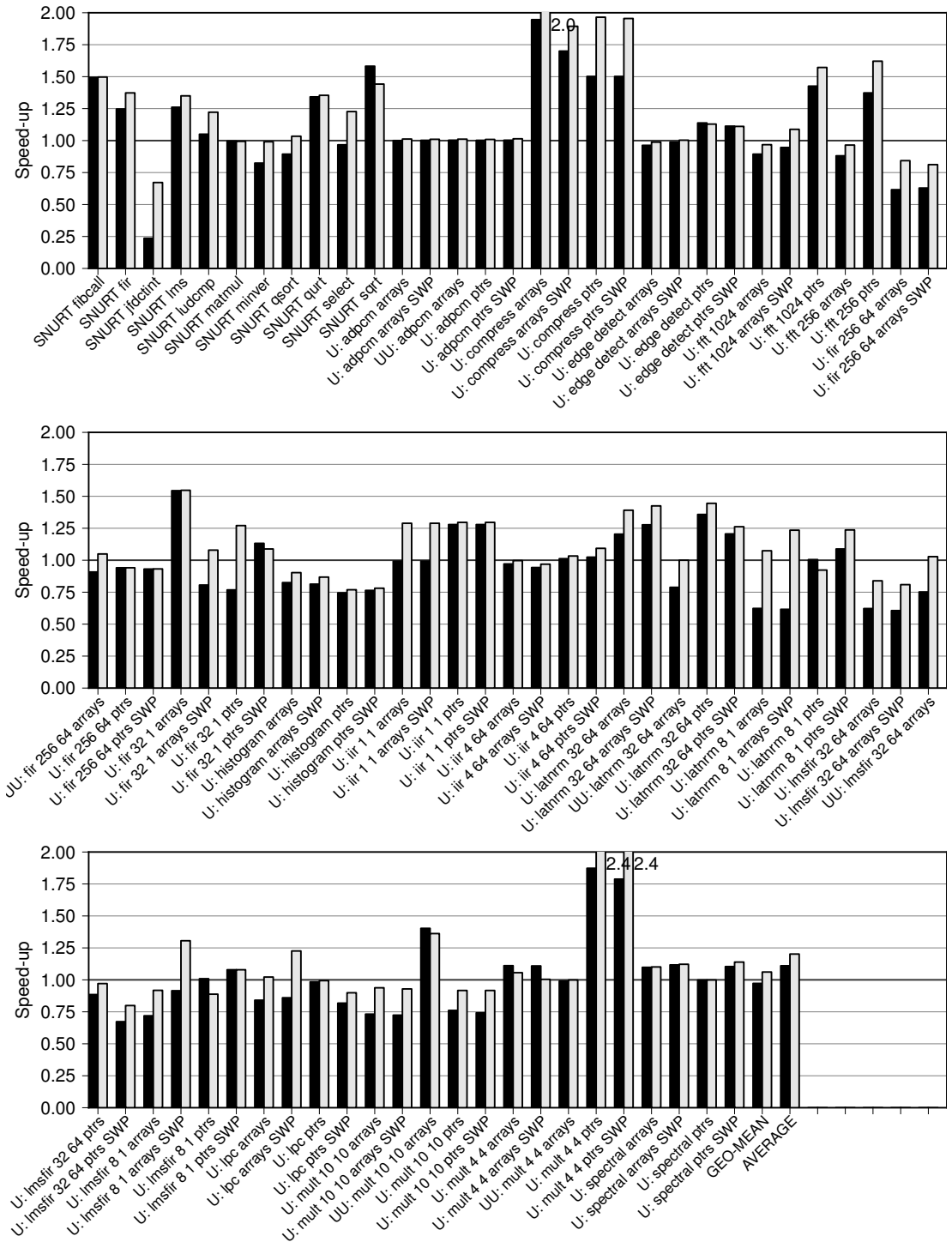


Figure B.25 (continued): *Speed-ups.*

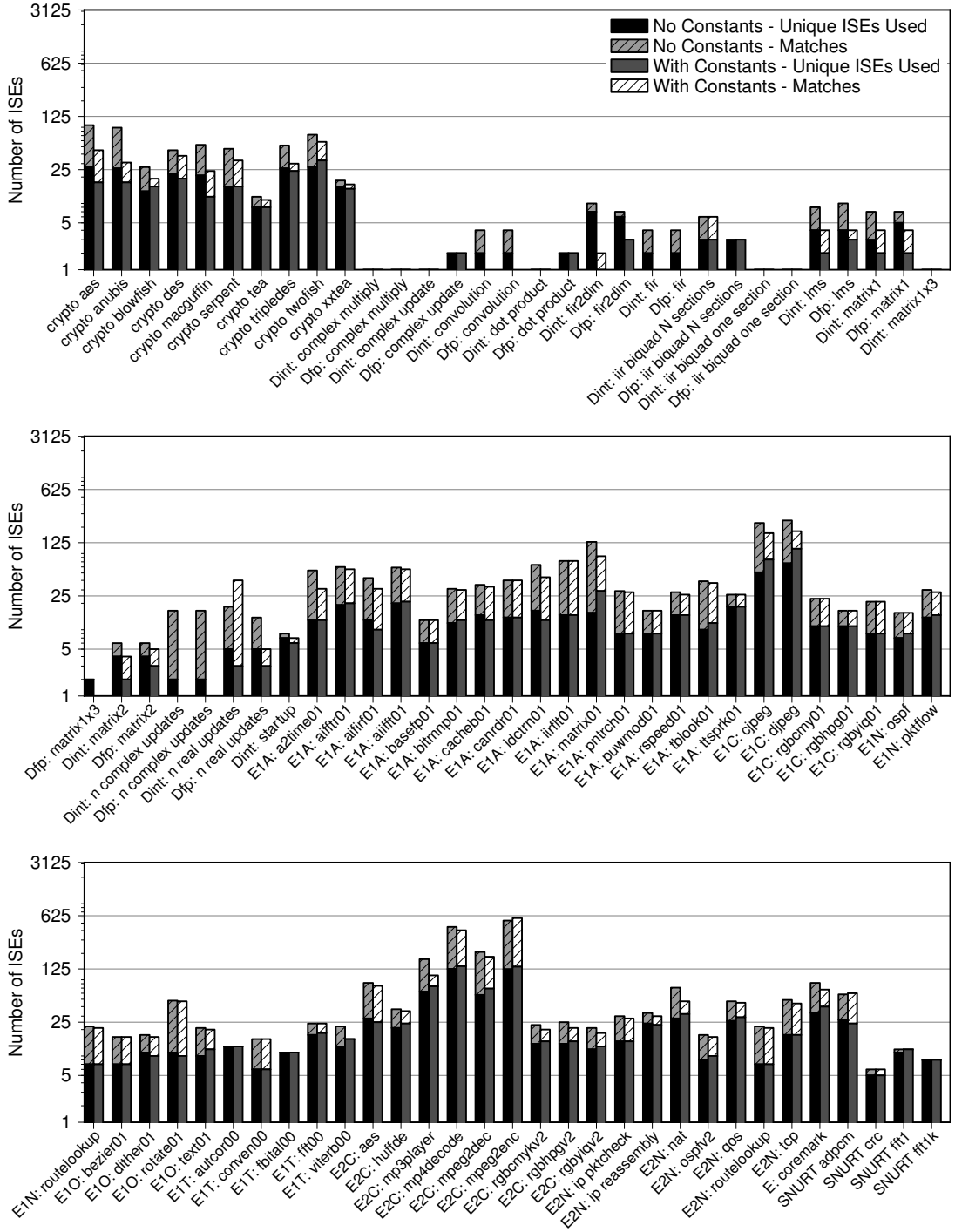


Figure B.26: Note: this is the full version of figure 5.3(b) on page 82. *The effect that introducing hard-wired constant values into extension instructions has on the end results and* MapISE's *ability to use* (Continued on the next page.)

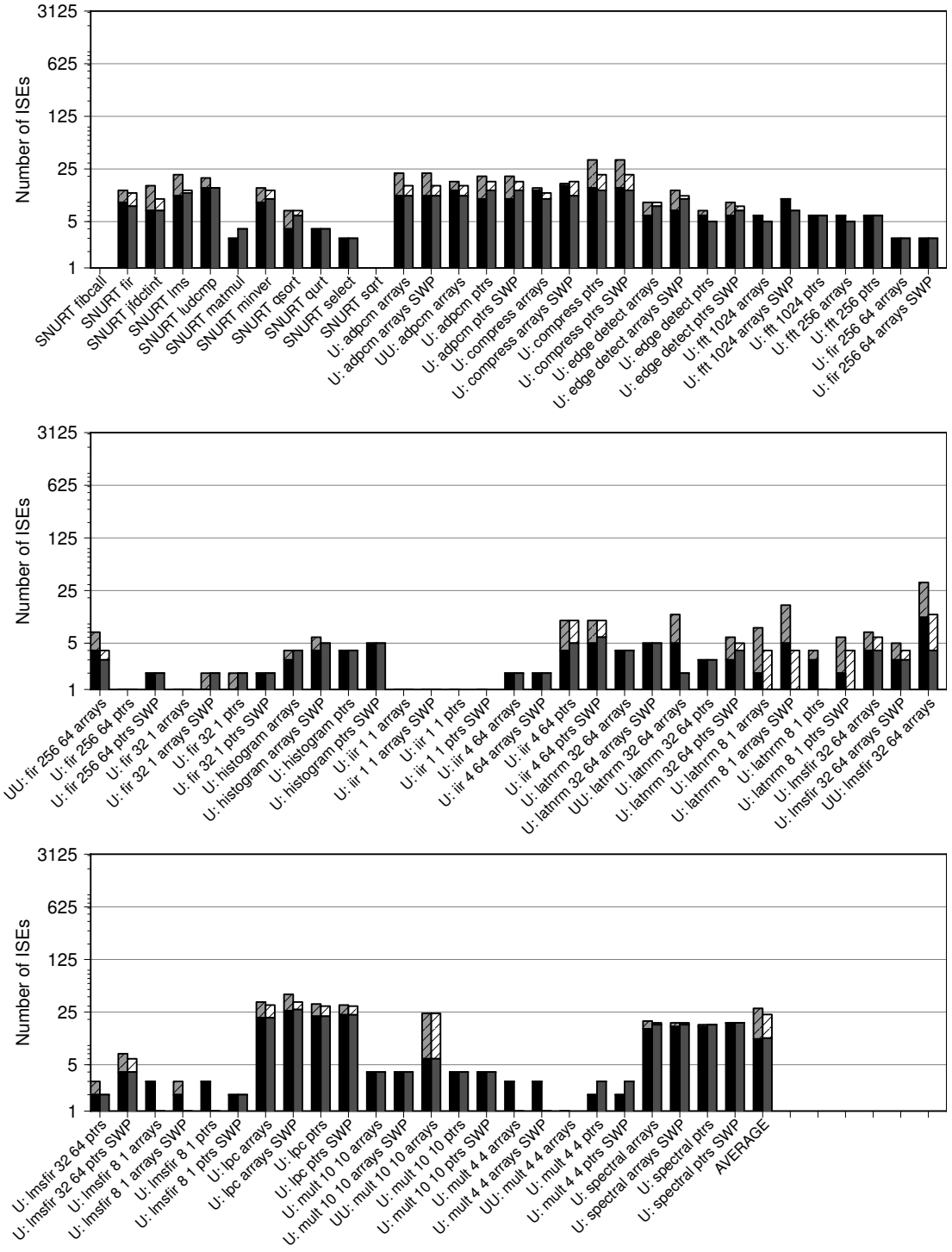


Figure B.26 (continued): *Mapping quality information.*

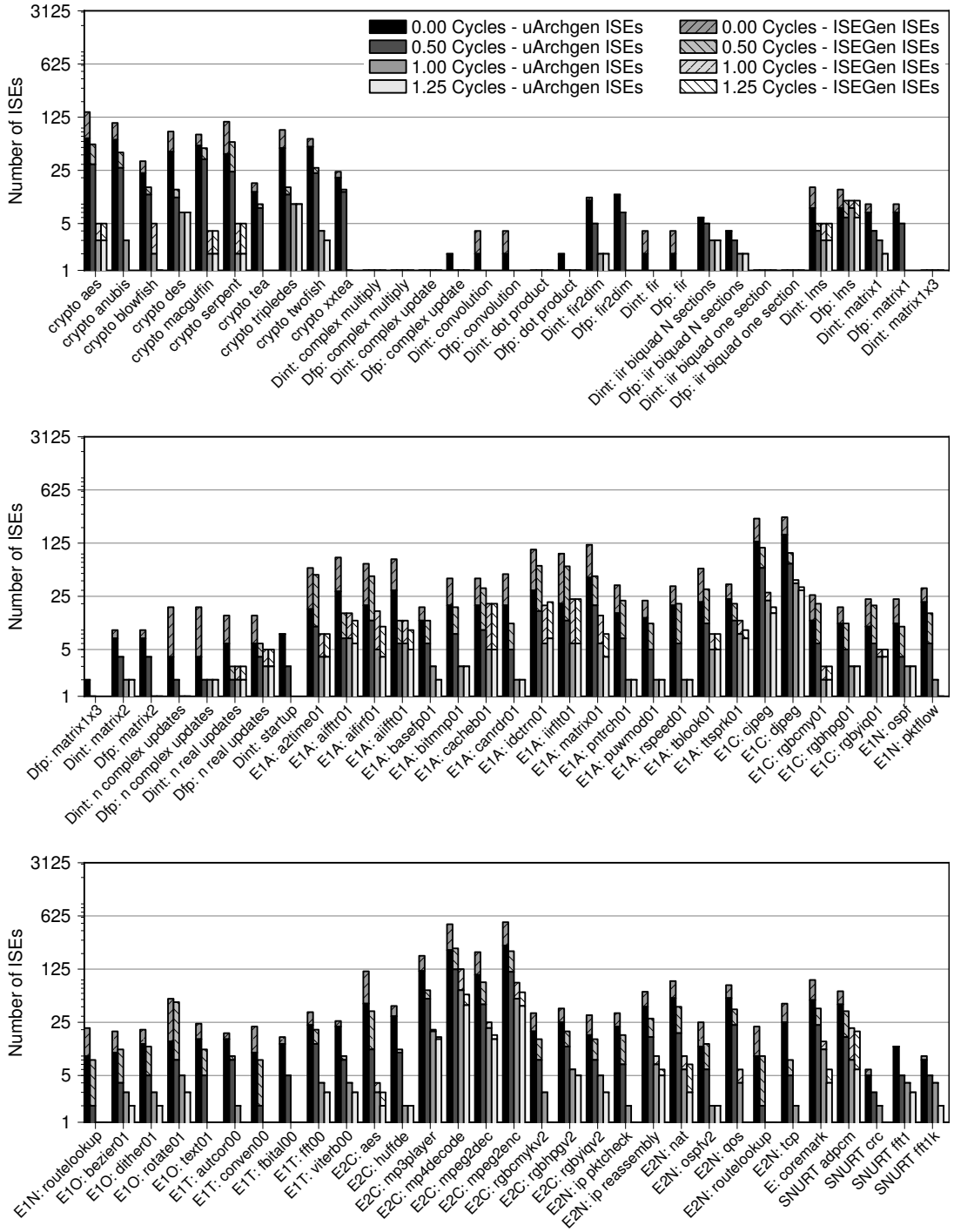


Figure B.27: Note: this is the full version of figure 5.4(a) on page 83. *A comparison of different register cost values to provide to* ISEGen's *heuristics. Note: Figures B.35 and B.29 on pages 224 and 212 cover additional parameter values (0.00, 0.25, 0.75 and 1.50 cycles).* (Continued on the next page.)

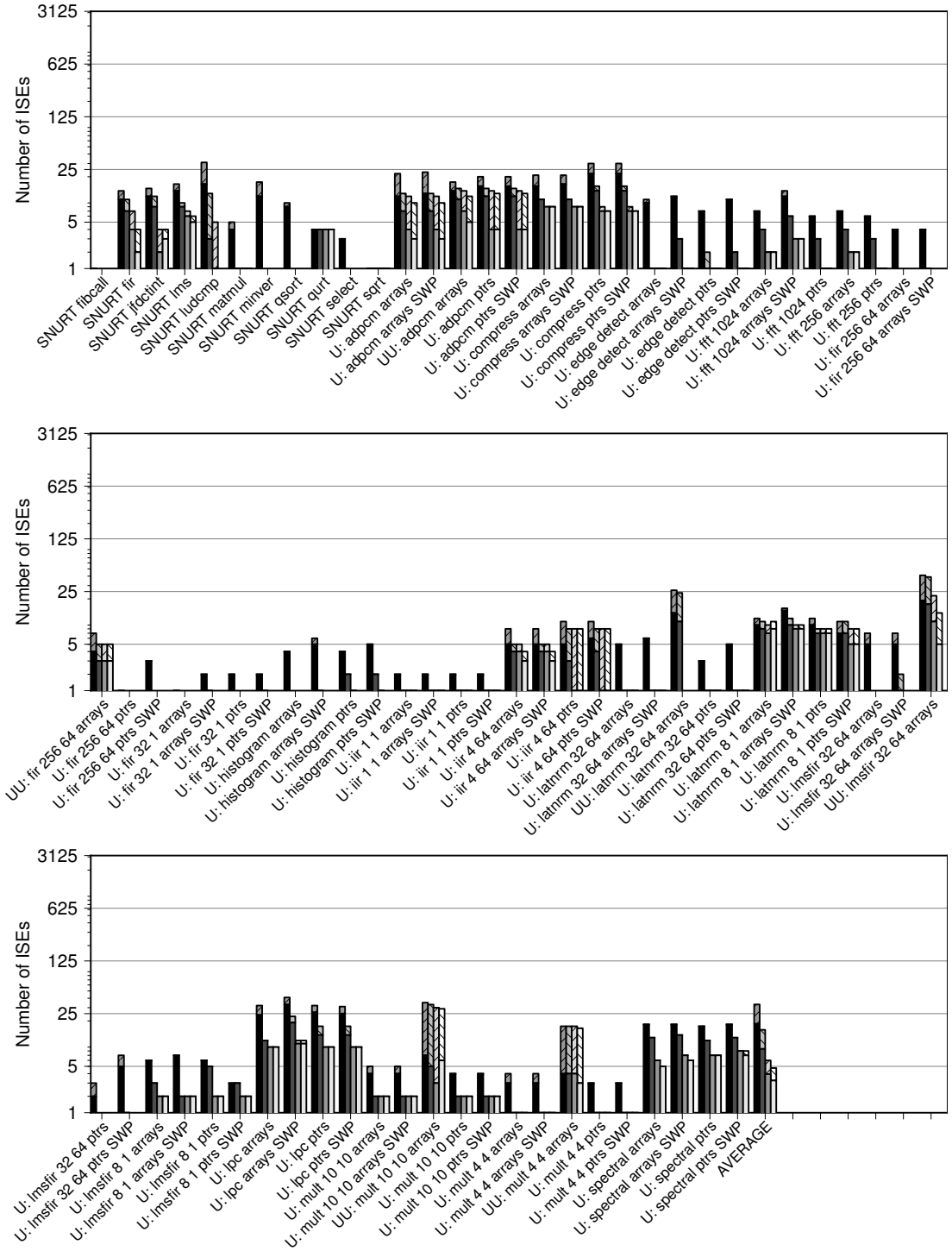


Figure B.27 (continued): *The effect that the register load cost parameter has on the number of extension instructions that* ISEGen *finds.*

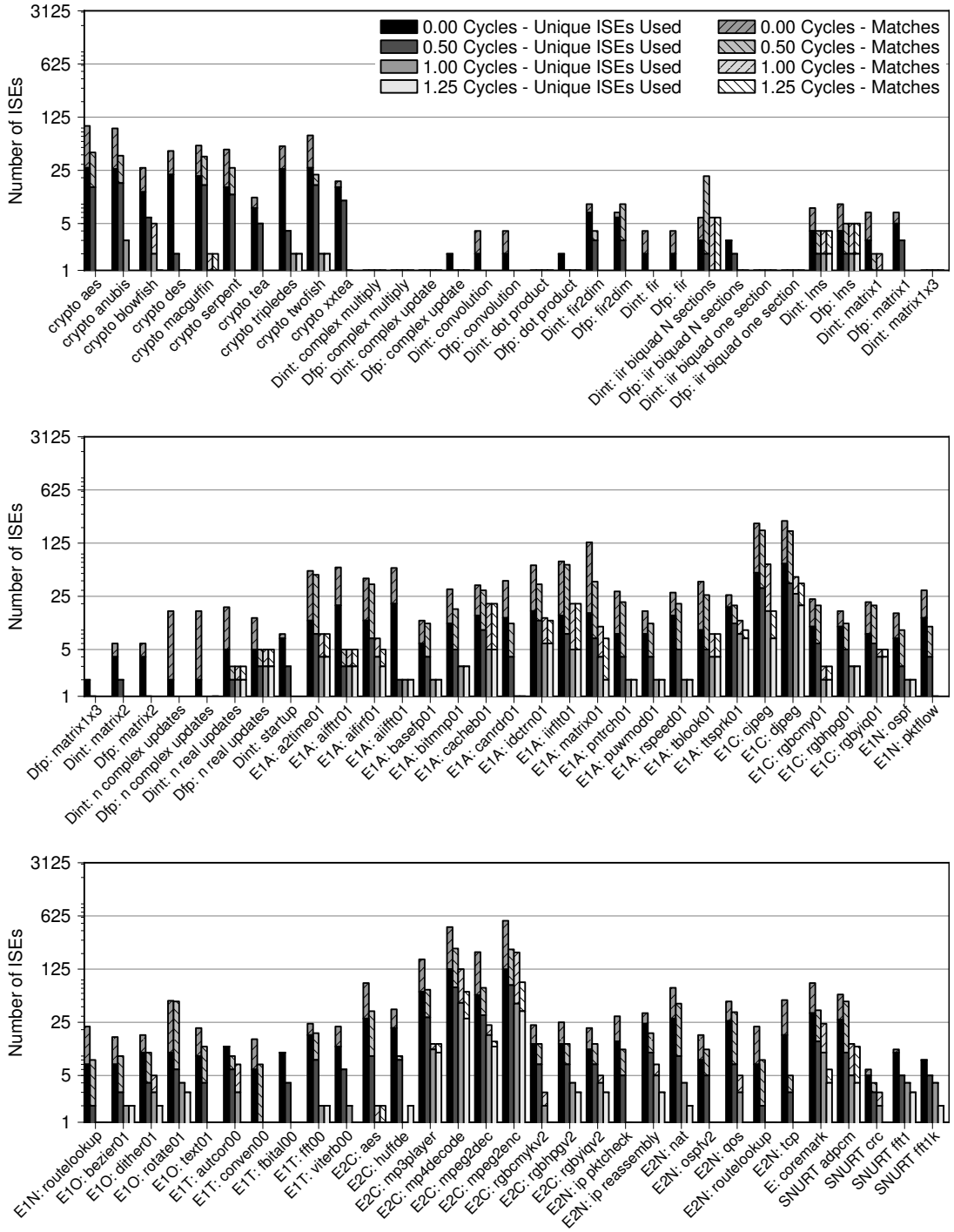


Figure B.28: Note: this is the full version of figure 5.4(b) on page 83. *A comparison of different register cost values to provide to* ISEGen's *heuristics. Note: Figures B.35 and B.29 on pages 224 and 212 cover additional parameter values (0.00, 0.25, 0.75 and 1.50 cycles).* (Continued on the next page.)

