Code Optimizations for Narrow Bitwidth Architectures

Indu Bhagat

Departament d'Architectura de Computadors

Universitat Politècnica de Catalunya

Advisors :

Enric Gibert Codina, Intel Barcelona Research Center

Jesús Sánchez Navarro, Intel Barcelona Research Center

Antonio González Colás, Intel Barcelona Research Center & UPC

A thesis submitted for the degree of

Doctor of Philosophy / Doctor per la UPC

2011 December

Code Optimizations for Narrow Bitwidth Architectures

Indu Bhagat

Departament d'Architectura de Computadors

Universitat Politècnica de Catalunya

Advisors :

Enric Gibert Codina

Intel Barcelona Research Center

Intel Labs, Universitat Politècnica de Catalunya

Jesús Sánchez Navarro

Intel Barcelona Research Center & UPC

Intel Labs, Universitat Politècnica de Catalunya

Antonio González Colás

Intel Barcelona Research Center & UPC

Intel Labs, Universitat Politècnica de Catalunya

A thesis submitted for the degree of

Doctor of Philosophy / Doctor per la UPC

2011 December

Abstract

This thesis derives its motivation from an inherent computational inefficiency of traditional computer systems. This inefficiency is primarily the result of the fact that many of our contemporary applications (integer, network and multimedia processing) do not need wide datapath in the hardware. However, the hardware, oblivious to this, inadvertently utilizes its complete available datapath, resulting in the problem of computational inefficiency – usage of more resources like power and computational logic than necessary. In the past, many researchers have proposed techniques to exploit traditional narrow computations (i.e., those computations that do not require the complete available bitwidth of the processor's datapath) towards achieving multiple benefits ranging from achieving power reductions, reducing hardware-complexity, and also to gain performance.

This thesis takes a hardware-software collaborative approach to tackle the problem of computational inefficiency in a holistic manner. The hardware is designed for energy-efficiency by restraining the execution core to a strictly 16-bit datapath (integer datapath only). This redesign is referred to as the Narrow Bitwidth Architecture and its interface to the outside (software) world is termed as the Narrow ISA. The gain here is that the hardware functions as an extremely simple, low-cost, low-complexity execution core.

Narrow Bitwidth Architecture is unique in that although the datapath is squeezed to 16-bits, it continues to support 32-bit / 64-bit computations (and hence, the current software stack). This is made possible by a non-conventional memory interface design which combines a 16-bit data interface with a 64-bit address interface. Further, it is the software which is responsible for efficiently mapping the current stack of 64-bit applications onto the 16-