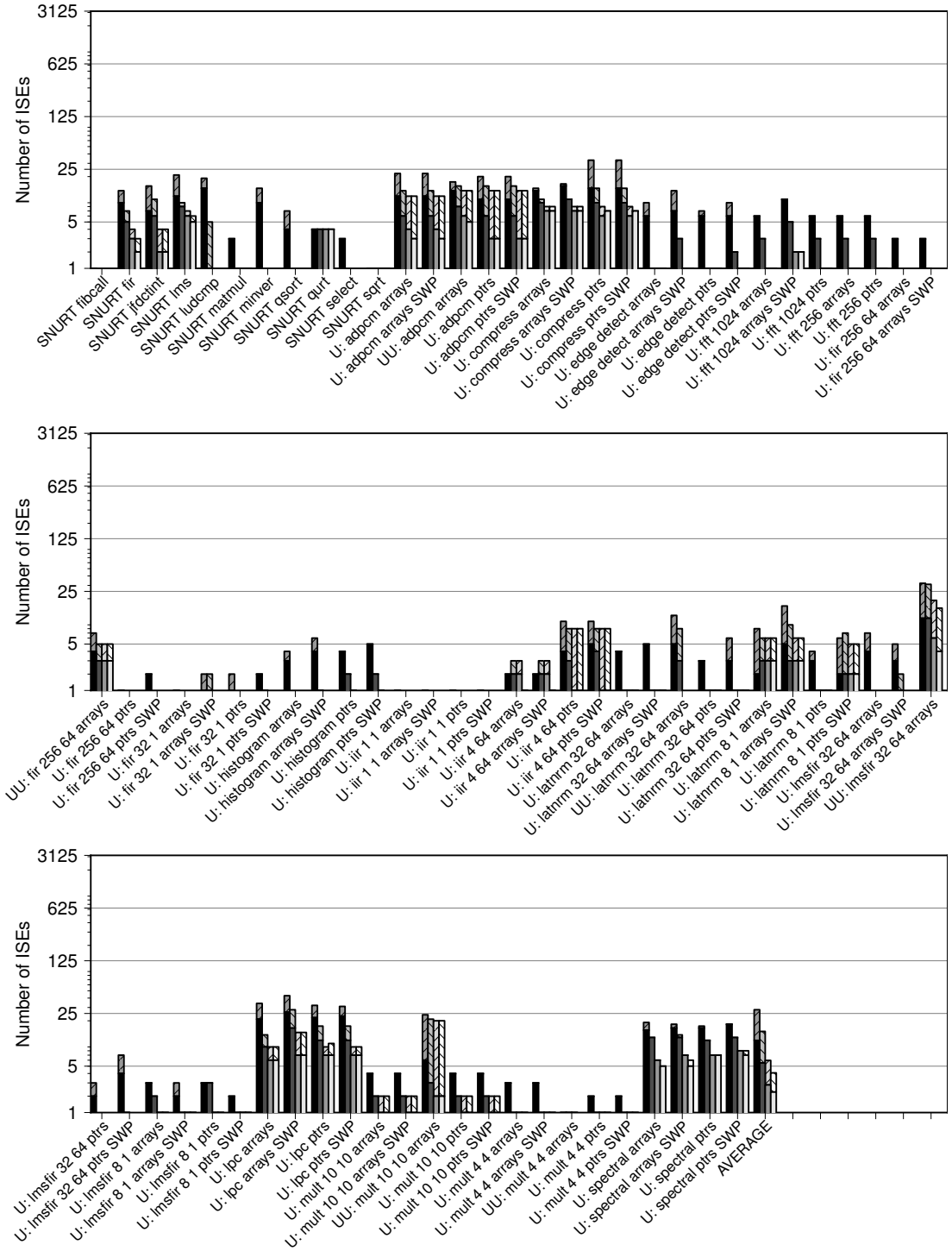


Figure B.28 (continued): MapISE's *ability to use the instructions found with different register load cost parameter values.*

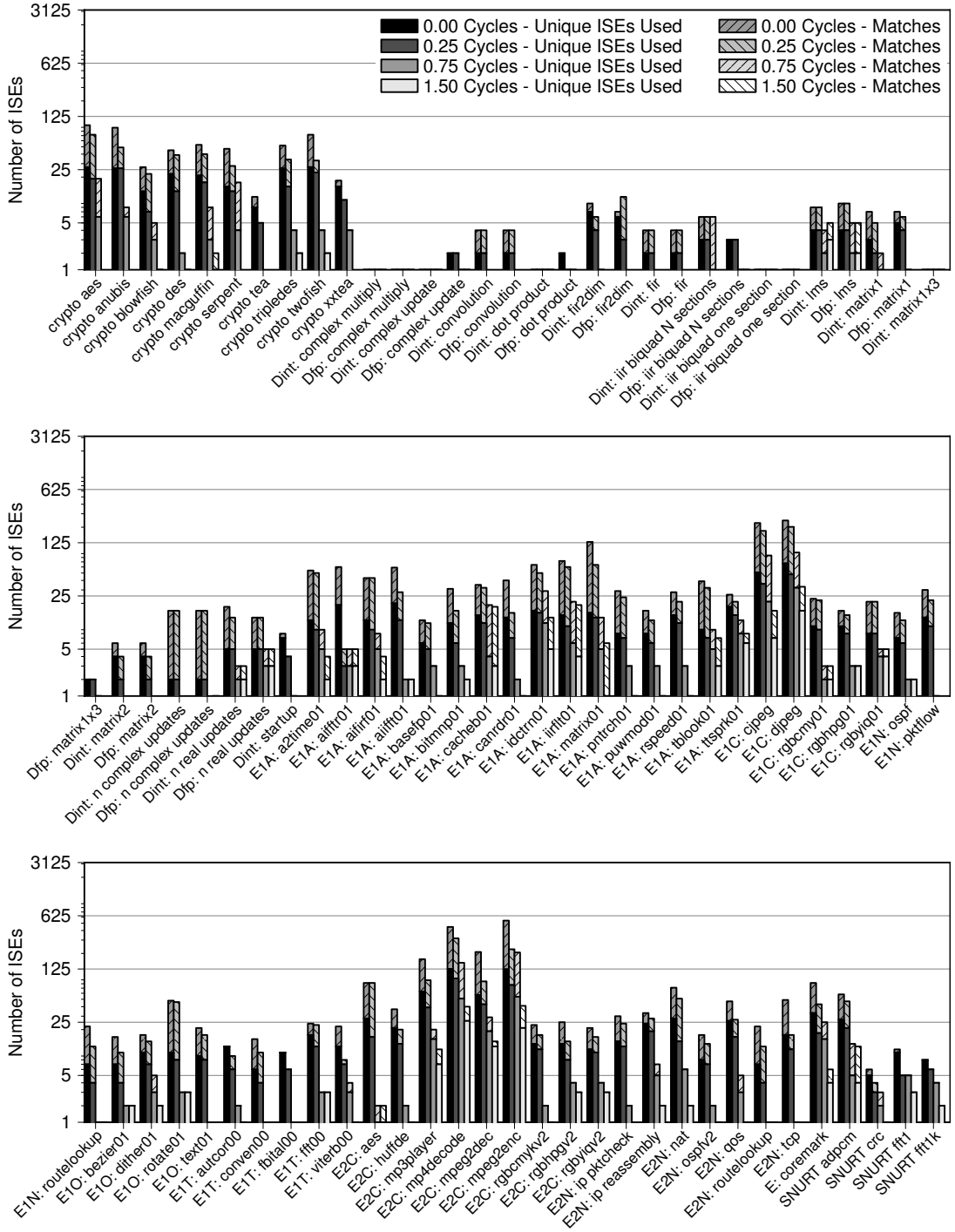


Figure B.29: *Note: This is a variation of figure B.28, which, in turn, is the full version of figure 5.4(b) on page 83.* (Continued on the next page.)

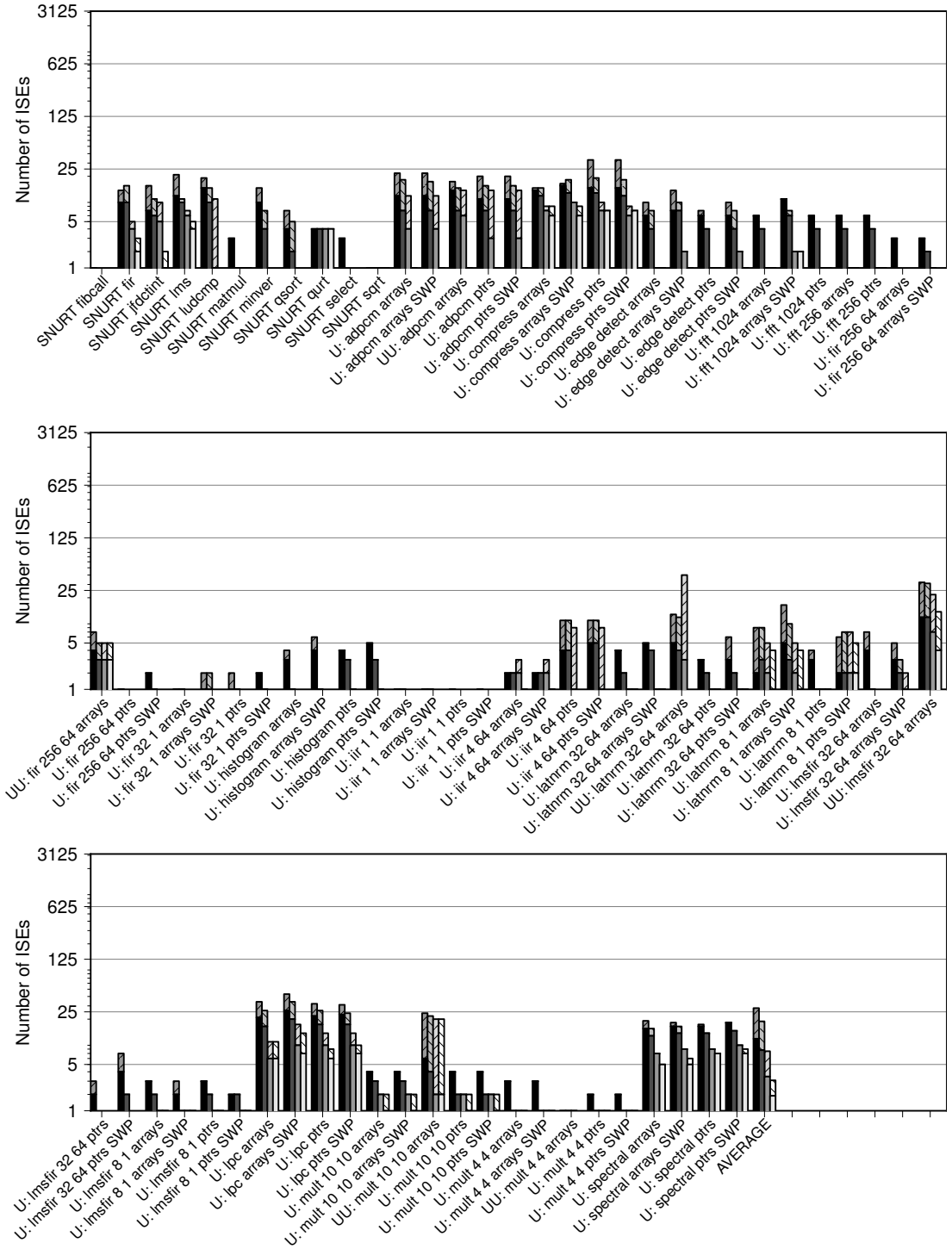


Figure B.29 (continued):*These charts contain three parameter values that were not included in figures 5.4(b) or B.28.*

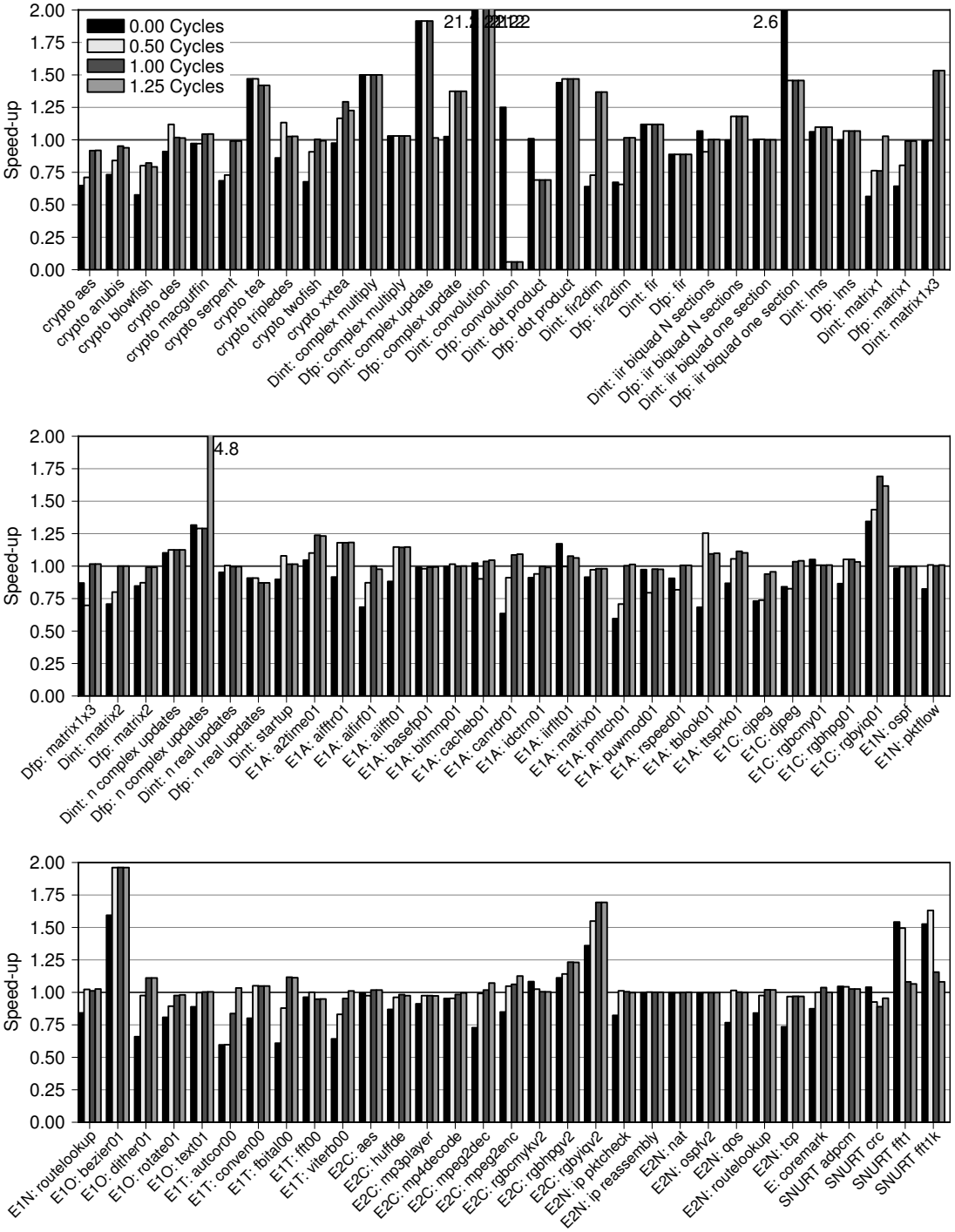


Figure B.30: Note: this is the full version of figure 5.5 on page 84. *The speed-ups obtainable for extension instructions produced with different register cost parameter values.* (Continued on the next page.)

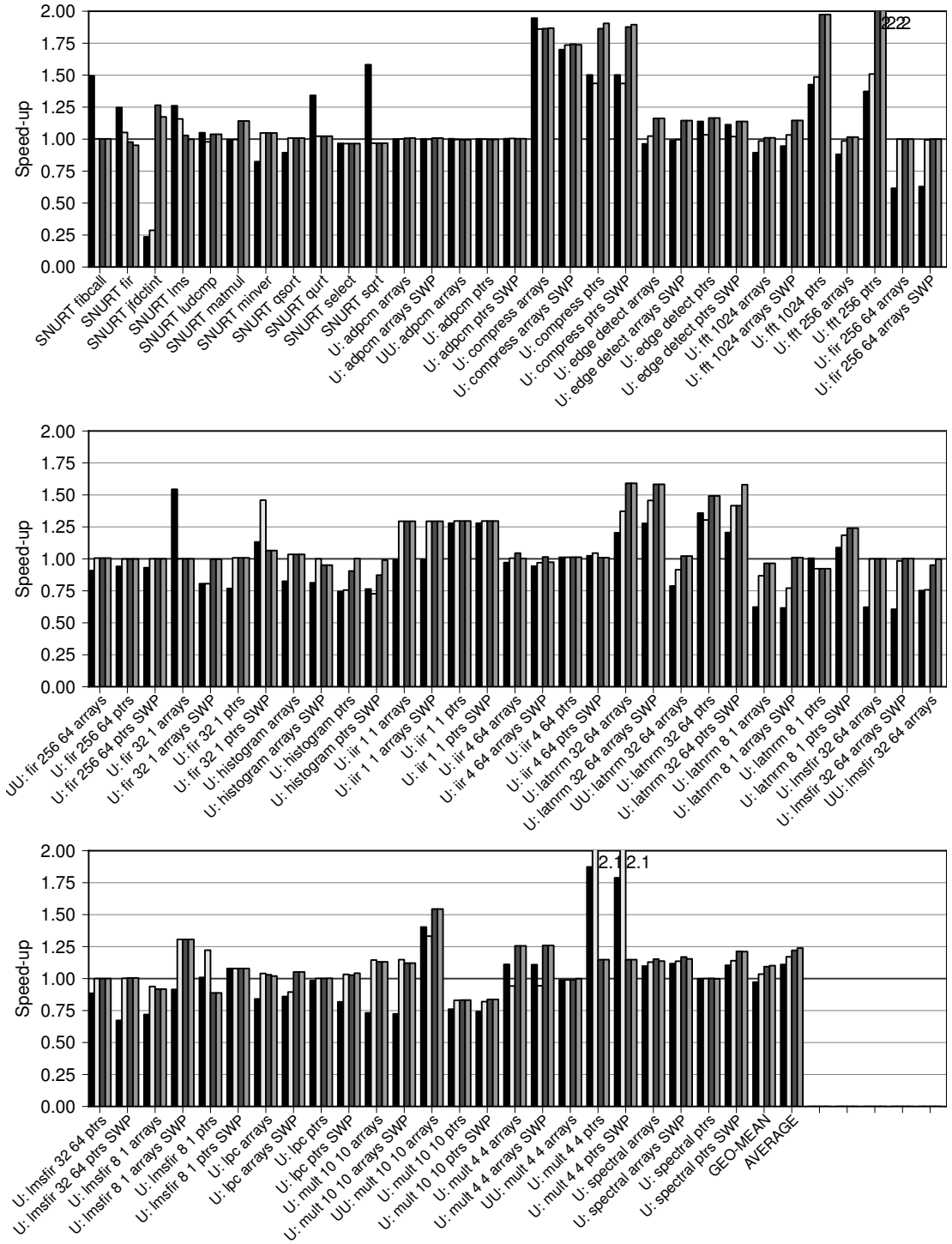


Figure B.30 (continued): *Note: Figure B.31 on page 216 is the equivalent of this chart but with different parameter values (0.00, 0.25, 0.75 and 1.50 cycles).*

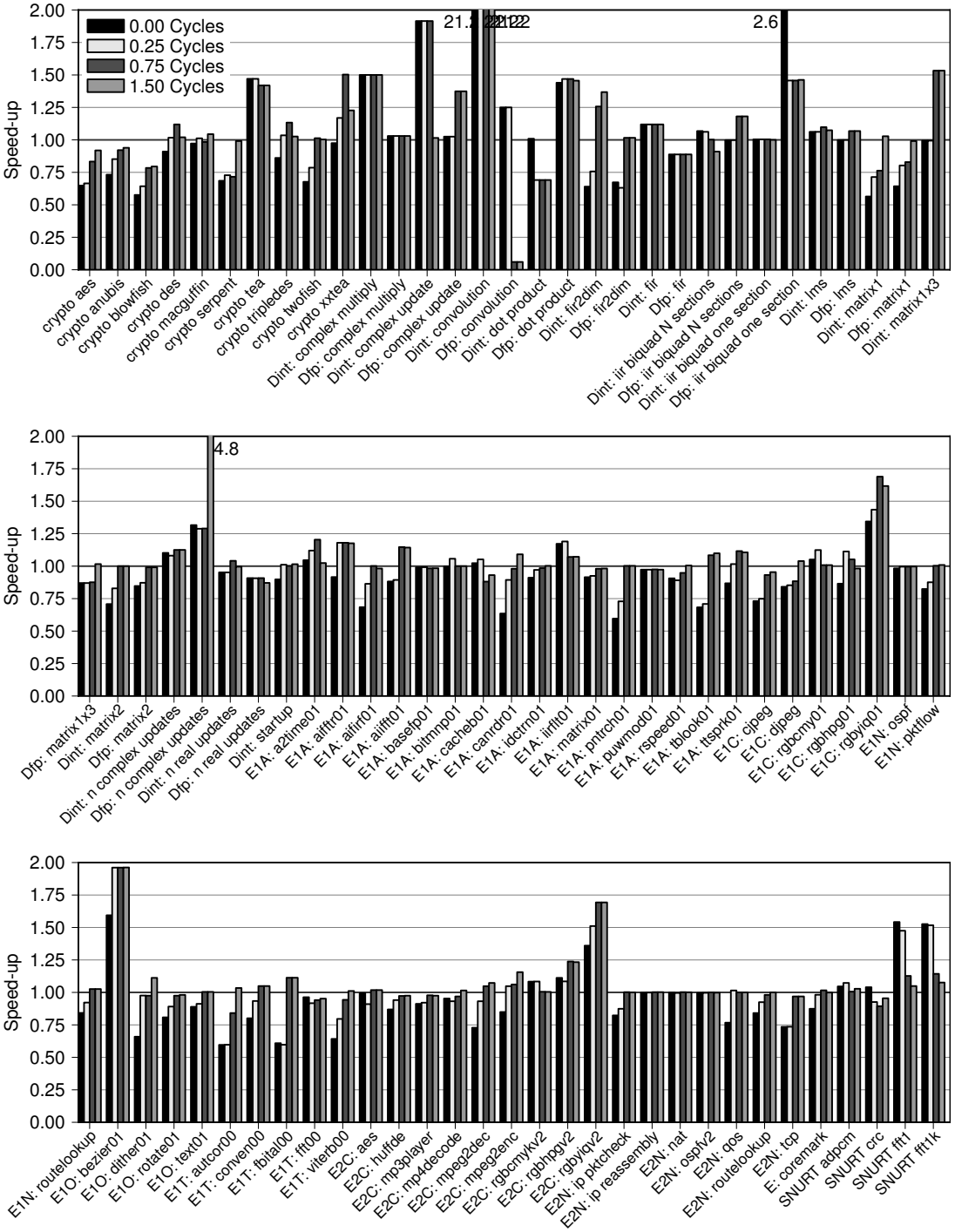


Figure B.31: *Note: This is a variation of figure B.30, which, in turn, is the full version of figure 5.5 on page 84.* (Continued on the next page.)

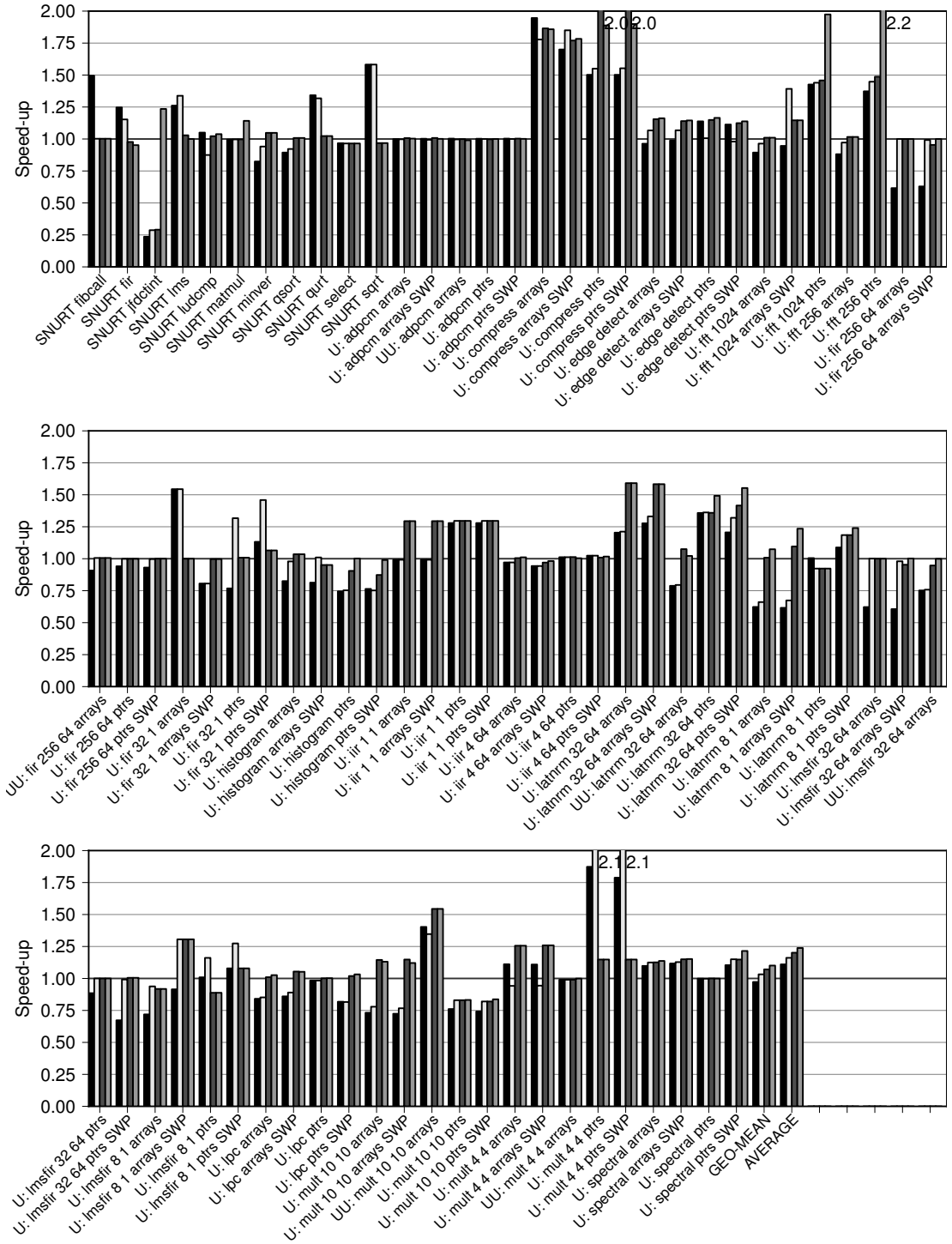


Figure B.31 (continued): *These charts contain three parameter values that were not included in figures 5.5 or B.30.*

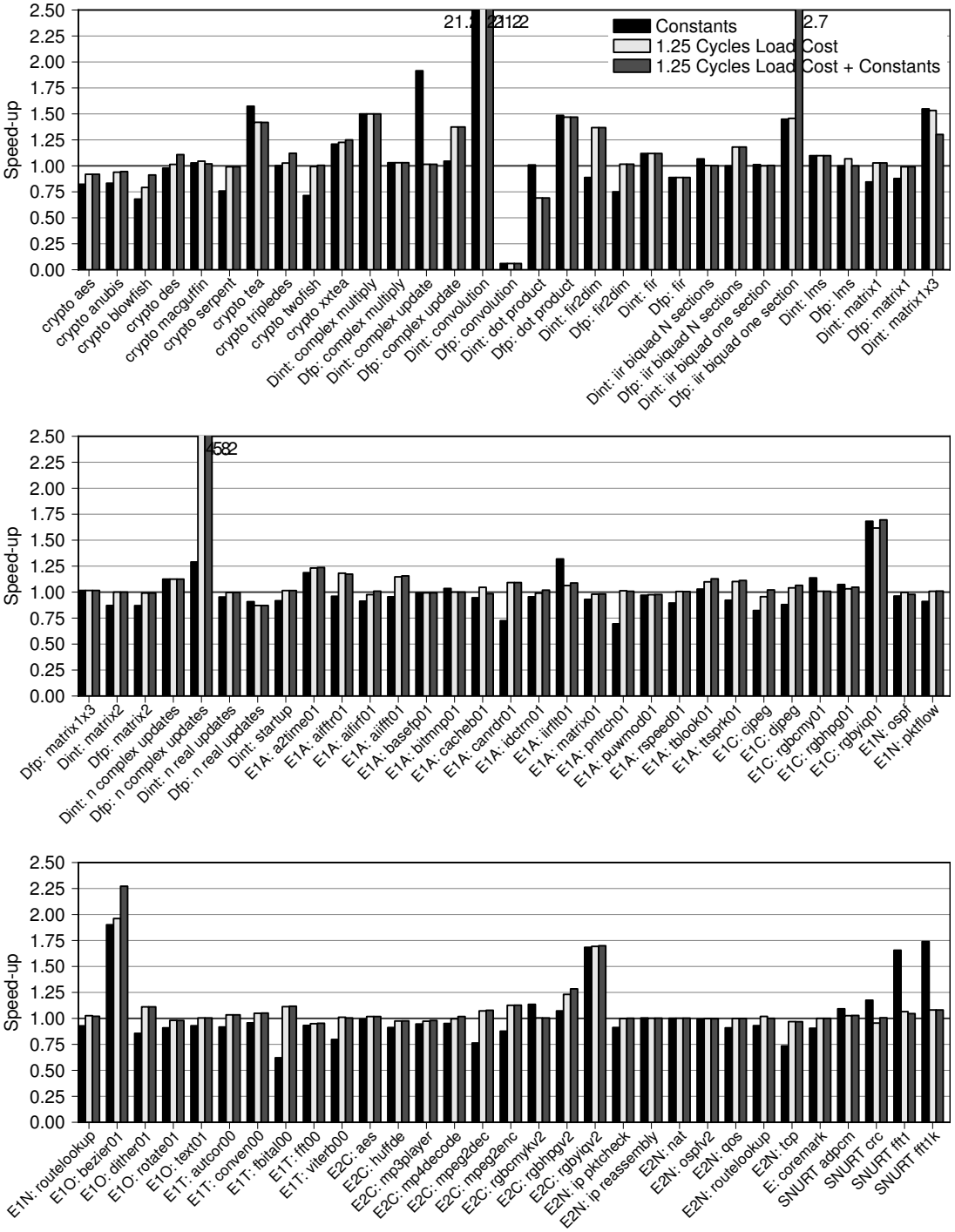


Figure B.32: Note: this is the full version of figure 5.6 on page 84. *A comparison of hard-wired constants alone, a register load cost* (Continued on the next page.)

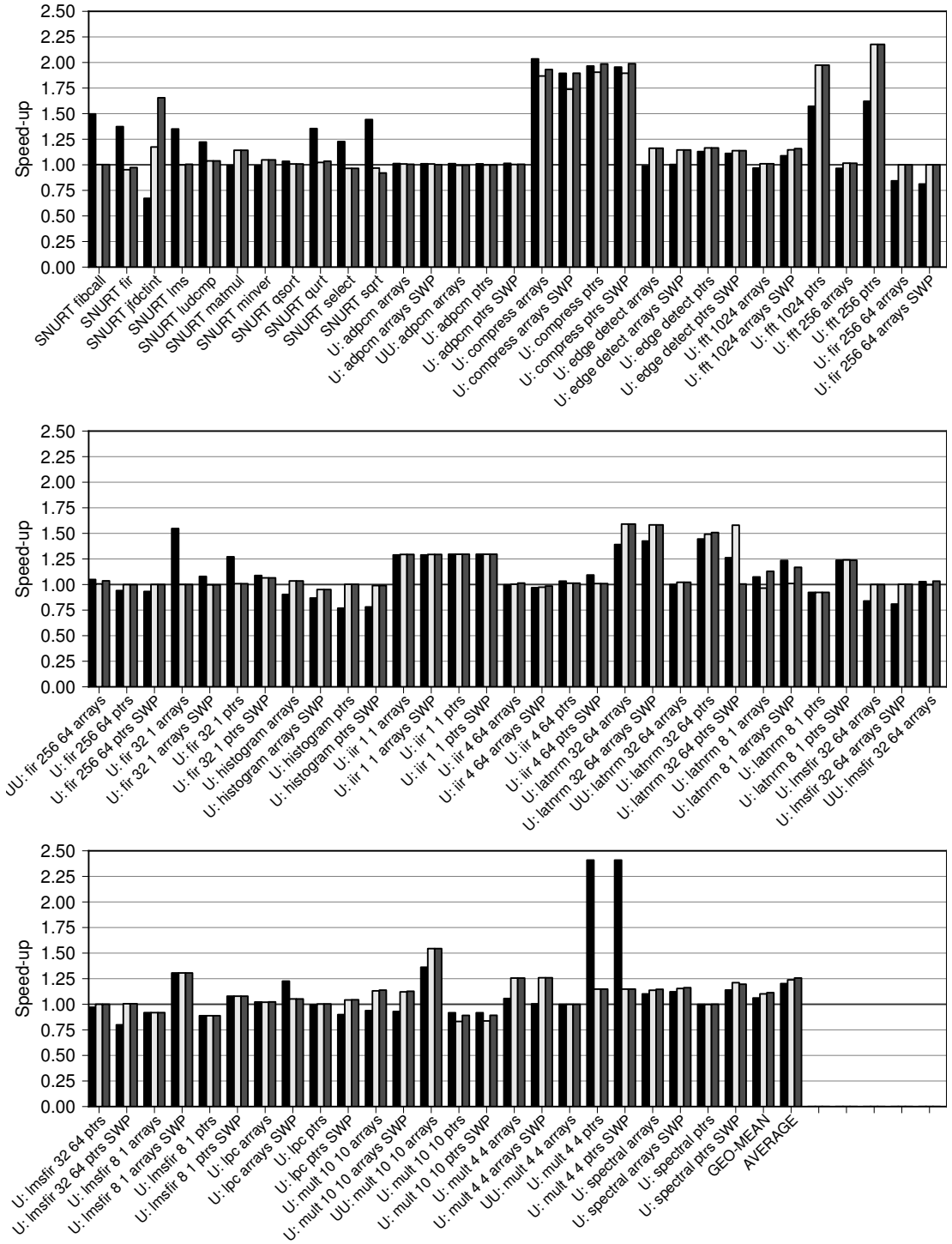


Figure B.32 (continued): *of 1.25 cycles alone, and the two combined as single run.*

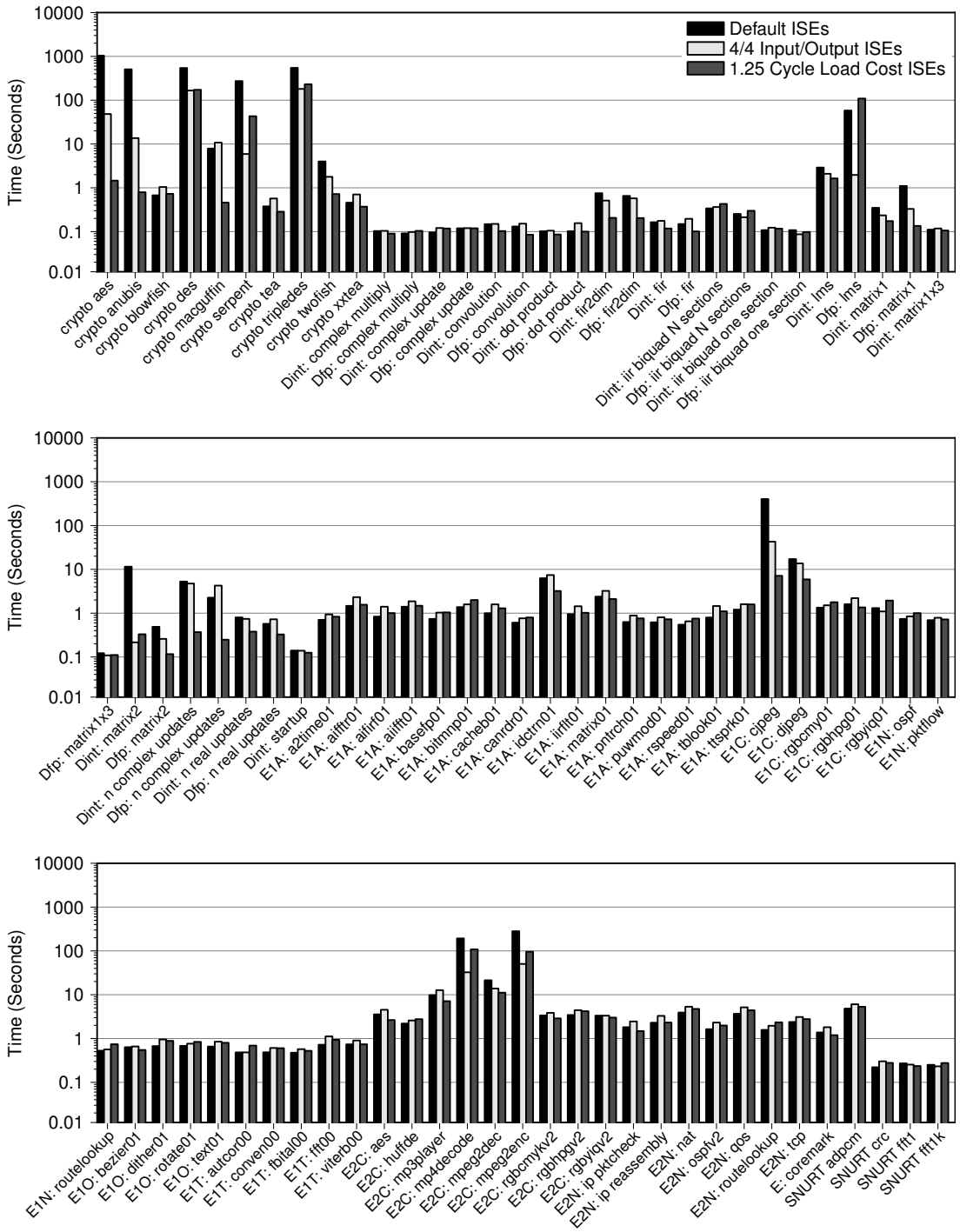


Figure B.33: Note: this is the full version of figure 5.7 on page 85. *A runtime comparison for* MapISE *when presented with default extension instructions,*
(Continued on the next page.)

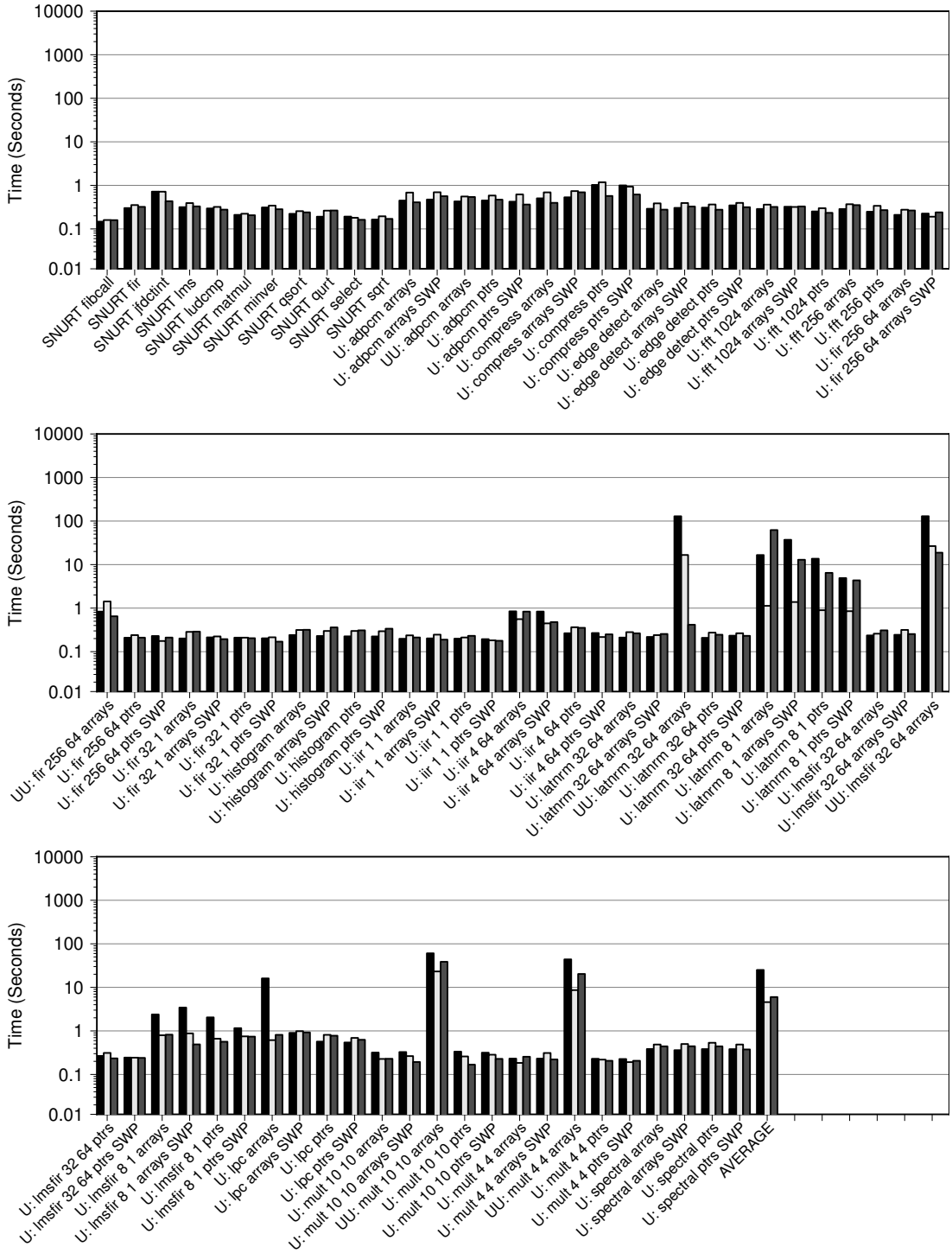


Figure B.33 (continued): *instructions with a 4/4 input/output constraint, and instructions generated with a 1.25 cycle load cost.*

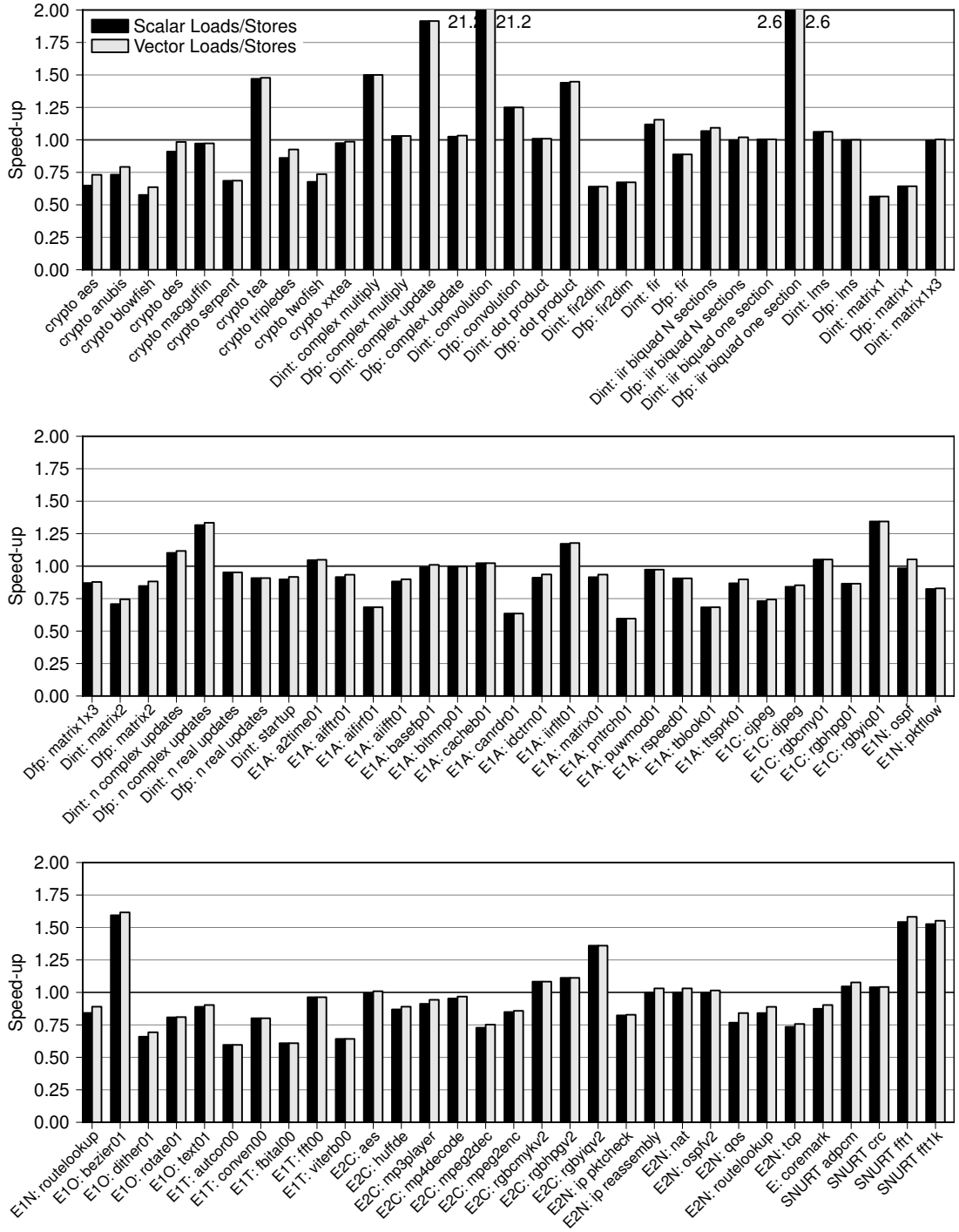


Figure B.34: Note: this is the full version of figure 5.8 on page 86. *A comparison of a system with no vector loads or stores (the default)* (Continued on the next page.)

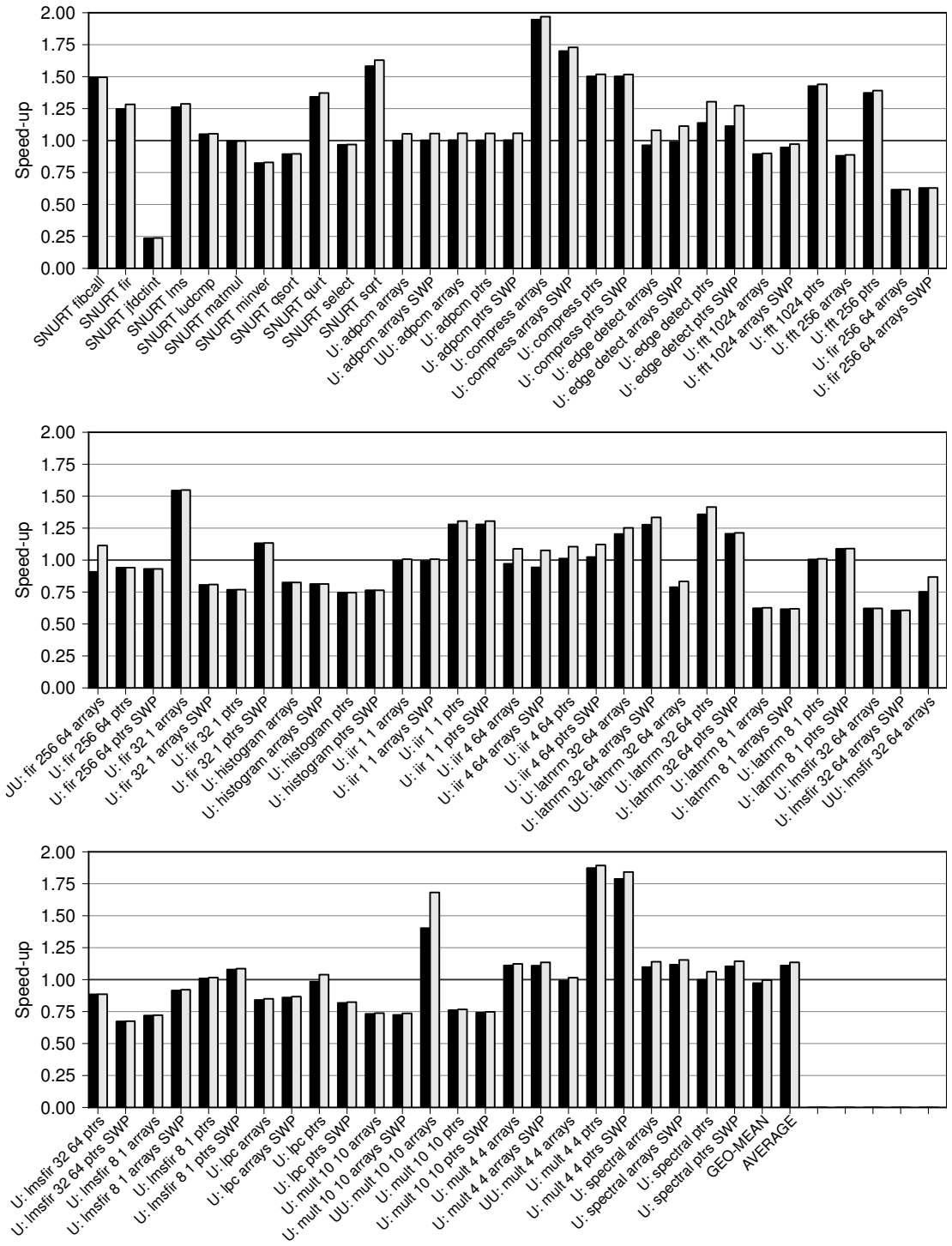


Figure B.34 (continued): *with one that does have them.*

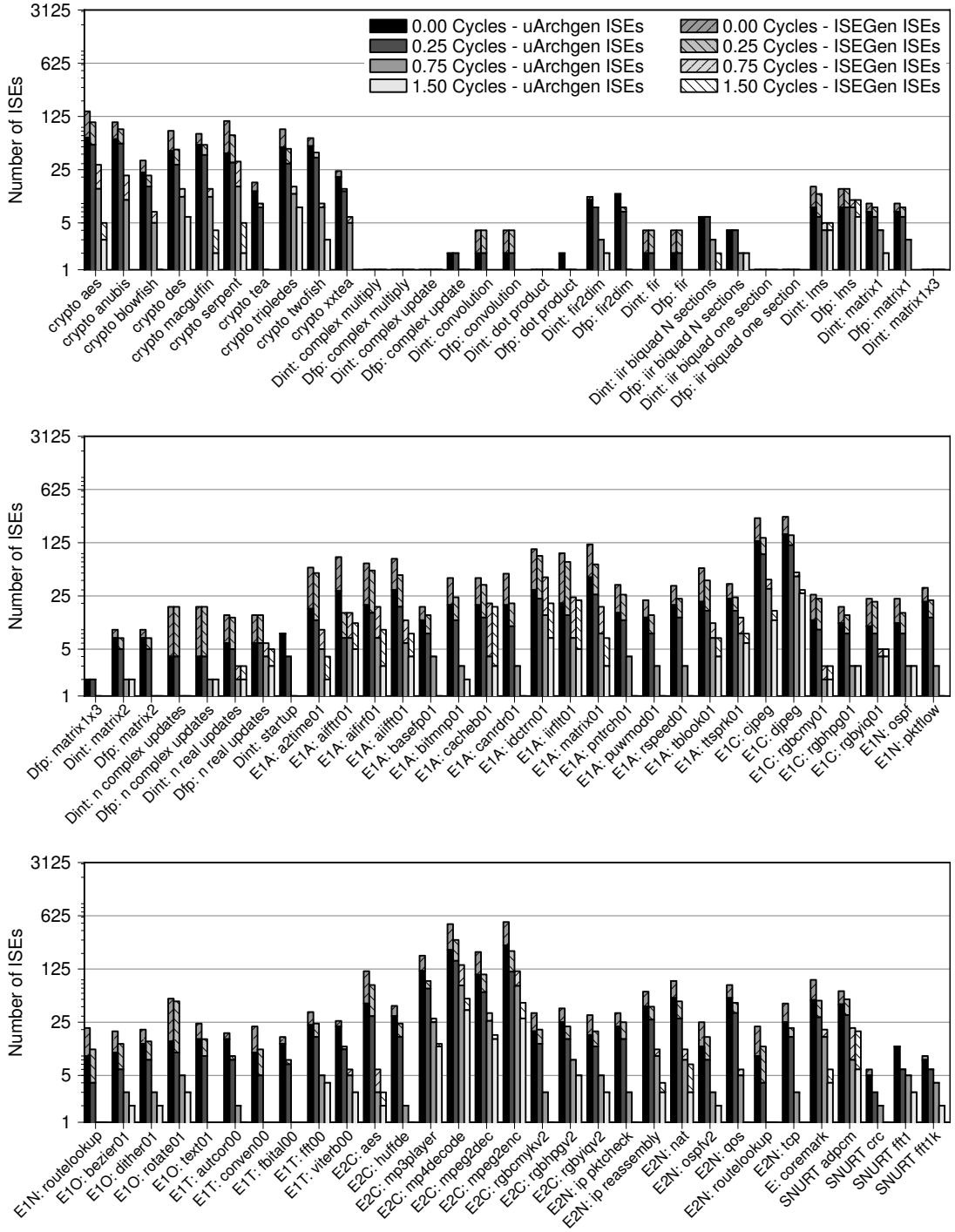


Figure B.35: *Note: This is a variation of figure B.27, which, in turn, is the full version of figure 5.4(a) on page 83.* (Continued on the next page.)

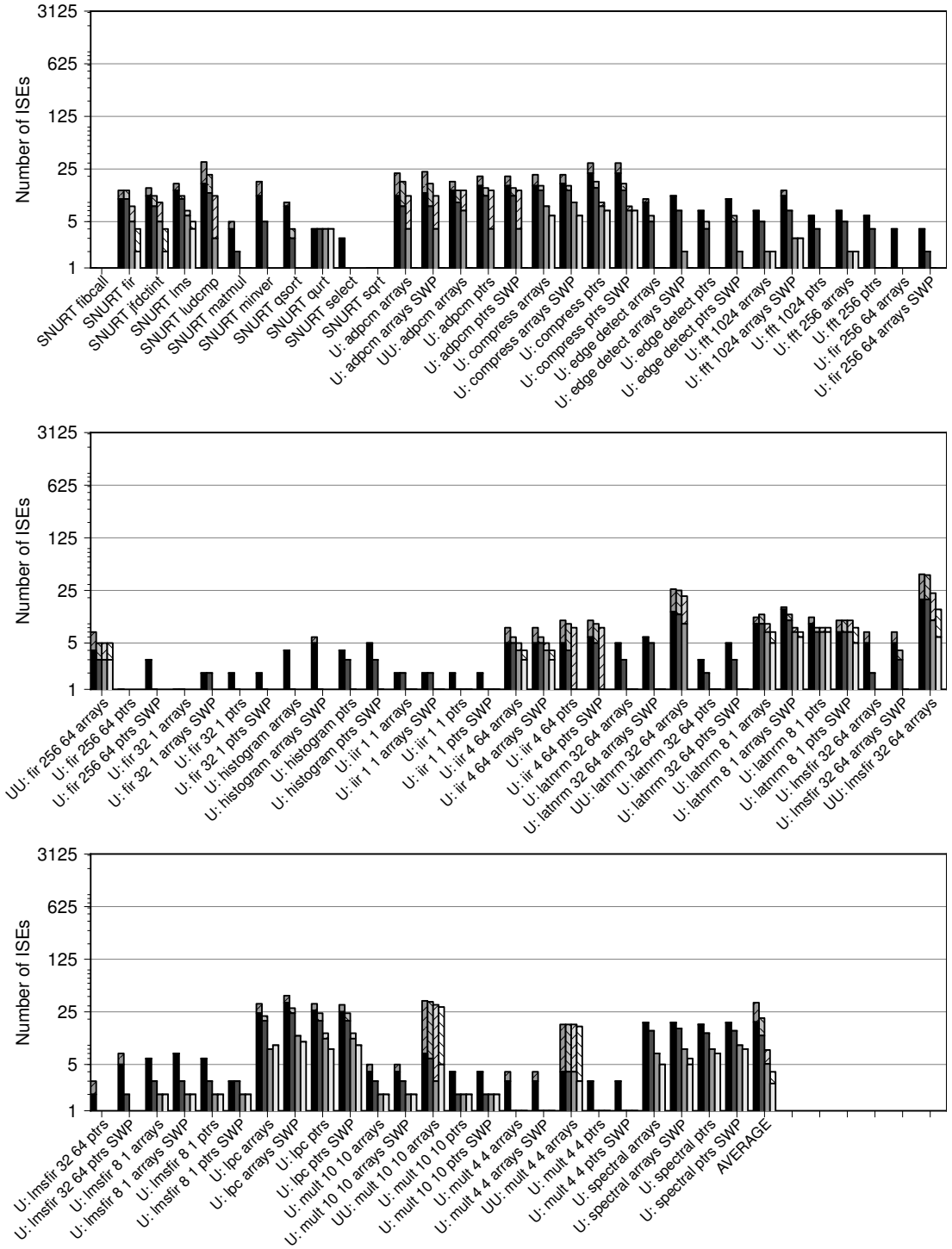


Figure B.35 (continued): *These charts contain three parameter values that were not included in figures 5.5 or B.30.*

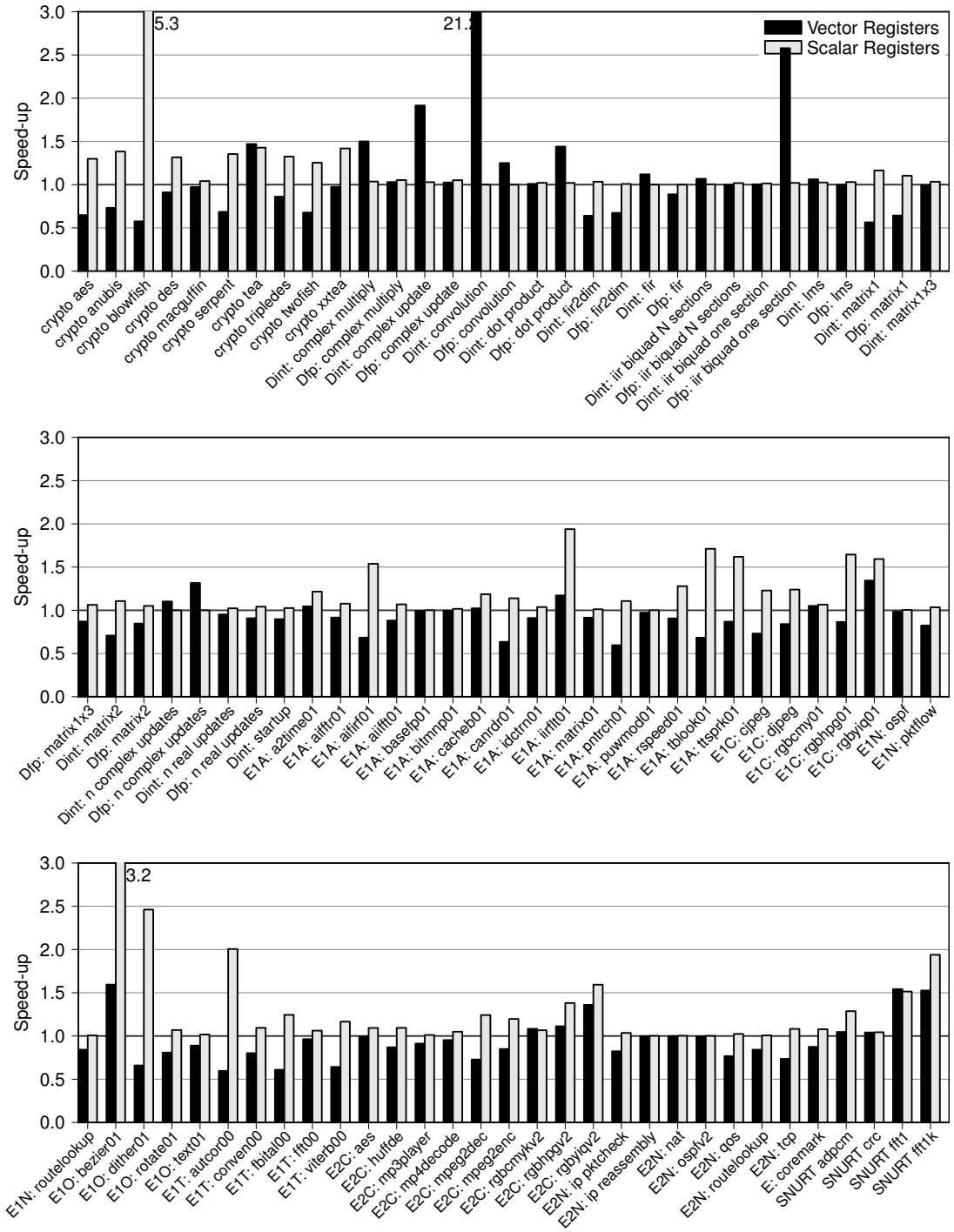


Figure B.36: Note: this is the full version of figure 5.9 on page 87. (Continued on the next page.)

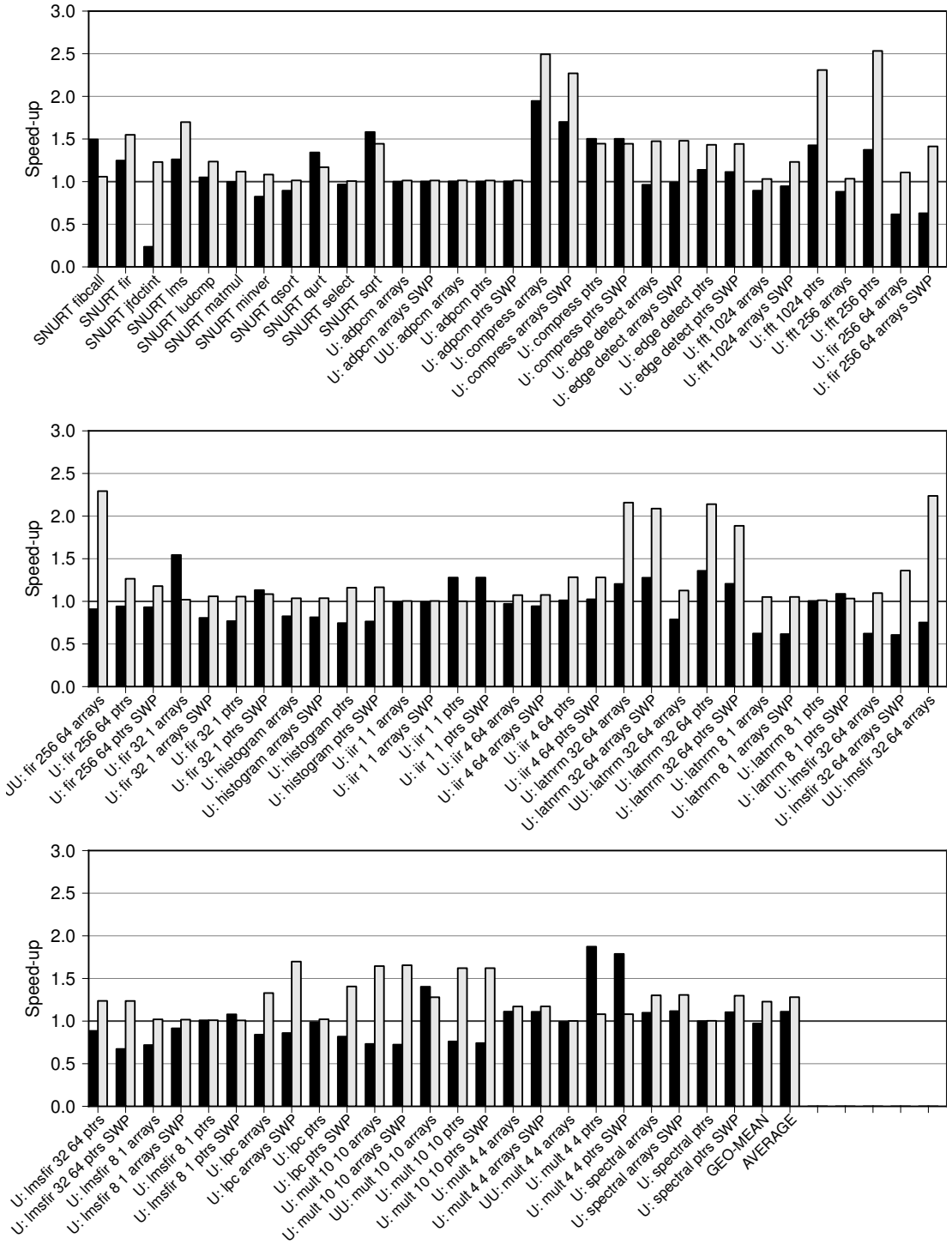


Figure B.36 (continued): *The speed-ups provided by using extension instructions with vector registers (left-hand bar) or scalar registers (right-hand bar).*

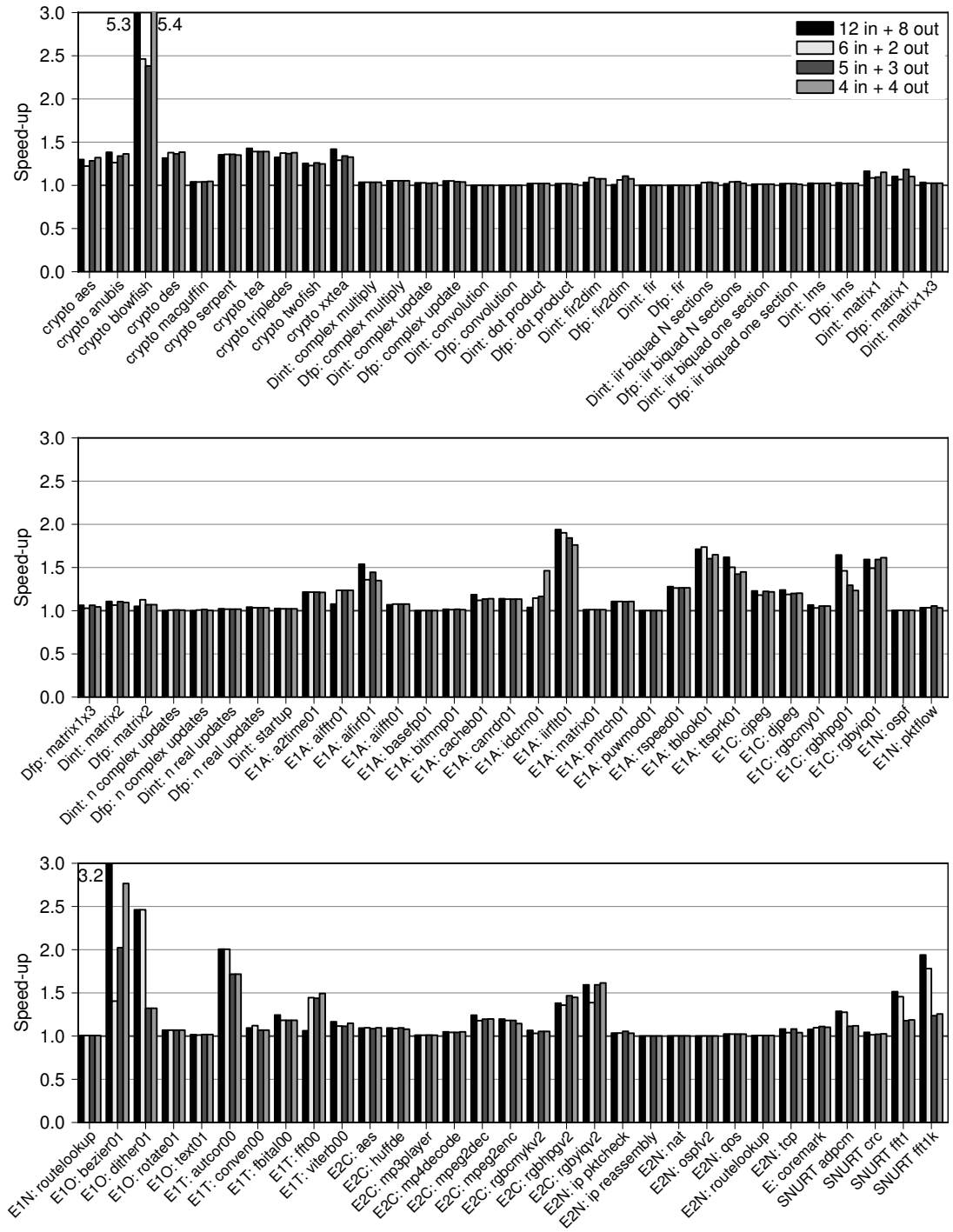


Figure B.37: Note: this is the full version of figure 5.10(a) on page 88. *An evaluation of different input and output constraints for scalar register based extension instructions.* (Continued on the next page.)

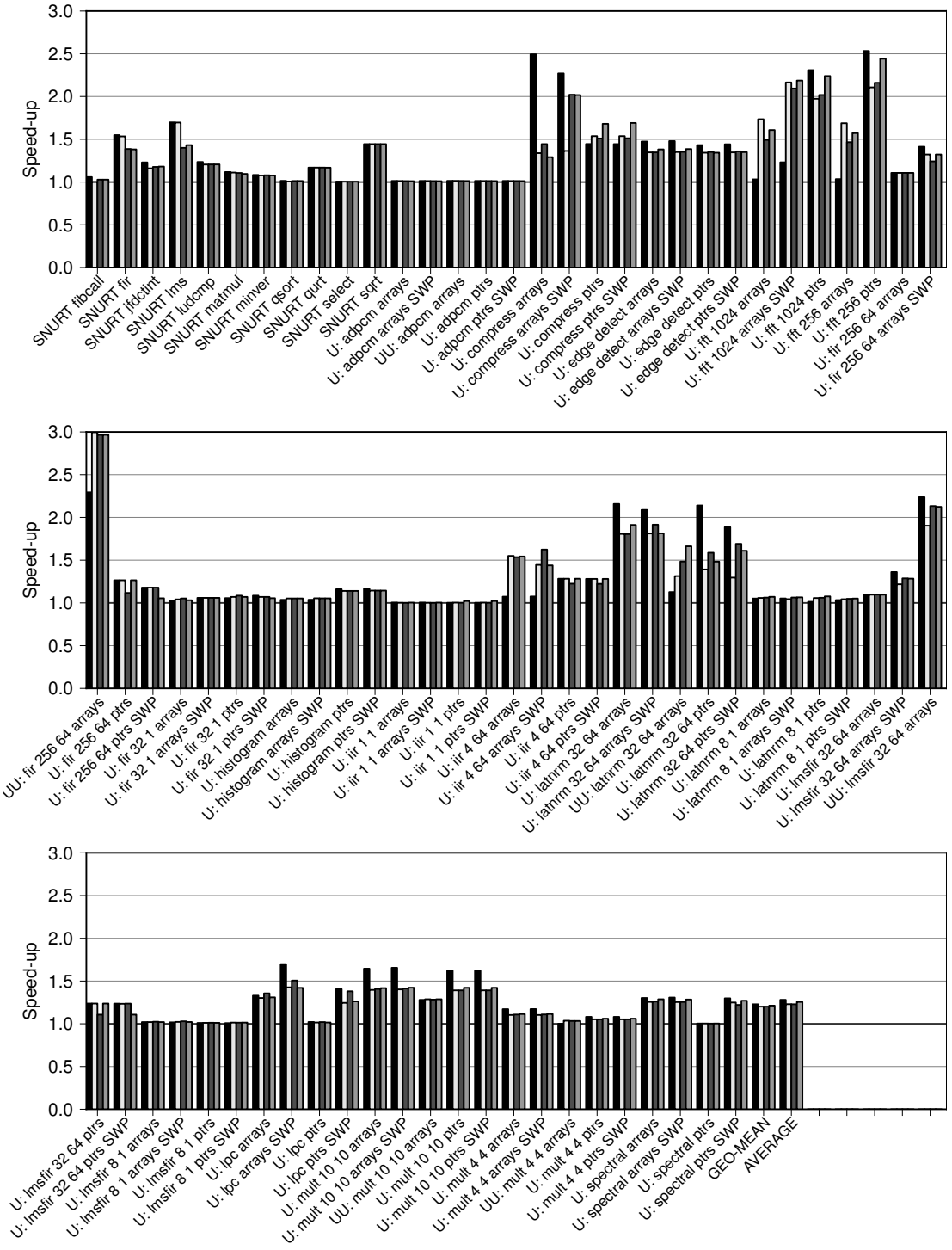


Figure B.37 (continued): *Speed-up.*

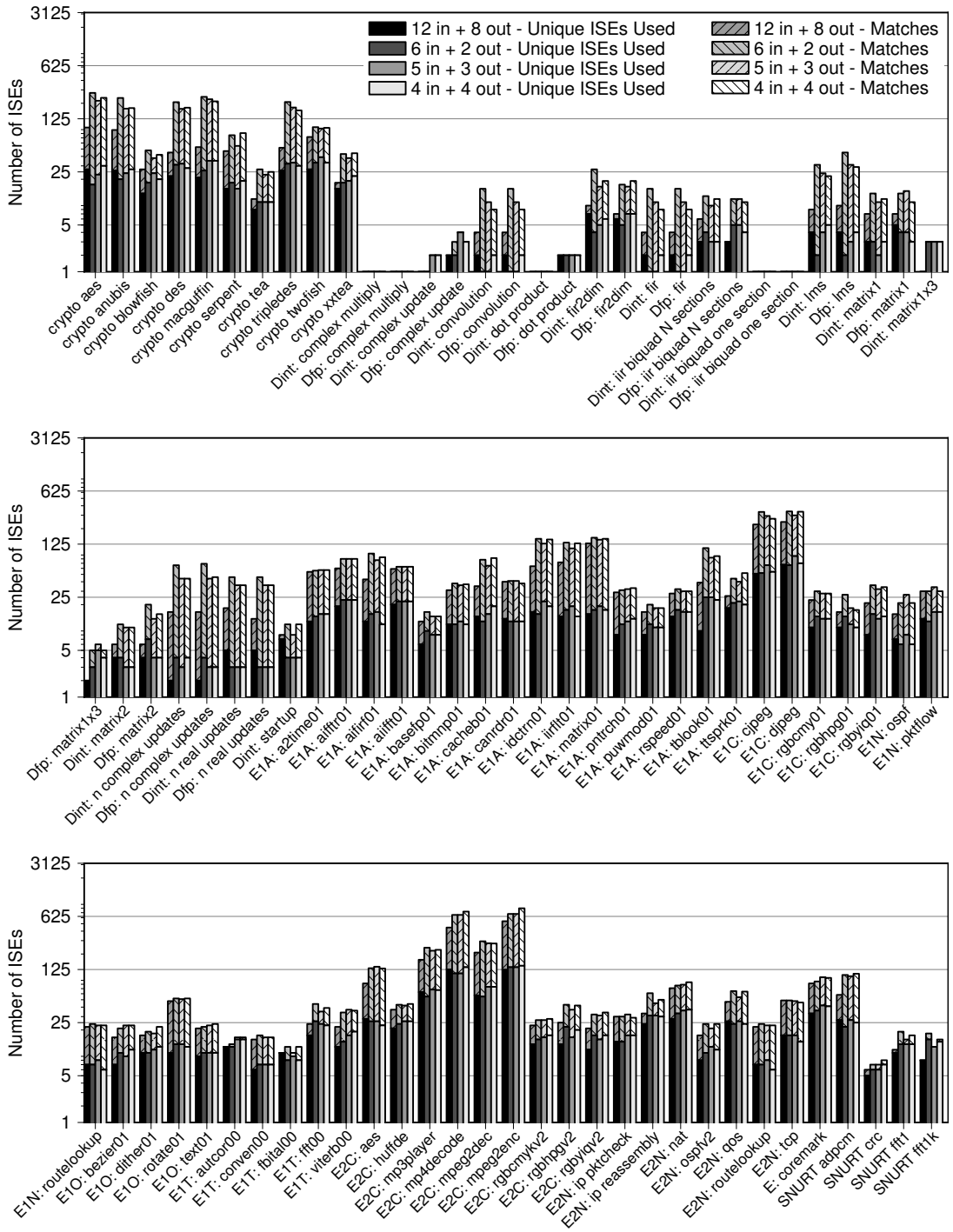


Figure B.38: Note: this is the full version of figure 5.10(b) on page 88. *An evaluation of different input and output constraints for scalar register based extension instructions.* (Continued on the next page.)

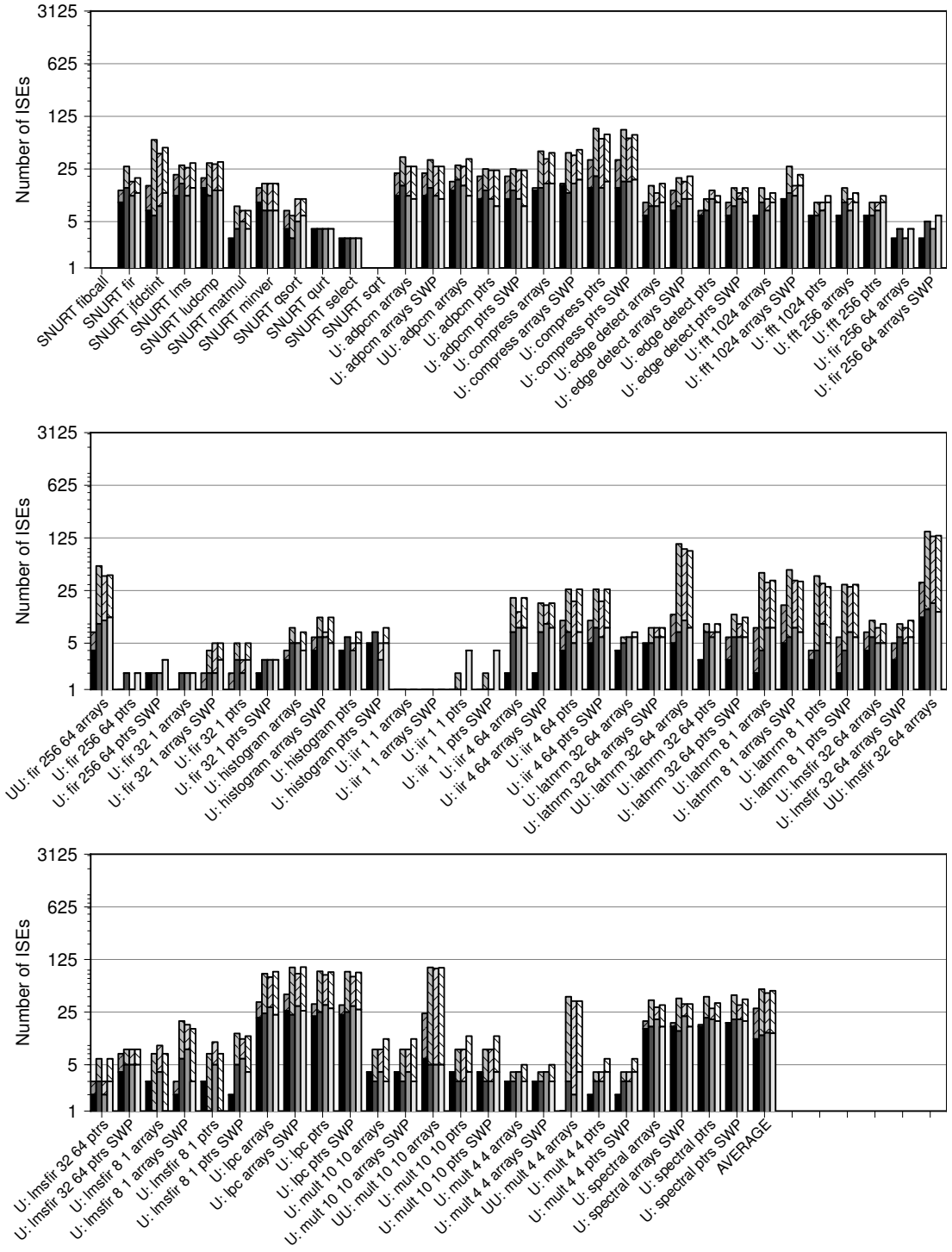


Figure B.38 (continued): *Mapping quality information.*

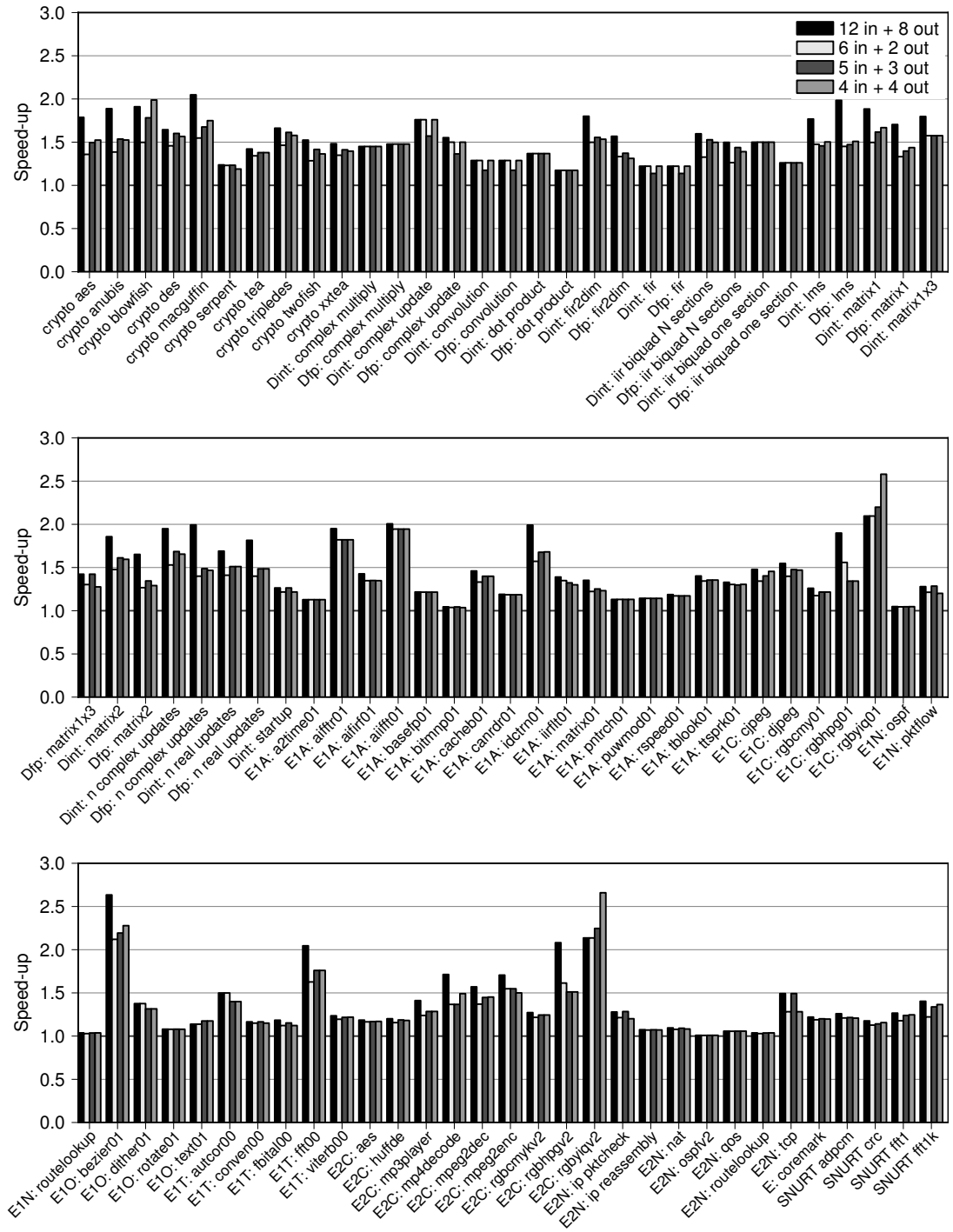


Figure B.39: Note: this is the full version of figure 5.11 on page 89. ISEGen's *predicted speed-ups for various register constraints which are* (Continued on the next page.)

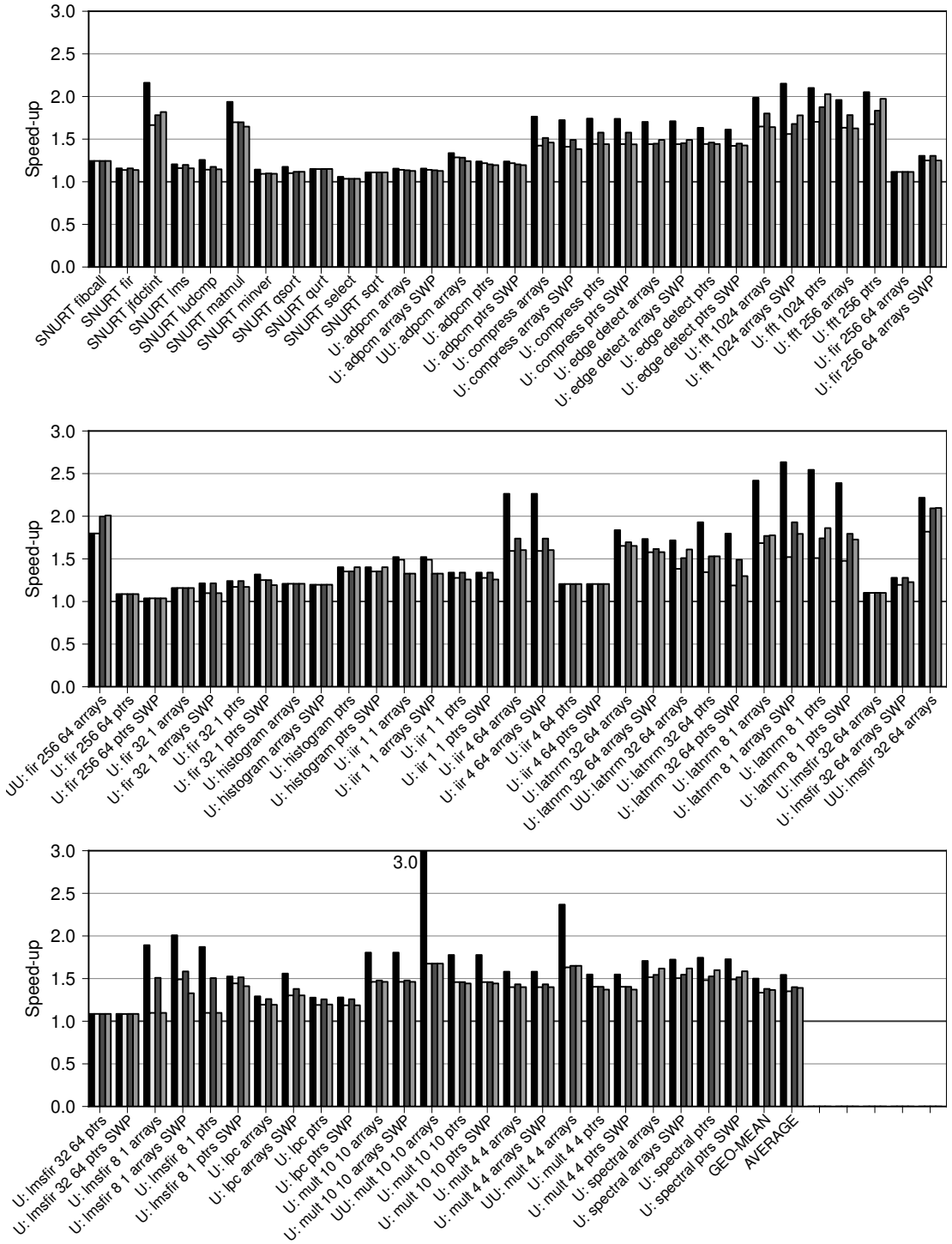


Figure B.39 (continued): *realistic for scalar register based extension instructions, the default 12-input, 8-input mode is also included for comparison.*

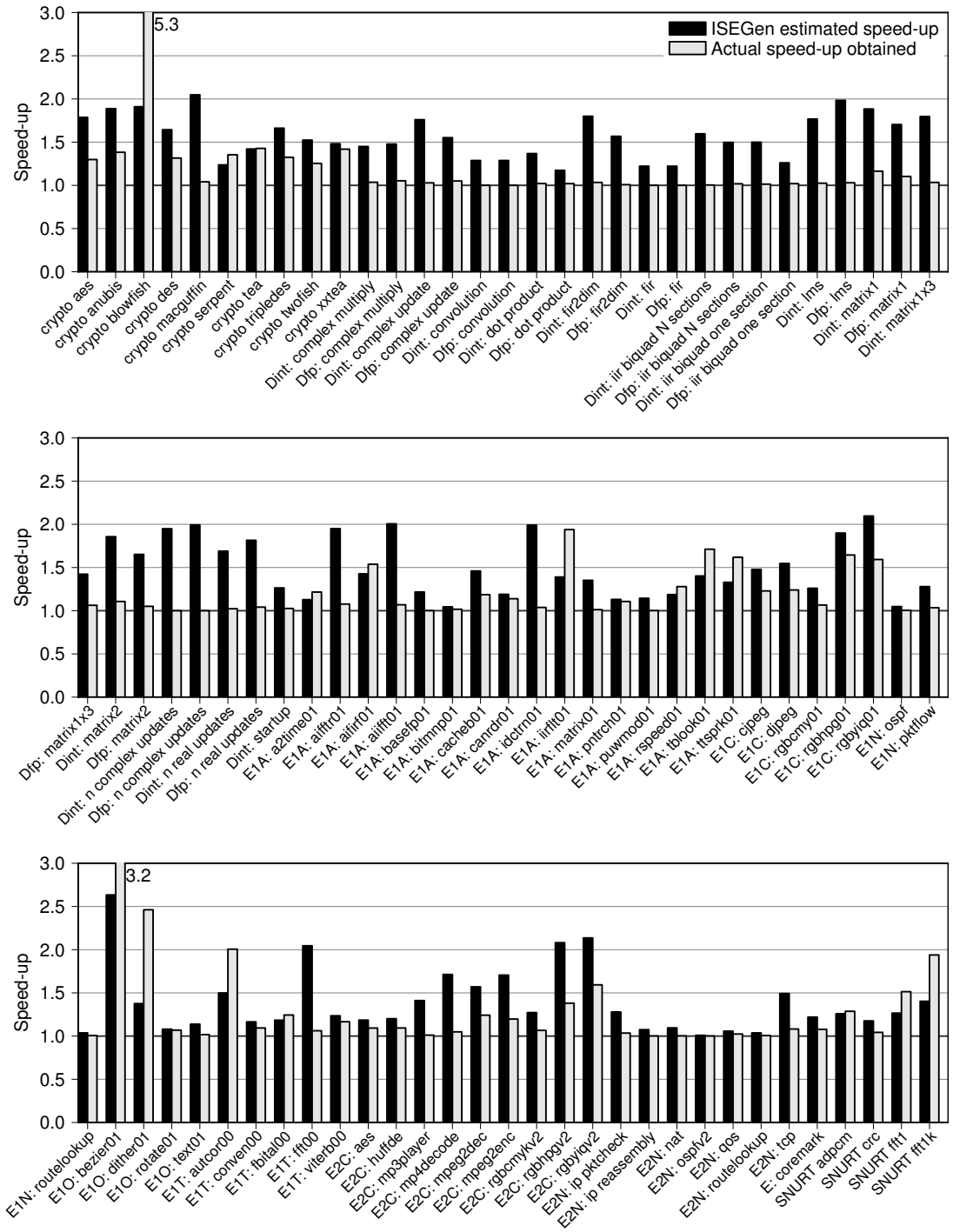


Figure B.40: Note: this is the full version of figure 5.12 on page 90. *A re-evaluation of the comparison of* MapISE's *performance with* ISEGen's, (Continued on the next page.)

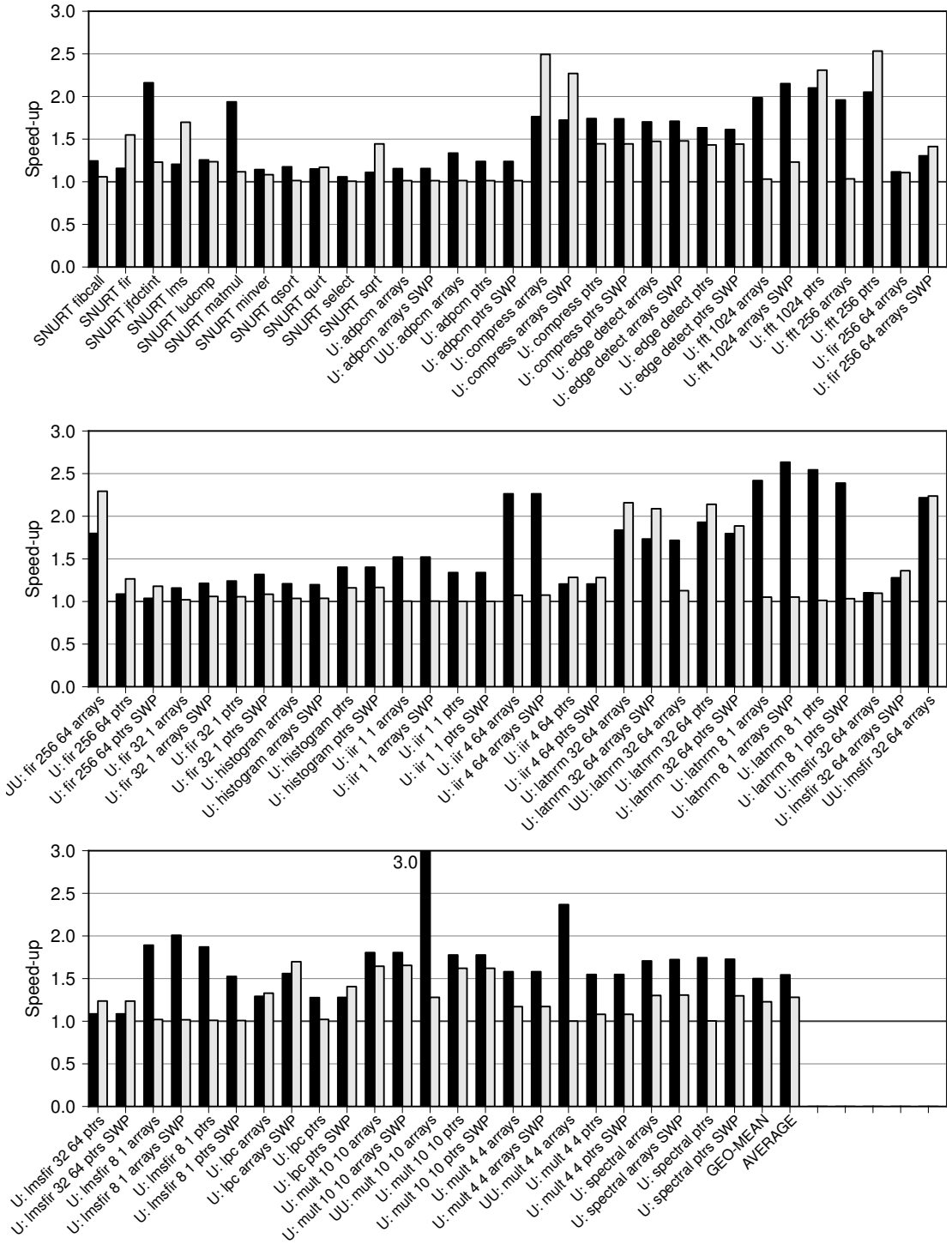


Figure B.40 (continued): *but now* MapISE *is using scalar register based extension instructions (or "wide instruction").*

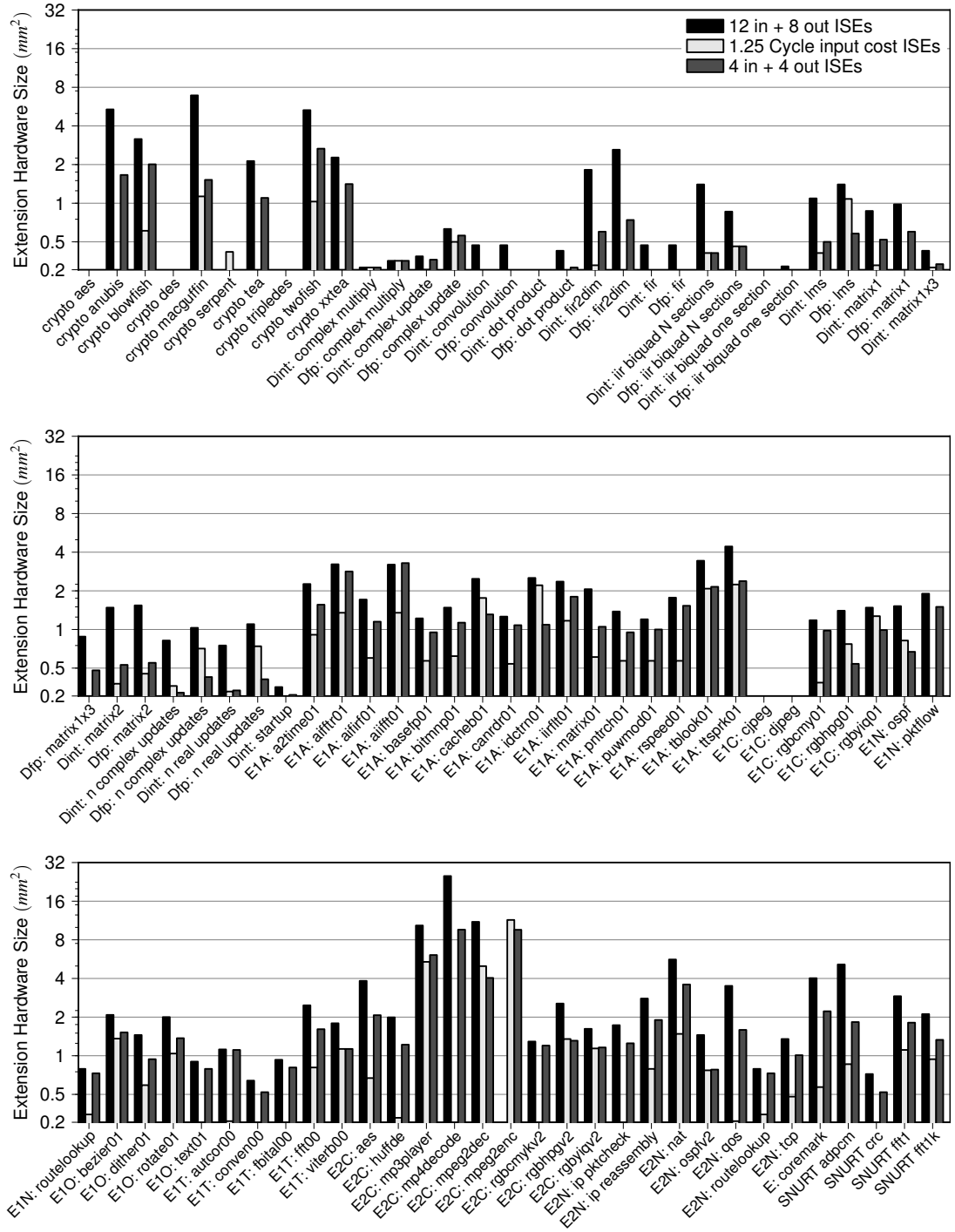


Figure B.41: Note: this is the full version of figure 5.13 on page 91. *Extension unit on-die hardware size per-benchmark* (Continued on the next page.)

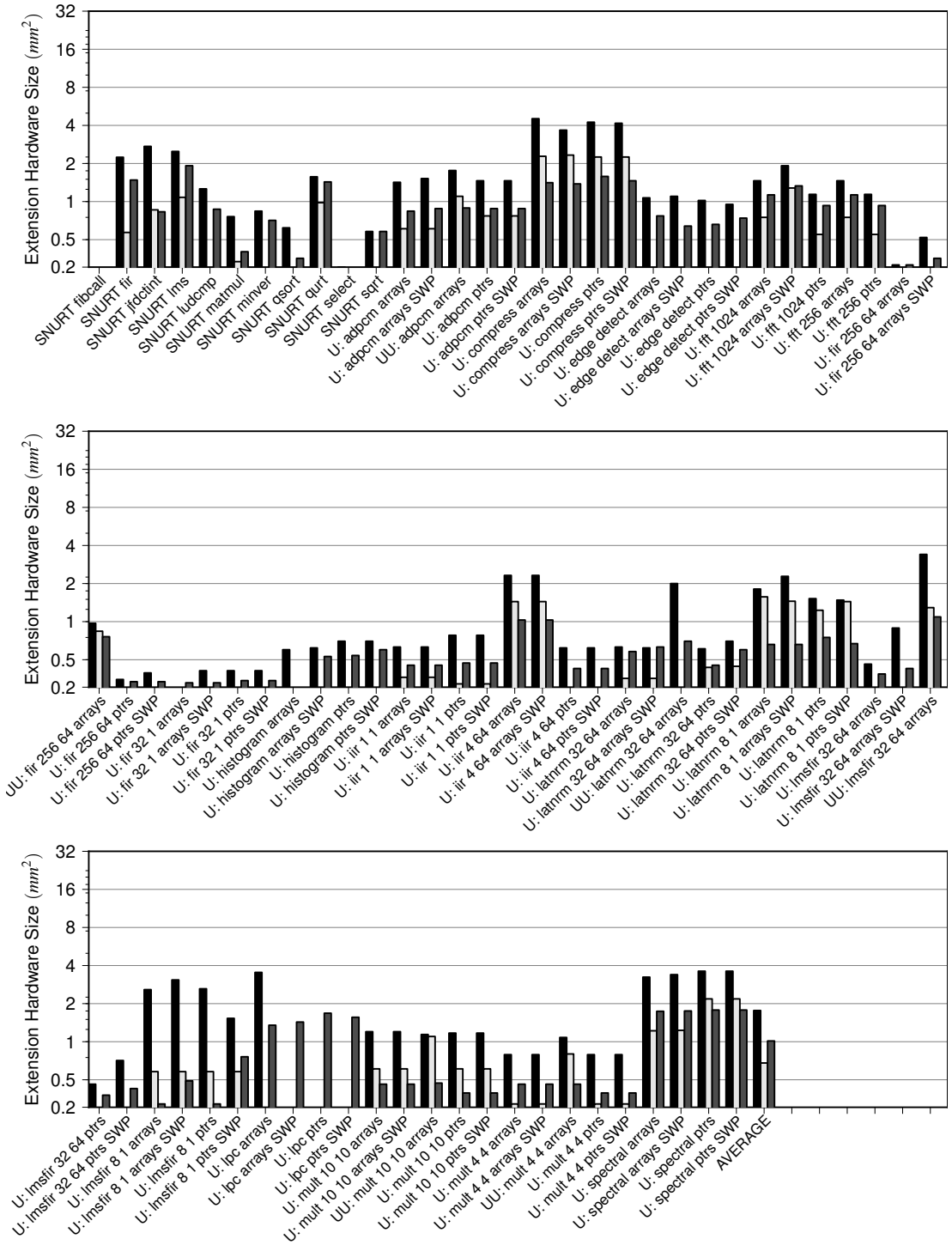


Figure B.41 (continued): *for different hardware configurations, based on a* $130nm$ *process.*

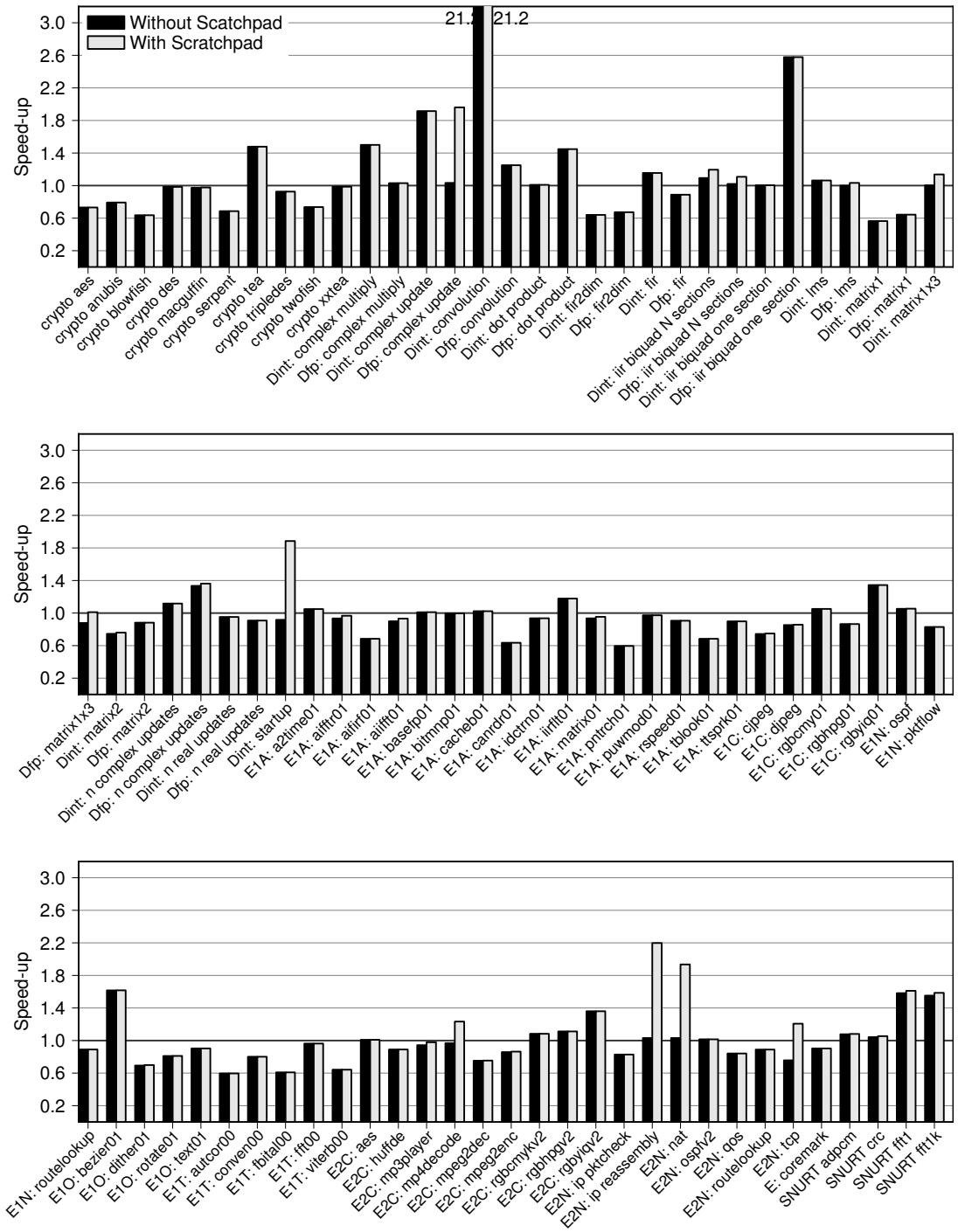


Figure B.42: Note: this is the full version of figure 6.1 on page 94. *These results are speculative estimates (see text) and are not as reliable as related results in chapters 4 and 5.* (Continued on the next page.)

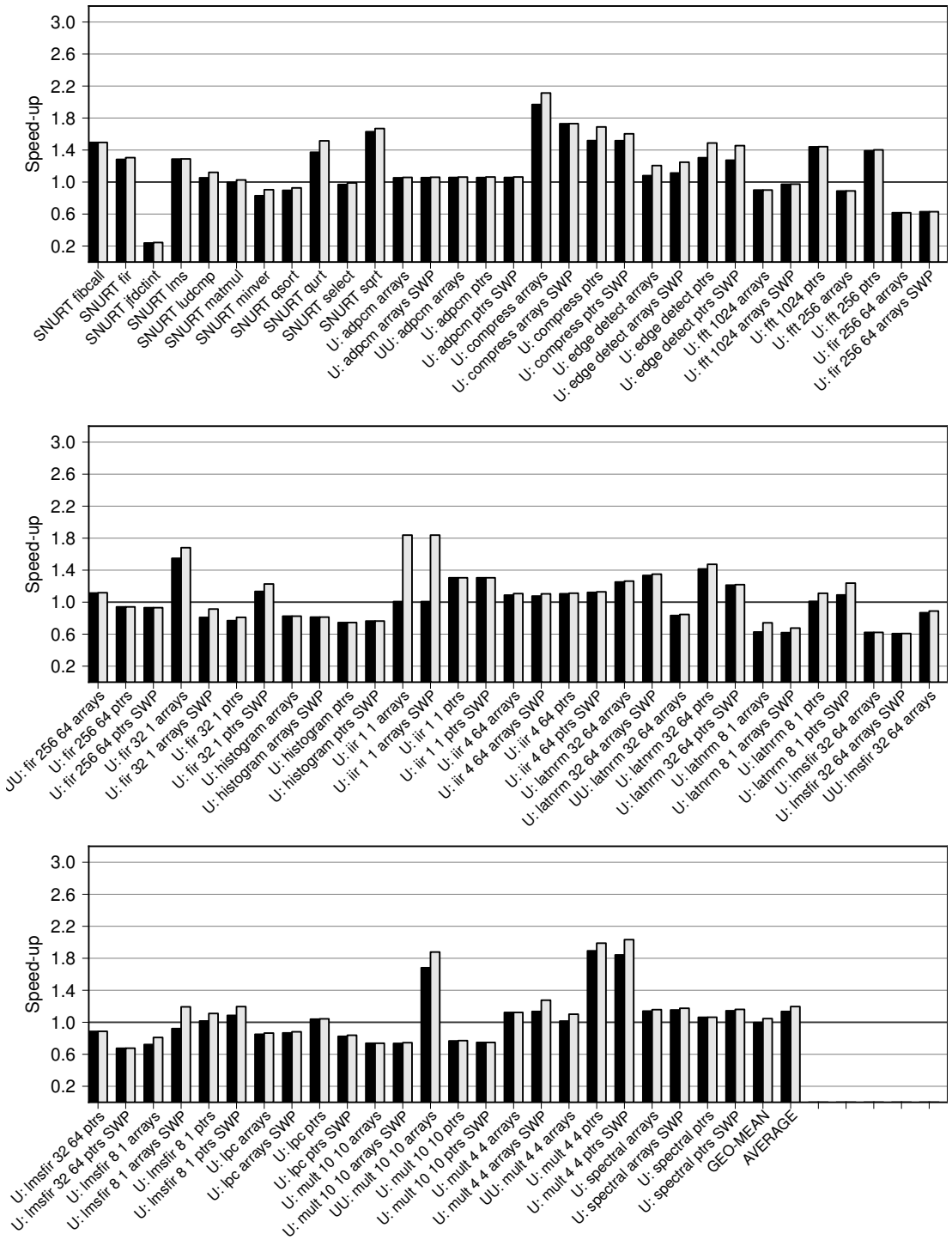


Figure B.42 (continued): *The left bar for each benchmark is the speed-up achieved by adding extension instructions with vector registers and vector load/stores to the baseline processor, as described in section 5.2. The right bars add a scratchpad memory to this.*

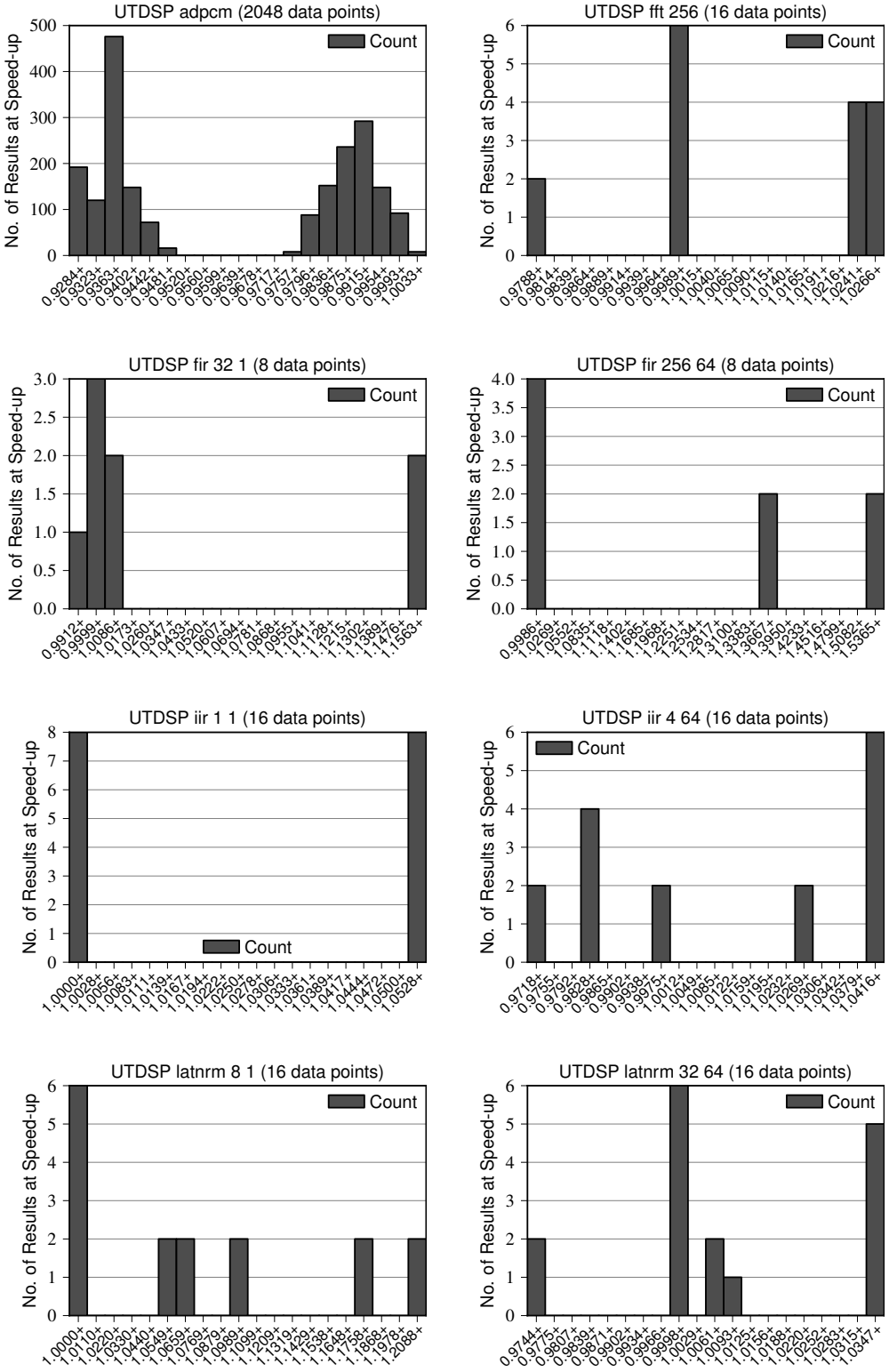


Figure B.43: Note: this is the full version of figure 6.8 on page 114. (Continued on the next page.)

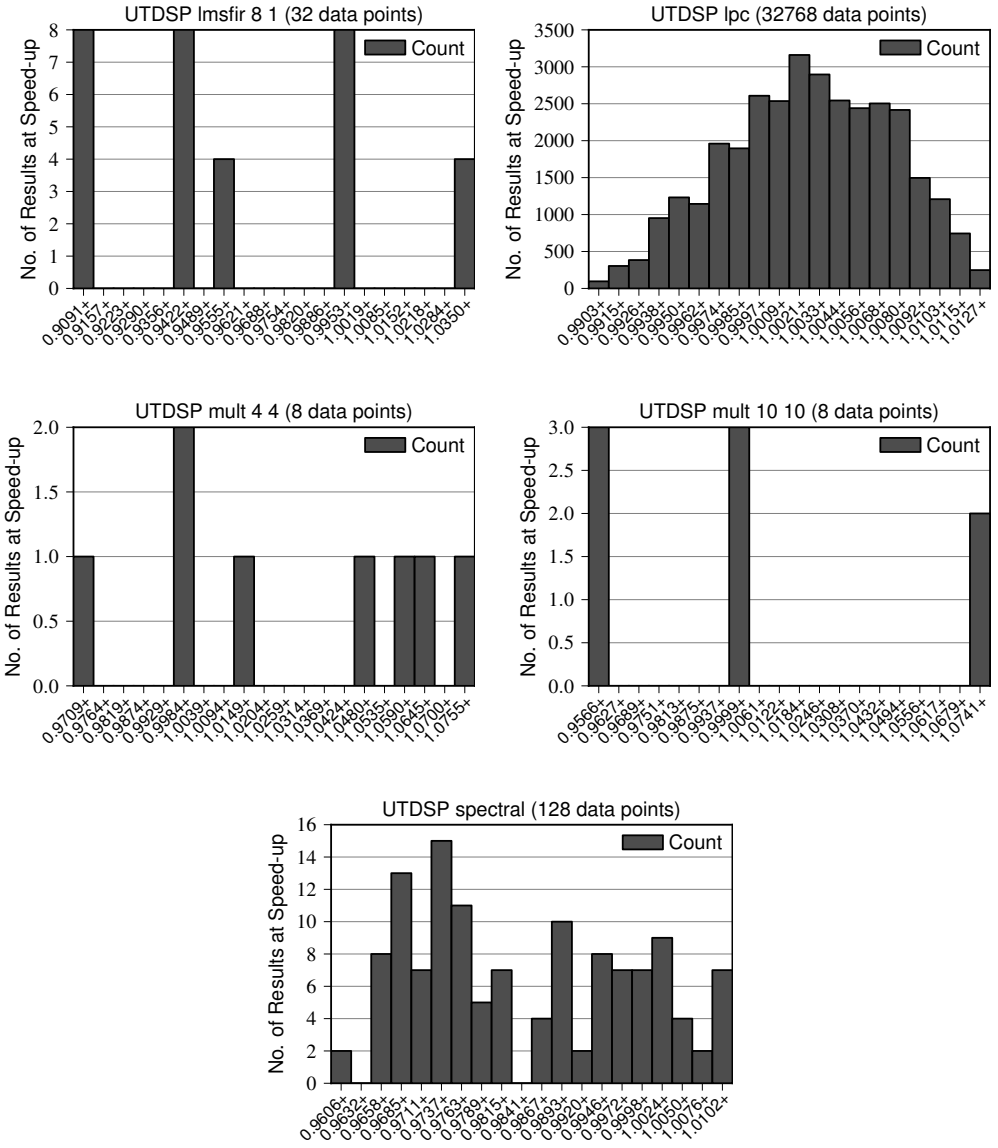


Figure B.43 (continued): *The distribution of the performance of the exhaustive set of memory bank assignments. Note that many of the benchmarks only have a small number of possible assignments which results in a disjoint distribution.*

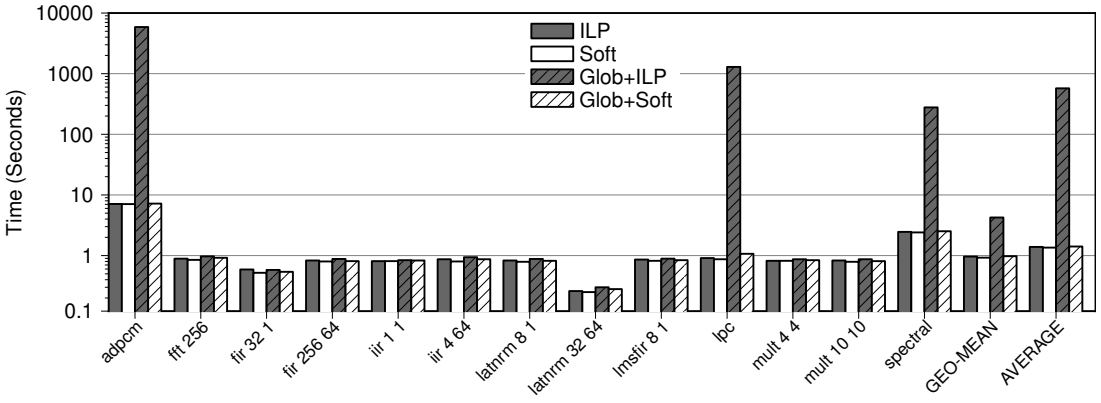


Figure B.44: Note: this is the full version of figure 6.10(a) on page 116. *Time taken to perform memory assignment.*

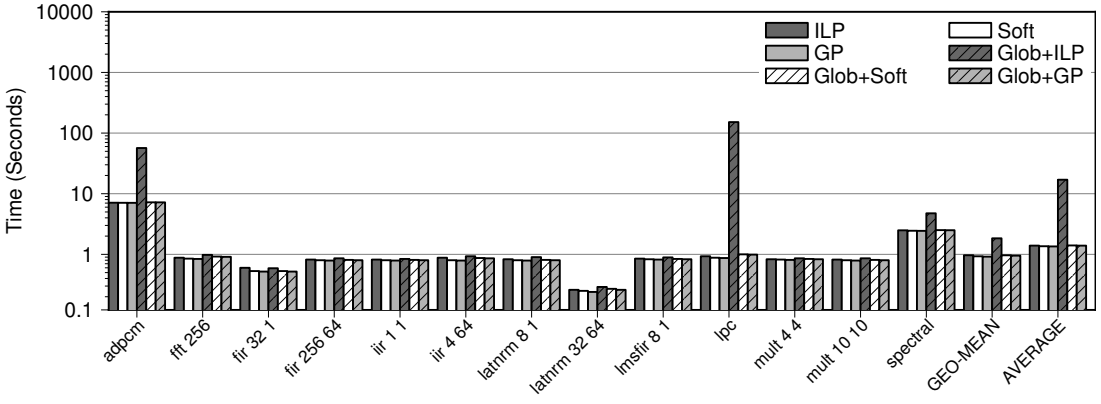


Figure B.45: Note: this is the full version of figure 6.10(b) on page 116. *Timings with an additional* automatic node.

Appendix C

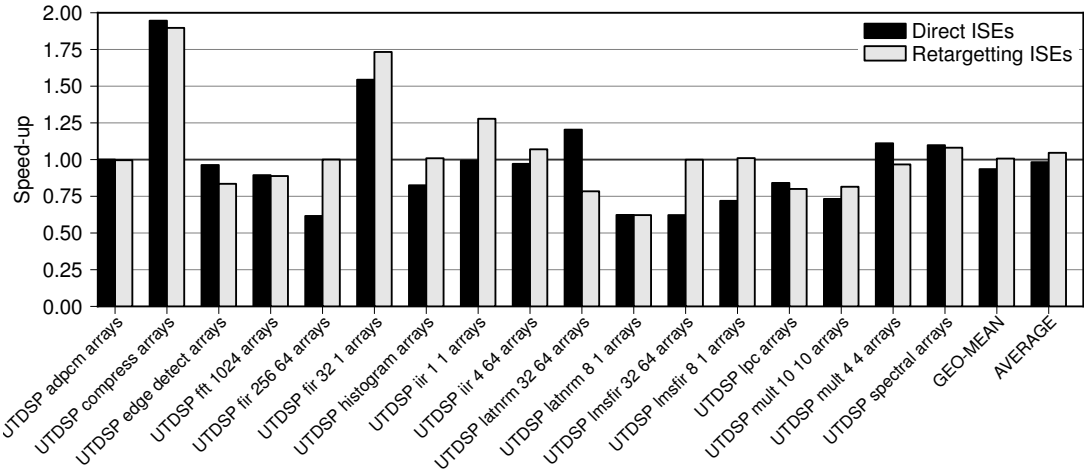
Retargeting Extension Instructions

"The first 90% of the code accounts for the first 90% of the development time. The remaining 10% of the code accounts for the other 90% of the development time."
— Tom Cargill, *Computer Programmer*, *Writer*.

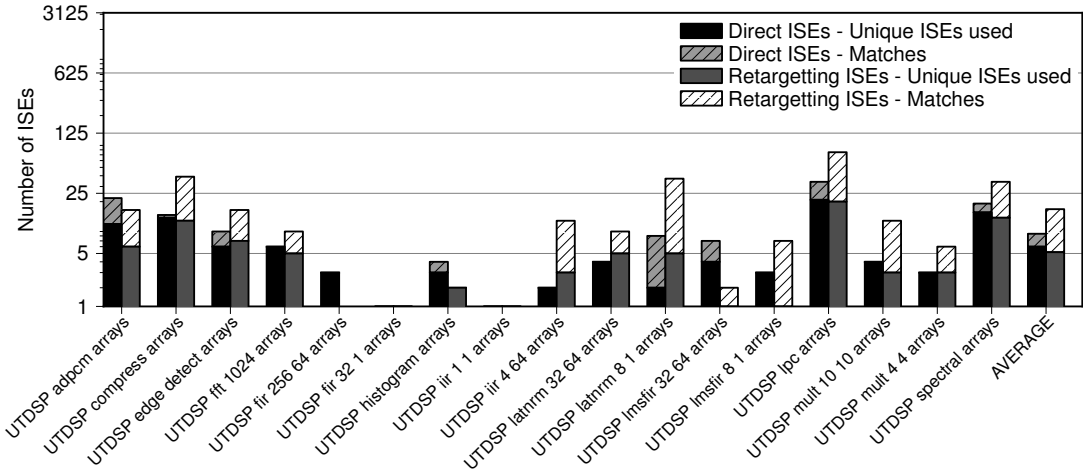
The retargeting of extension results is considered in this appendix. The experiments performed in section 4.8 are repeated for three of the changes proposed in chapter 5: "4/4 Input/Output Registers", "Hard-Wiring Constant Values" and "Wide Instructions". The results are not summarised as they do not demonstrate anything that was not already shown in section 4.8 – they are just included in this appendix so as that may be verified. The only difference in the results is that "Wide Instructions" seem to be slightly worse for retargeting than vector register based extension instructions. This seems surprising and may just be an artifact of the evaluation model.

C.1 Reducing the Number of I/O Ports

Evaluates the technique presented in section 5.1.1 in the context of retargeting extension instructions.

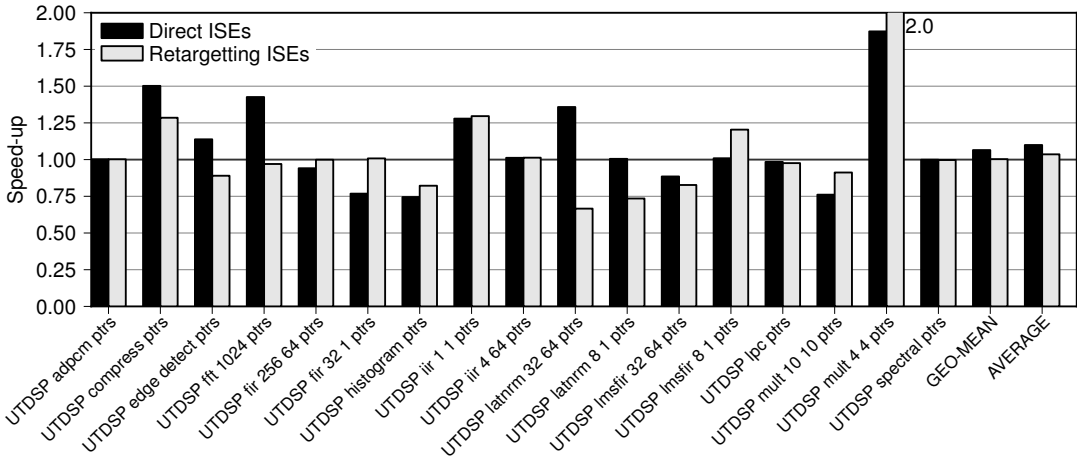


(a) Speed-ups.

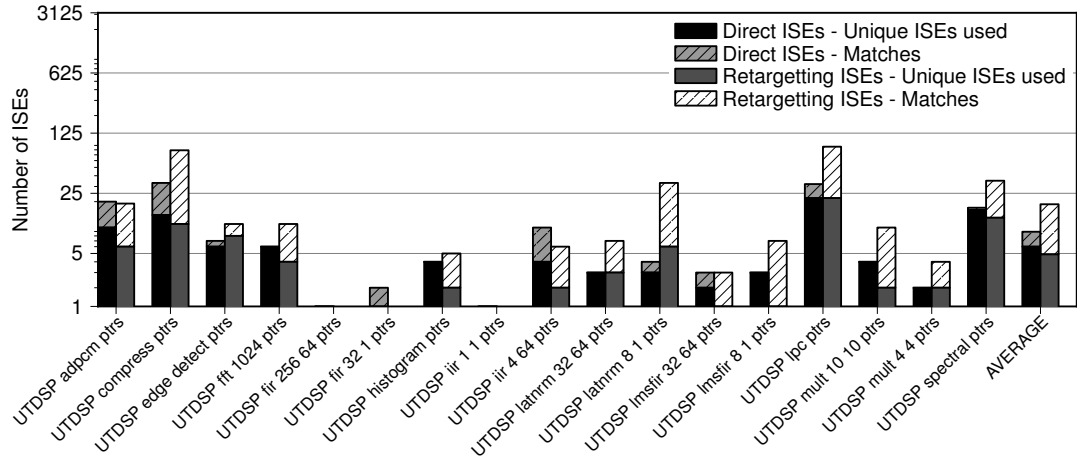


(b) Mapping quality information.

Figure C.1: *Extension instructions are generated for* ptrs *benchmarks and then exploited on* arrays *benchmarks. Both data-sets are for 4 input + 4 output extension instructions.*

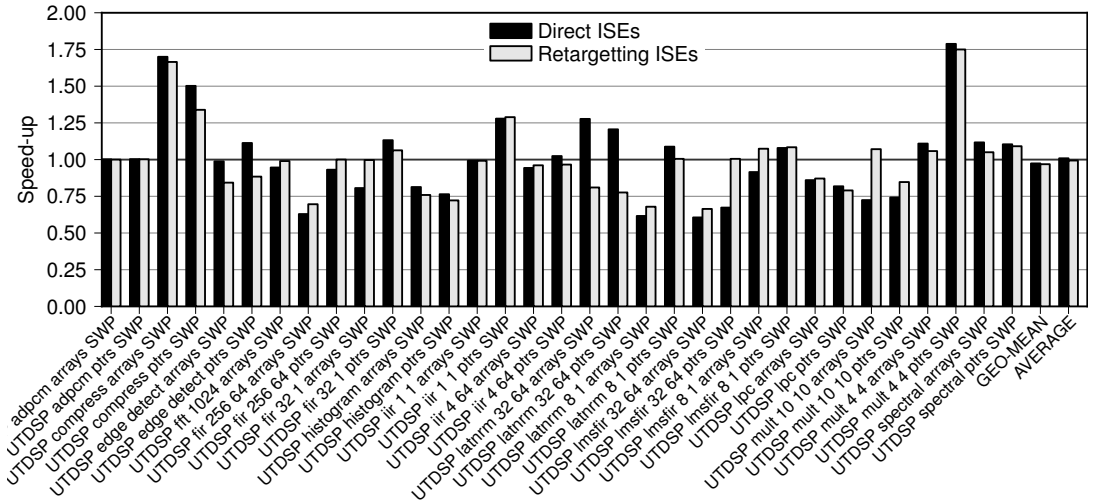


(a) Speed-ups.

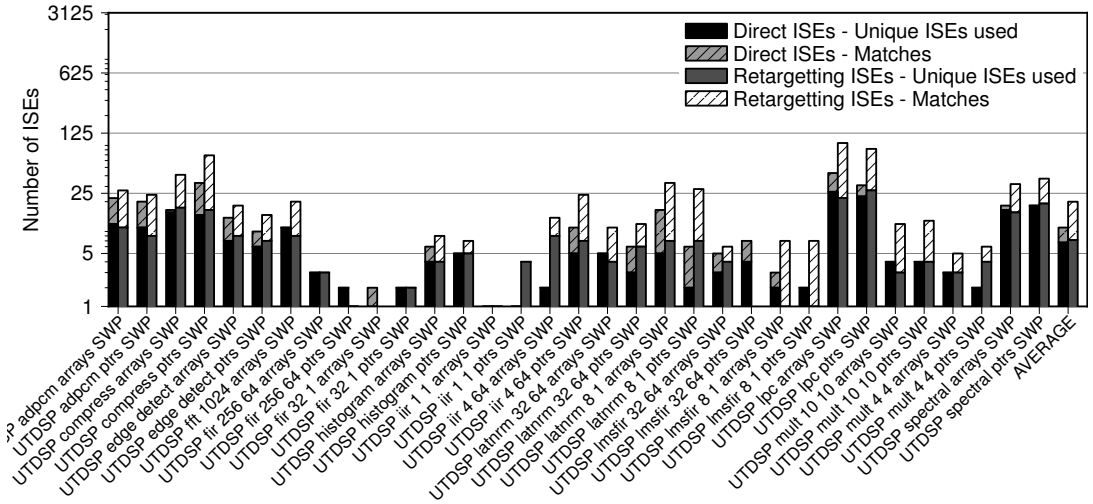


(b) Mapping quality information.

Figure C.2: *Extension instructions are generated for* arrays *benchmarks and then exploited on* ptrs *benchmarks. Both data-sets are for 4 input + 4 output extension instructions.*

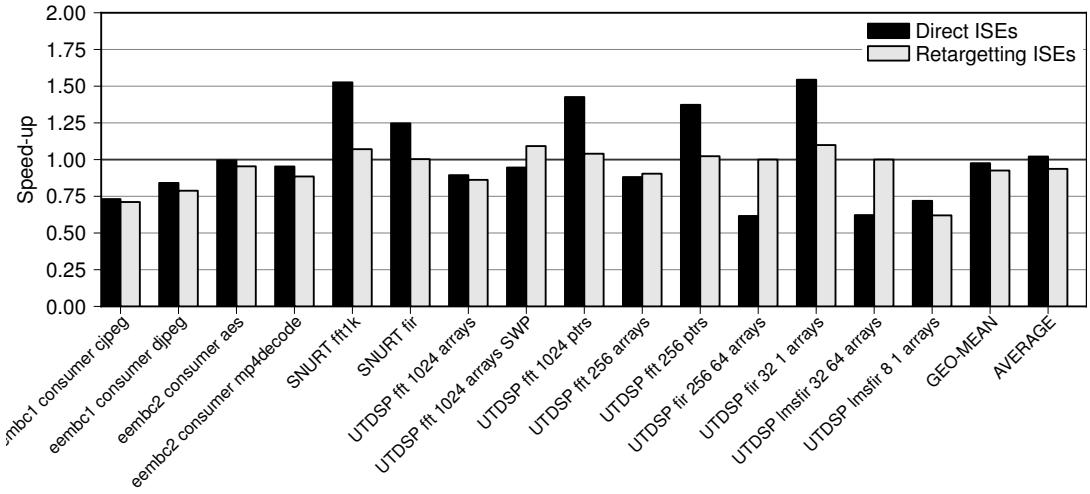


(a) Speed-ups.

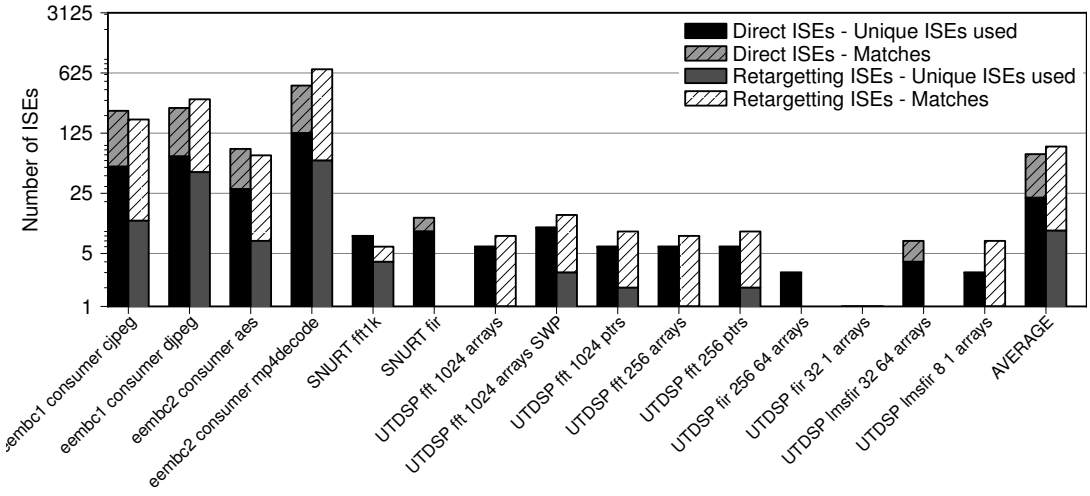


(b) Mapping quality information.

Figure C.3: *Extension instructions are generated for* arrays *benchmarks and then exploited on* arrays-SWP *benchmarks. Both data-sets are for 4 input + 4 output extension instructions.*

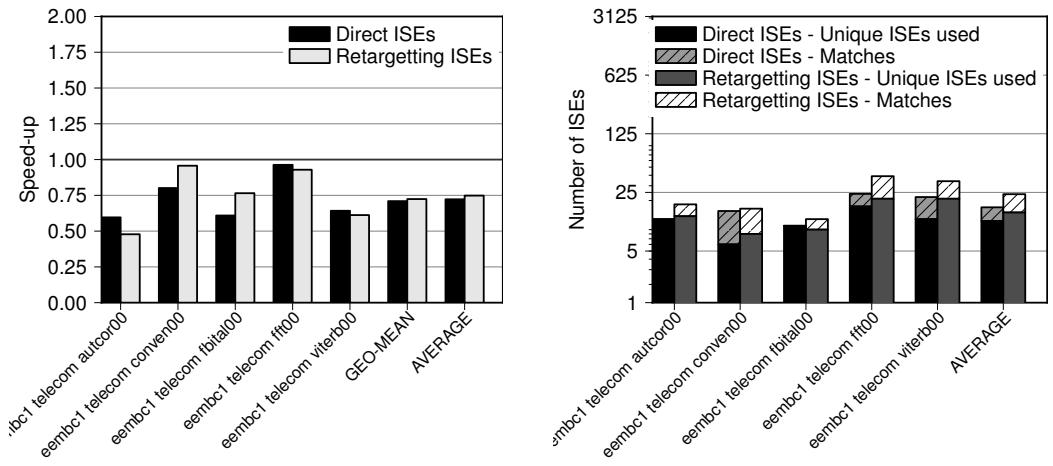


(a) Speed-ups.



(b) Mapping quality information.

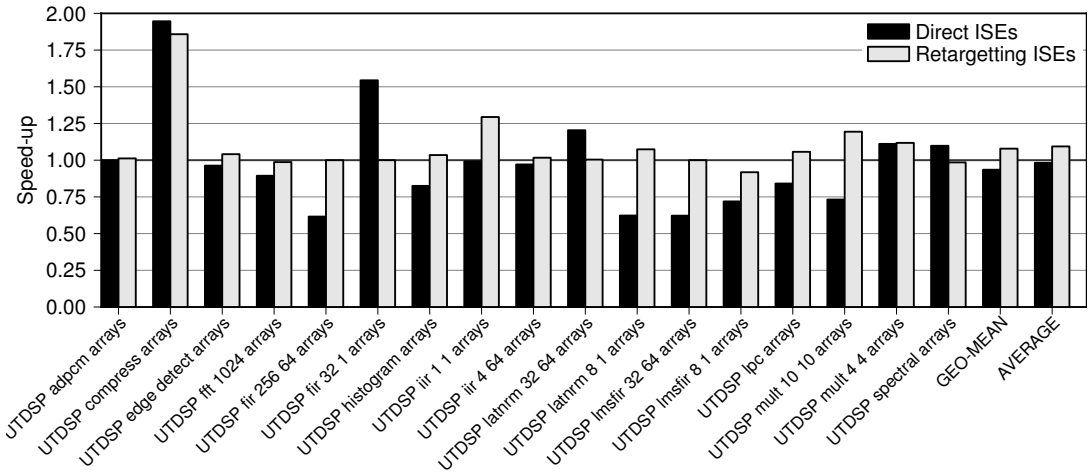
Figure C.4: *Extension instructions are generated for one benchmark (see table 4.3 on page 64) and then exploited on one or more related benchmarks. Both data-sets are for 4 input + 4 output extension instructions.*



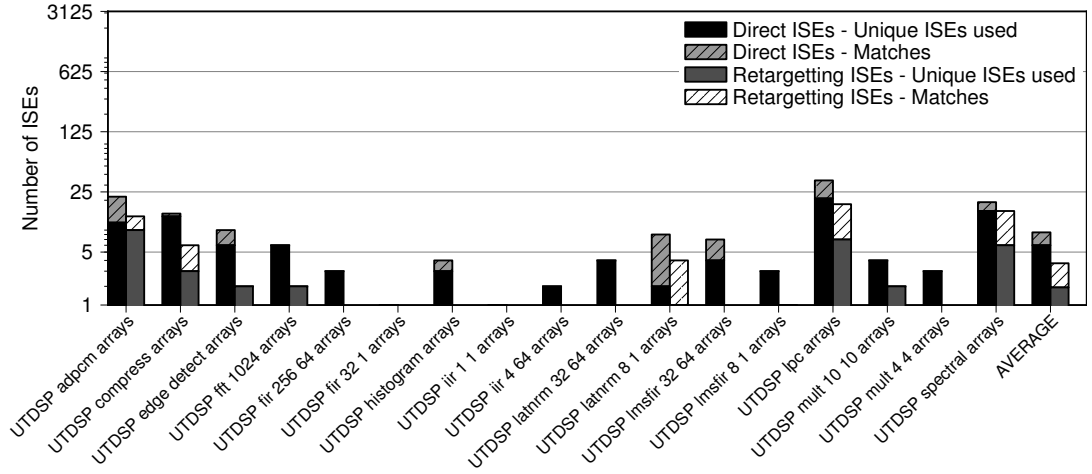
(a) Speed-ups.

(b) Mapping quality information.

Figure C.5: *Extension instructions are generated for each benchmark and then combined to create a large set of instructions,* MapISE *then maps instructions from the entire set to each benchmark. Both data-sets are for 4 input + 4 output extension instructions.*



(a) Speed-ups.

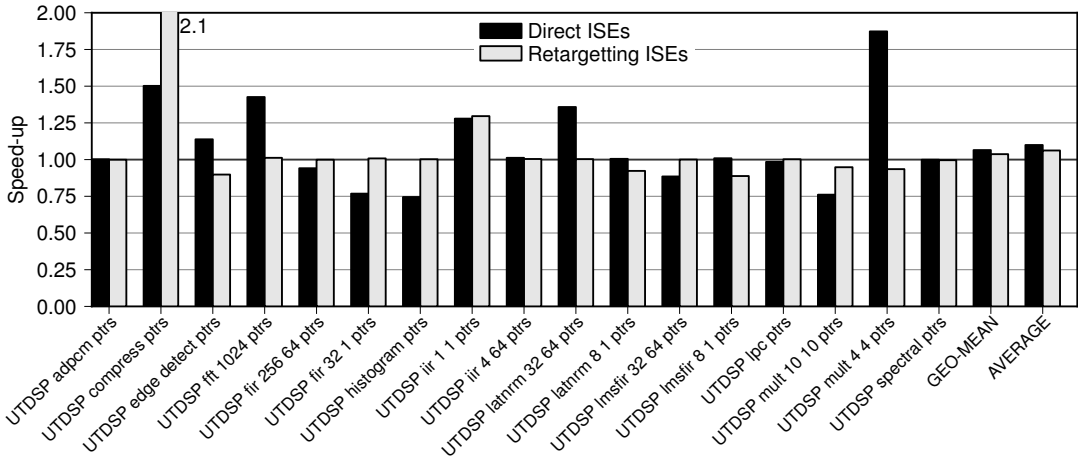


(b) Mapping quality information.

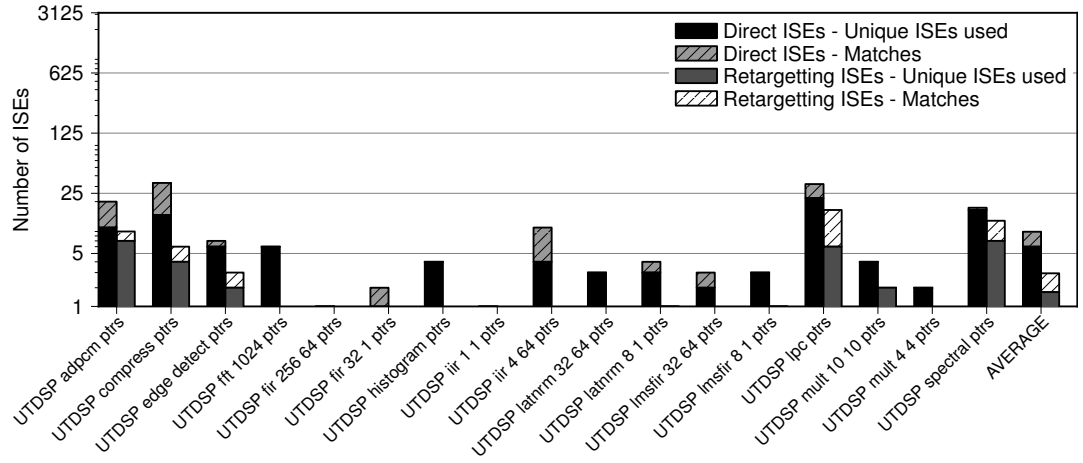
Figure C.6: *Extension instructions are generated for* ptrs *benchmarks and then exploited on* arrays *benchmarks. Both data-sets are for extension instructions containing constants.*

C.2 Hard-Wiring Constant Values

Evaluates the technique presented in section 5.1.2 in the context of retargeting extension instructions.

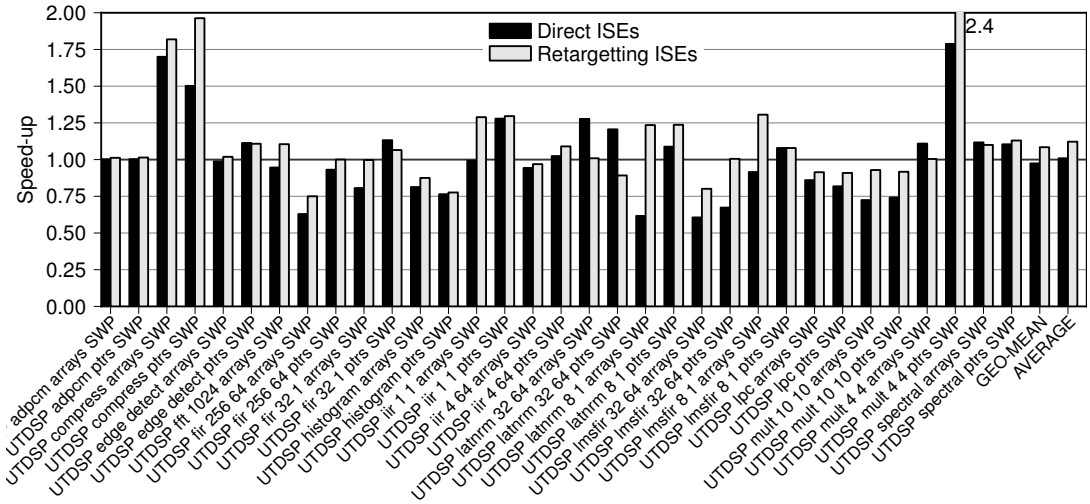


(a) Speed-ups.

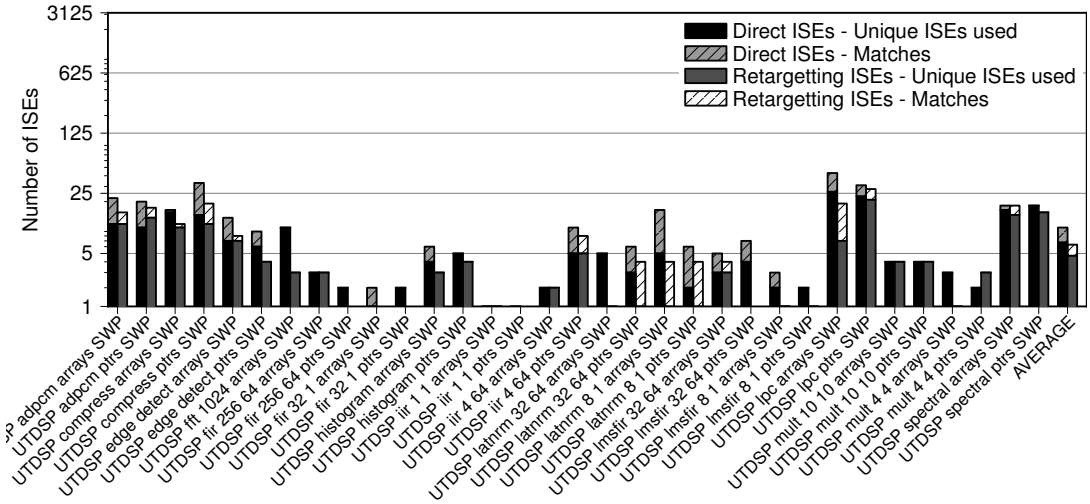


(b) Mapping quality information.

Figure C.7: *Extension instructions are generated for* arrays *benchmarks and then exploited on* ptrs *benchmarks. Both data-sets are for extension instructions containing constants.*

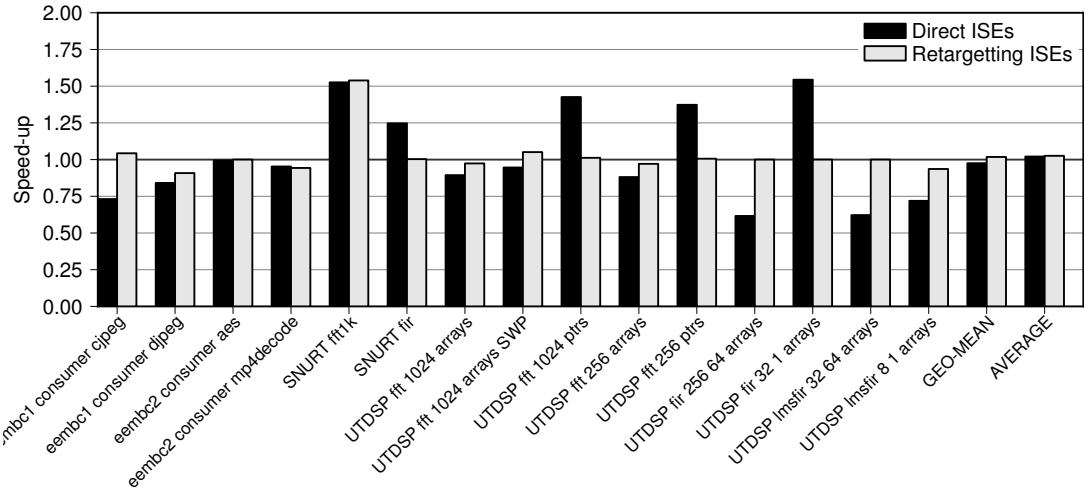


(a) Speed-ups.

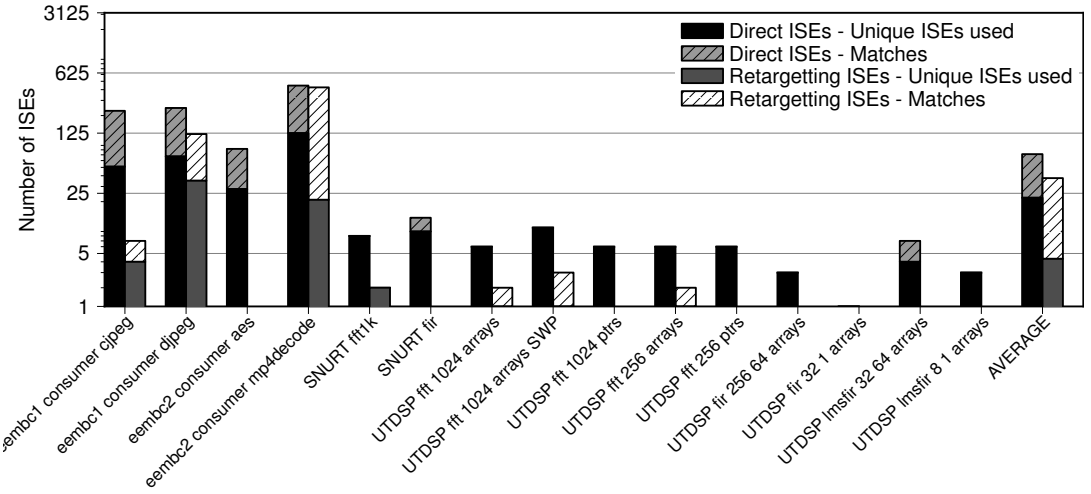


(b) Mapping quality information.

Figure C.8: *Extension instructions are generated for* arrays *benchmarks and then exploited on* arrays-SWP *benchmarks. Both data-sets are for extension instructions containing constants.*

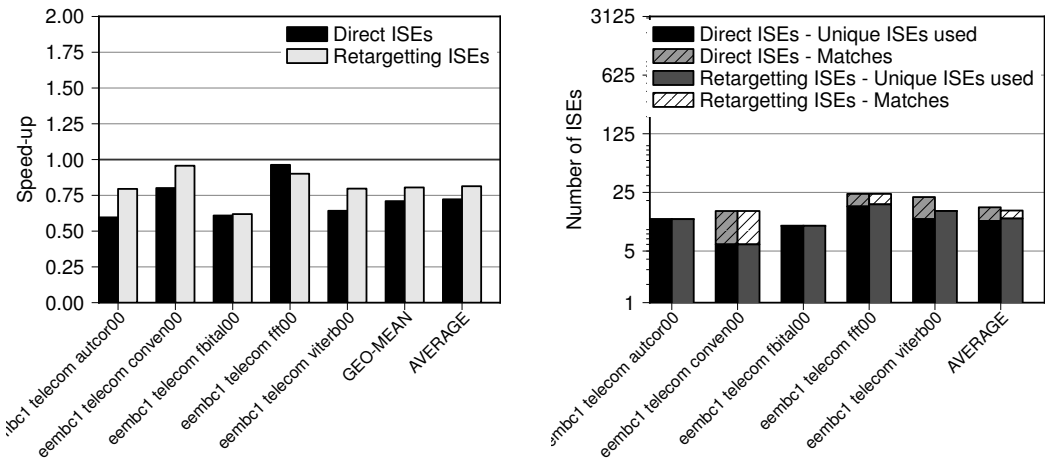


(a) Speed-ups.



(b) Mapping quality information.

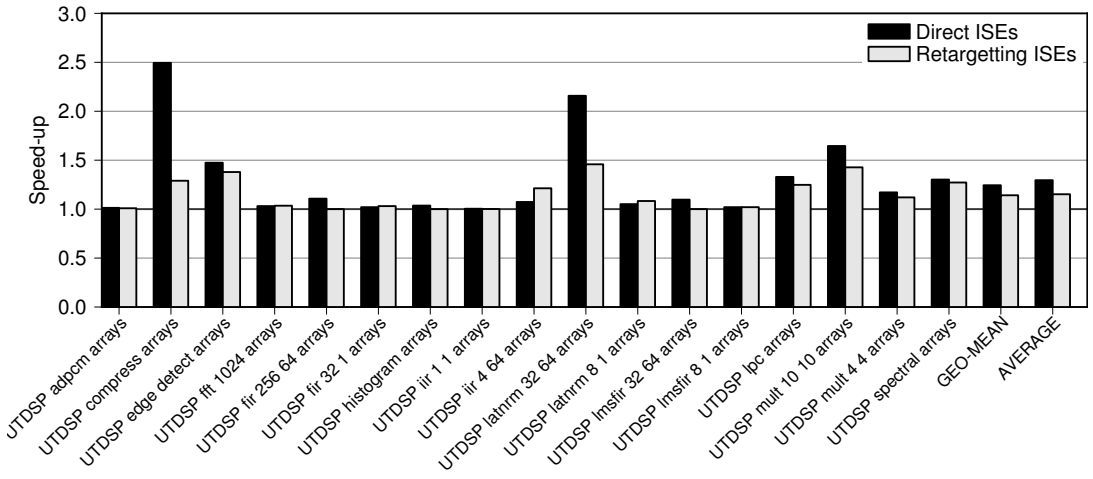
Figure C.9: *Extension instructions are generated for one benchmark (see table 4.3 on page 64) and then exploited on one or more related benchmarks. Both data-sets are for extension instructions containing constants.*



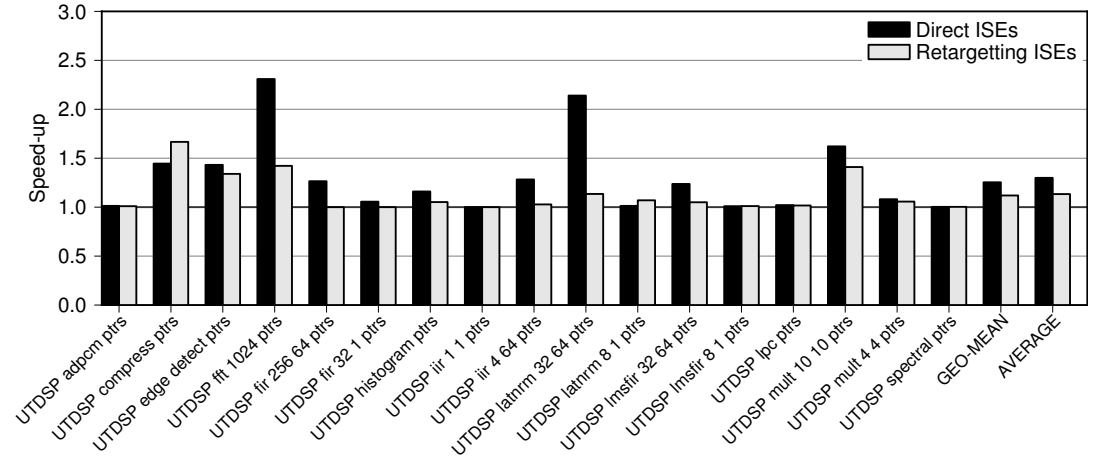
(a) Speed-ups.

(b) Mapping quality information.

Figure C.10: *Extension instructions are generated for each benchmark and then combined to create a large set of instructions,* MapISE *then maps instructions from the entire set to each benchmark. Both data-sets are for extension instructions containing constants.*



(a) Extension instructions are generated for *ptrs* benchmarks and then exploited on *arrays* benchmarks.

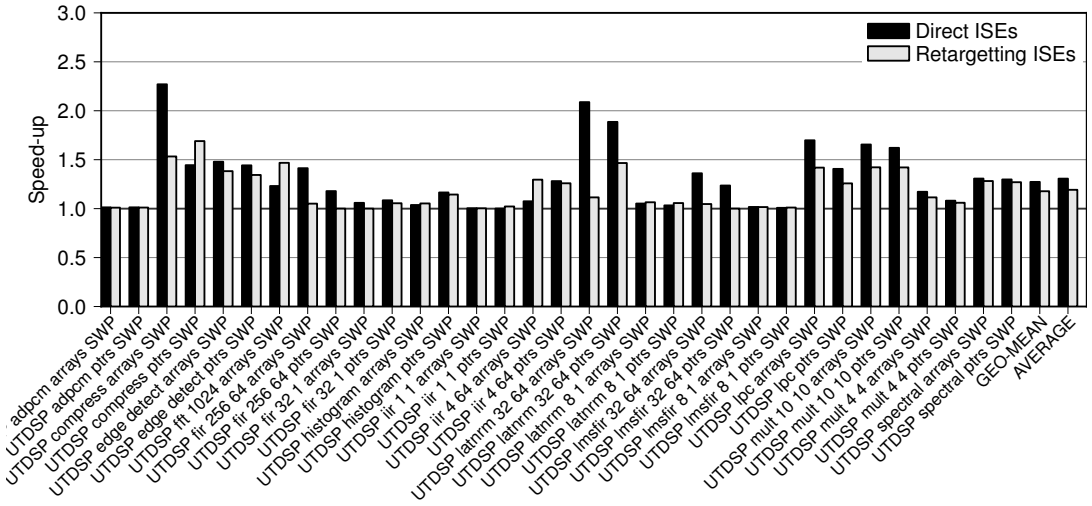


(b) Extension instructions are generated for *arrays* benchmarks and then exploited on *ptrs* benchmarks.

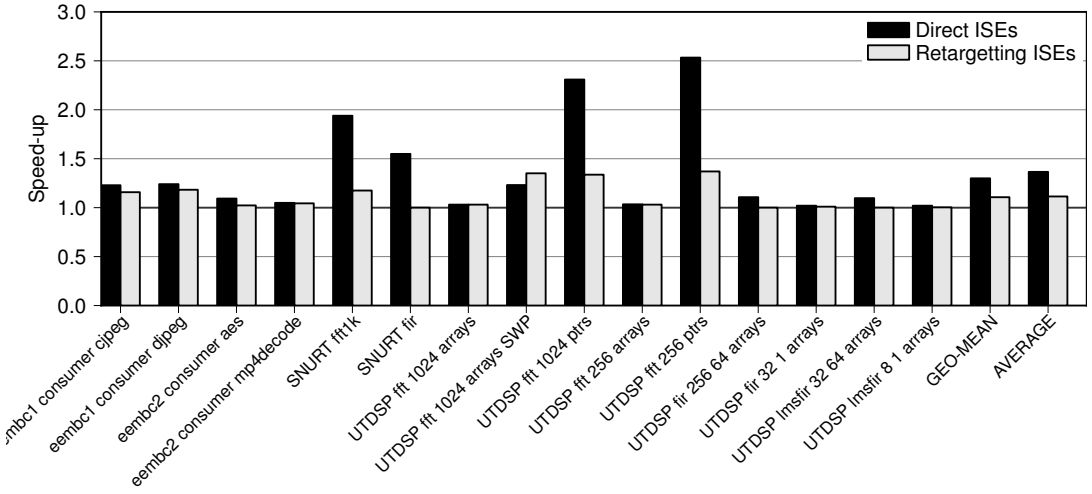
Figure C.11: *Retargeting wide extension instructions with 4 inputs + 4 outputs.*

C.3 Wide Instructions

Evaluates the technique presented in section 5.3 in the context of retargeting extension instructions.



(a) Extension instructions are generated for *arrays* benchmarks and then exploited on *arrays-SWP* benchmarks.



(b) Extension instructions are generated for one benchmark (see table 4.3 on page 64) and then exploited on one or more related benchmarks.

Figure C.12: *Retargeting wide extension instructions with 4 inputs + 4 outputs.*

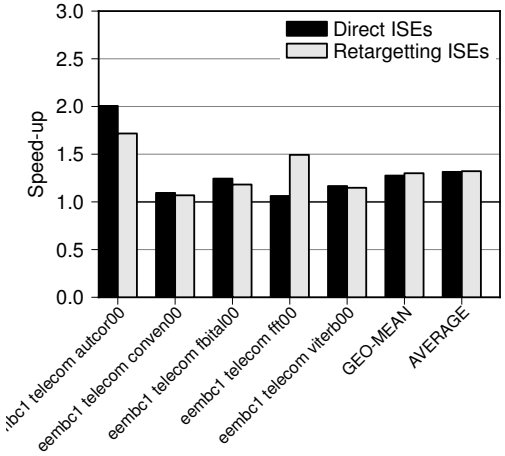


Figure C.13: *Extension instructions are generated for each benchmark and then combined to create a large set of instructions,* MapISE *then maps instructions from the entire set to each benchmark. Both data-sets are for wide extension instructions with 4 inputs + 4 outputs.*

Bibliography

ECJ package. http://cs.gmu.edu/~eclab/projects/ecj/, 2008.

lp_solve package. http://lpsolve.sourceforge.net/5.5/, 2010.

ACE Associated Compiler Experts. DSP-C, an extension to ISO/IEC IS 9899:1990. Technical report, ACE Associated Compiler Experts bv, 1998.

ACE Associated Compiler Experts. ACE CoSy website. http://www.ace.nl/compiler/cosy.html, 2011.

Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Michael F.P. O'Boyle, John Thomson, Marc Toussaint, and Christopher K.I. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the 4th Annual International Symposium on Code Generation and Optimization (CGO '04)*, pages 295–305, March 2006.

Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.

Oscar Almer, Richard Bennett, Igor Böhm, Alastair Murray, Xinhao Qu, Marcela Zuluaga, Björn Franke, and Nigel Topham. An end-to-end design flow for automated instruction set extension and complex instruction selection based on GCC. In *Proceedings of the First International Workshop on GCC Research Opportunities (GROW '09)*, pages 49–60, January 2009.

ARC International. ARC FPX white paper, 2007. URL http://www.arc.com/configurablecores/fpx.

ARC International. ARC XY advanced DSP product brief, 2010.

ARM. ARM website. http://www.arm.com/, 2011.

ARM Ltd. *Cortex™-A8 Technical Reference Manual, Revision: r3p2*. 2010a.

ARM Ltd. *Cortex™-A9 Technical Reference Manual, Revision: r2p2*. 2010b.

Marnix Arnold and Henk Corporaal. Designing domain-specific processors. In *Proceedings of the 9th International Symposium on Hardware/Software Codesign (CODES '01)*, pages 61–66, April 2001.

Kubilay Atasu, Günhan Dündar, and Can Özturan. An integer-linear programming approach for identifying instruction-set extensions. In *Proceedings of the Third IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '05)*, pages 172–177, September 2005a.

Kubilay Atasu, Günhan Dündar, and Can Özturan. An integer linear programming approach for identifying instruction-set extensions. In *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 172–177, September 2005b.

F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: an integrated electronic system design environment. *Computer*, 36(4):45–52, April 2003.

Marcel Beemster, Hans van Someren, Willem Wakker, and Walter Banks. The Embedded C extension to C. `http://www.ddj.com/cpp/184401988`, 2005.

Richard V. Bennett, Alastair C. Murray, Björn Franke, and Nigel Topham. Combining source-to-source transformations and processor instruction set extensions for the automated design-space exploration of embedded systems. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '07)*, pages 83–92, June 2007.

Michel Berkelaar. Mixed integer programming (MIP) solver. `http://groups.yahoo.com/group/lp_solve/`, 2008.

Partha Biswas, Sundarshan Banerjee, Nikil D. Dutt, Laura Pozzi, and Paolo Ienne. ISEGEN: An iterative improvement-based ISE generation technique for fast customization of processors. *IEEE Transactions on VLSI*, 14(7):754–762, July 2006a.

Partha Biswas, Nikil Dutt, Paolo Ienne, and Laura Pozzi. Automatic identification of application-specific functional units with architecturally visible storage. In *Proceedings of Design Automation and Test in Europe DATE '06)*, pages 212–217, March 2006b.

Paolo Bonzini and Laura Pozzi. Code transformation strategies for extensible embedded processors. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '06)*, pages 242–252, October 2006.

Alexandre Borghi, Valentin David, and Akim Demaille. C-Transformers - A framework to write C program transformations. *ACM Crossroads*, 12(3):3–3, 2006.

F Brandner, D Ebner, and A Krall. Compiler generation from structural architecture descriptions. *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 13–22, 2007.

D.L. Brown, William D. Henshaw, and Daniel J. Quinlan. Overture: An object-oriented framework for solving partial differential equations on overlapping grids. In *Proceedings of the SIAM Conference on Object Oriented Methods for Scientfic Computing*, 1999.

John Cavazos and Michael F. P. O'Boyle. Automatic tuning of inlining heuristics. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 14, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 1-59593-061-2. doi: http://dx.doi.org/10.1109/SC.2005.14.

Jianjiang Ceng, Manuel Hohenauer, Rainer Leuper, Gerd Ascheid, Heinrich Meyr, and Gunnar Braun. C compiler retargeting based on instruction semantics models. *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 1–6, Dec 2005.

Kingsum Chow and Youfeng Wu. Feedback-directed selection and characterization of compiler optimizations. In *Proceedings of the 2nd Workshop on Feedback Directed Optimization*, November 1999.

E. Chung, L. Benini, and G. De Micheli. Energy efficient source code transformation based on value profiling. In *Proceedings of the International Workshop on Compilers and Operating Systems for Low Power*, Philadelphia, USA, October 2000.

Keith D. Cooper and Todd Waterman. Investigating adaptive compilation using the MIPSpro compiler. In *In Proc. of the Symp. of the Los Alamos Computer Science Institute*, 2003.

L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. Graph matching: A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.

CoWare. Processor designer datasheet. `http://www.coware.com/PDF/products/LISATek.pdf`, 2007.

Nicheal Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In *Proceedings of the International Conference on Genetic Algorithms and their Applications (ICGA85)*, pages 183–187, 1985.

Reinhard Diestel. *Graph Theory (Fourth Edition)*. 2010.

Dietmar Ebner, Florian Brandner, Bernhard Scholz, Andreas Krall, Peter Wiedermann, and Albrecht Kadlec. Generalized instruction selection using SSA-graphs. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '08)*, pages 31–40, March 2008.

Heiko Falk and Peter Marwedel. *Source Code Optimization Techniques for Data Flow Dominated Embedded Software*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2004.

Heiko Falk and Manish Verma. Combined data partitioning and loop nest splitting for energy consumption minimization. In *Proceedings of the 8th International Workshop on Software and Compilers for Embedded Systems (SCOPES '04)*, pages 137–151, September 2004.

Joseph A. Fisher, Paolo Faraboschi, and Cliff Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers, and Tools*. Elsevier Inc., 2005.

Stephen Fitzpatrick and Lambert Meertens. An experimental assessment of a stochastic, anytime, decentralized, soft colourer for sparse graphs. In *Proceedings of the International Symposium on Stochastic Algorithms (SAGA '01)*, pages 49–64, December 2001.

H. Francis. ARM DSP-enhanced extensions, 2001. URL `http://www.arm.com/pdfs/ARM-DSP.pdf`.

Björn Franke and Michael O'Boyle. Array recovery and high-level transformations for DSP applications. *ACM Transactions on Embedded Computing Systems (ACM TECS)*, 2(2):132–162, May 2003a.

Björn Franke and Michael O'Boyle. Combining program recovery, auto-parallelisation and locality analysis for C programs on multi-processor embedded systems. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT '03)*, pages 104–113, September/October 2003b.

Björn Franke, Michael O'Boyle, John Thomson, and Grigori Fursin. Probabilistic source-level optimisation of embedded programs. In *Proceedings of the Conference on Languages, Compilers and Tools for Embedded Systems*, pages 78–86, June 2005.

Christopher W. Fraser and David R. Hanson. A retargetable compiler for ANSI C. *ACM SIGPLAN Notices*, 26(10):29–43, October 1991.

Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code generator generator. 1(3):213–226, 1992.

Stefan Fröhlich and Bernhard Wess. Integrated approach to optimized code generation for heterogeneous-register architectures with multiple data-memory banks. In *Proceedings of the 14th Annual IEEE International ASIC/SOC Conference*, pages 122–126, September 2001.

G. Fursin, M. O'Boyle, and P. Knijnenburg. Evaluating iterative compilation, 2002.

Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Phil Barnard, Elton Ashton, Eric Courtois, Francois Bodin, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, and Michael O'Boyle. Milepost gcc: machine learning based research compiler. In *Proceedings of the GCC Developers' Summit*, June 2008.

Carlo Galuzzi and Koen Bertels. The instruction-set extension problem: A survey. *ARC 2008, LNCS 4943*, page 12, Mar 2008.

Carlo Galuzzi and Koen Bertels. The instruction-set extension problem: A survey. *To appear in ACM Transactions on Reconfigurable Technology and Systems (ACM TRETS)*, 4(2), May 2011.

T. Glökler, A. Hoffmann, and H. Meyr. Methodical low-power ASIP design space exploration. *VLSI Signal Processing*, 33(3):229–246, March 2003.

G. Gréwal, S. Coros, D. Banerji, and A. Morton. Comparing a genetic algorithm penalty function and repair heuristic in the DSP application domain. In *Proceedings of the 24th IASTED International Conference on Artificial Intelligence and Applications (AIA '06)*, pages 31–39, February 2006a.

G. Gréwal, S. Coros, A. Morton, and D. Banerji. A multi-objective integer linear program for memory assignment in the DSP domain. In *Proceedings of the IEEE Workshop on Memory Performance Issues (WMPI '06)*, pages 21–28, February 2006b.

Gary Gréwal, Tom Wilson, and Andrew Morton. An EGA approach to the compile-time assignment of data to multiple memories in digital-signal processors. *SIGARCH Computer Architecture News*, 31(1):49–59, March 2003.

Rajiv Gupta and Rastislav Bodik. Register pressure sensitive redundancy elimination. *Lecture Notes in Computer Science*, 1575:107–122, 2004.

J. D. Hiser and J. W. Davidson. EMBARC: An efficient memory bank assignment algorithm for retargetable compilers. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '04)*, pages 182–191, June 2004.

M. Hohenauer, H. Scharwaechter, K. Karuri, O. Wahlen, T. Kogel, R. Leupers, G. Ascheid, and H. Meyr. Compiler-in-loop architecture exploration for efficient application specific embedded processor design. Feb 2004.

Paolo Ienne and Rainer Leupers. *Customizable Embedded Processors*. Elsevier Inc., 2007.

Intel. Intel PXA270 processor for embedded computing, 2007. URL `http://www.intel.com`.

Diviya Jain, Anshul Kumar, Laura Pozzi, and Paolo Ienne. Automatically customising VLIW architectures with coarse grained application-specific functional units. In *Proceedings of the 8th International Workshop on Software and Compilers for Embedded Systems (SCOPES '08)*, pages 17–32, September 2004.

Xiaoyi Jiang and Horst Bunke. Marked subgraph isomorphism of ordered graphs. In *Advances in Pattern Recognition*, volume 1451 of *Lecture Notes in Computer Science*, pages 122–131. Springer Berlin / Heidelberg, 1998.

Rahul Joshi, Uday Khedker, Vinay Kakade, and Medha Trivedi. Some interesting results about applications of graphs in compilers. *CSI Journal*, 31(4), 2002.

JTC1, SC22, and WG14. Programming languages - C - extensions to support embedded processors. Technical report, ISO/IEC, 2004.

K. Keutzer, S. Malik, A.R. Newton, J.M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 19:1523–1543, 2000.

Kurt Keutzer, Sharad Malic, and A. Richard Newton. From ASIC to ASIP: The next design discontinuity. In *Proceedings of the IEEE International Conference on Computer: VLSI in Computers and Processors (ICCD '02)*, pages 84–90, September 2002.

M-Y. Ko and S. S. Bhattacharyya. Data partioning for DSP software synthesis. In *Proceedings of the International Workshop on Software and Compilers for Embedded Systems (SCOPES '03)*, pages 344–358, September 2003.

Shinsuke Kobayashi, Yoshinori Takeuchi, Akira Kitajima, and Masaharu Imai. Compiler generation in PEAS-III: An ASIP development system. In *Proceedings of the Workshop on Software and Compilers for Embedded Processors (SCOPES '01)*, March 2001.

John Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, 1992.

C. Kulkarni, F. Catthoor, and H. De Man. Code transformations for low power caching in embedded multimedia processors. In *12th International Parallel Processing Symposium*, pages 292–297, 1998.

Martin Labrecque, Peter Yiannacouras, and J. Gregory Steffan. Custom code generation for soft processors. 2006.

Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '04)*, pages 75–88, March 2004.

Hugh Leather, Edwin Bonilla, and Michael O'Boyle. Automatic feature generation for machine learning based optimizing compilation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '09)*, March 2009.

C. G. Lee. UTDSP benchmark suite. http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html, 1998.

R. Leupers and D. Kotte. Variable partioning for dual memory bank DSPs. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '01)*, volume 2, pages 1121–1124, May 2001.

Rainer Leupers and Steven Bashford. Graph-based code selection techniques for embedded processors. *ACM Transactions on Design Automation of Electronic Systems*, 5(4):794–814, October 2000.

Rainer Leupers and Peter Marwedel. Instruction selection for embedded DSPs with complex instructions. In *Proceedings of the Conference on European Design Automation (EURO-DAC '96)*, pages 200–205, 1996.

A. Linden and J. Fenn. Understanding gartner's hype cycles. *Gartner Research Report*.

Sean Luke and Liviu Panait. Lexicographic parsimony pressure. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 829–836, July 2002.

Victor De La Luz and Mahmut Kandemir. Array regrouping and its use in compiling data-intensive embedded applications. *IEEE Transactions on Computers*, 53(1):1–19, January 2004.

Brendan D. McKay. Nauty user's guide. http://cs.anu.edu.au/~bdm/nauty/, 2008.

MIPS Technologies. MIPS32(R) architecture for programmers, 2007. URL http://www.mips.com/.

Alastair Murray and Björn Franke. Fast source-level data assignment to dual memory banks. In *Proceedings of the 11th International Workshop on Software and Compilers for Embedded Systems (SCOPES '08)*, pages 43–52, March 2008.

Alastair Murray and Björn Franke. Using genetic programming for source-level data assignment to dual memory banks. In *Proceedings of the 3rd Workshop on Statistical and Machine Learning Approaches to Architectures and Compilation (SMART '09)*, pages 75–89, January 2009.

Alastair Murray and Björn Franke. Adaptive source-level data assignment to dual memory banks. *ACM Transactions on Embedded Computing Systems (ACM TECS)*, 11S(1), June 2012a.

Alastair Murray and Björn Franke. Compiling for automatically generated instruction set extensions. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '12)*, April 2012b.

Alastair C. Murray, Richard V. Bennett, Björn Franke, and Nigel Topham. Code transformation and instruction set extension. *ACM Transactions on Embedded Computing Systems (ACM TECS)*, 8(4):1–31, 2009.

The Complexity of Theorem-Proving Procedures. Stephen a. cook. In *Proceedings of the third annual ACM symposium on Theory of Computing (STOC '71)*, pages 151–158, 1971.

Preeti R. Panda, Nikil D. Dutt, and Alexandru Nicolau. On-chip vs off-chip memory: The data partioning problem in embedded processor-based systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 5(3):682–704, July 2000.

Armita Peymandoust, Laura Pozzi, Paolo Ienne, and Giovanni De Micheli. Automatic instruction set extension and utilization for embedded processors. In *Proceedings of the 14th International Conference on Application-Specific Systems, Architectures and Processors*, pages 108–118, June 2003.

Laura Pozzi and Paolo Ienne. Exploiting pipelining to relax register-file port constraints of instruction-set extensions. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 2–10, 2005.

Laura Pozzi, Kubilay Atasu, and Paolo Ienne. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(7):1209–1229, July 2006.

Todd A. Proebsting. Least-cost instruction selection in DAGs is NP-complete. `https://research.microsoft.com/en-us/um/people/toddpro/papers/proof.htm`, 1998.

Jerzy Rozenblit and Klaus Buchenrieder. *Codesign - Computer-Aided Software/Hardware Engineering*. IEEE Press, New York, 1995.

Radu Rugina and Martin Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)*, pages 77–90, May 1999.

Mazen A. R. Saghir, Paul Chow, and Corinna G. Lee. Exploiting dual data-memory banks in digital signal processors. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 234–243, September 1996.

Hanno Scharwaechter, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, Jonghee M. Youn, and Yunheung Paek. A code-generator generator for multi-output instructions. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '07)*, pages 131–136, October 2007.

Markus Schordan and Daniel J. Quinlan. A source-to-source architecture for user-defined optimizations. In *Proceedings of the Joint Modular Languages Conference*, pages 214–223, August 2003.

Seoul National University - Real-Time Research Group. SNU real-time benchmarks. `http://archi.snu.ac.kr/realtime/benchmark/`, 2008.

Viera Sipkovà. Efficient variable allocation to dual memory banks of DSPs. In *Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems (SCOPES '03)*, pages 359–372, September 2003.

Stanford Compiler Group. The SUIF Library: A set of core routines for manipulating SUIF data structures. `http://suif.stanford.edu/suif/suif1/docs/suif_toc.html`, 1996.

M. Stephenson, M. Martin, U.-M. O'Reilly, and S. Amarasinghe. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '03)*, pages 77–90, June 2003a.

Mark Stephenson, Una-May O'Reilly, Martin C. Martin, and Saman Amarasinghe. Genetic programming applied to compiler heuristic optimization. In *Proceedings of the 6th European Conference on Genetic Programming*, April 2003b.

Stretch Inc. SCP architecture reference. http://www.stretchinc.com/, 2007.

Tensilica Inc. The XPRES compiler: Triple-threat solution to code performance challenges, 2005.

Steve W. K. Tjiang. An olive twig. Technical report, Synopsys Inc., 1993.

Various. FFmpeg. http://ffmpeg.org, 2011a.

Various. x264. http://www.videolan.org/developers/x264.html, 2011b.

Ajay K. Verma and Paolo Ienne. Improved use of the carry-save representation for the synthesis of complex arithmetic circuits. In *Proceedings of the International Conference on Computer Aided Design*, pages 791–798, 2004.

Ajay K. Verma and Paolo Ienne. Towards the automatic exploration of arithmetic circuit architectures. In *Proceedings of the 43rd Design Automation Conference*, pages 445–450, 2006.

Manish Verma and Peter Marwedel. *Advanced Memory Optimisation Techniques for Low-Power Embedded Processors*. Springer, 2007.

Yijian Wang and David Kaeli. Source level transformations to improve I/O data partitioning. In *Proceedings of the International Workshop on Storage Network Architecture and Parallel I/Os*, pages 27–35, 2003.

Zheng Wang and Michael F.P. O'Boyle. Partitioning streaming parallelism for multi-cores: A machine learning based approach. In *PACT '10: Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 307–318, September 2010.

Douglas B. West. *Introduction to Graph Theory (Second Edition)*. 2000.

Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12):31–37, December 1994.

B.D. Winters and A.J. Hu. Source-level transformations for improved formal verification. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers & Processors*, pages 599–602, September 2000.

Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. Application-specific customization of soft processor microarchitecture. pages 201–210, 2006.

V. Zivojnović, J. M. Velarde, C. Schläger, and H. Meyr. DSPstone: A DSP-orientated benchmarking methodology. In *Proceedings of the 6th International Conference on Signal Processing Applications and Technology (ICSPAT '94)*, October 1994.

Index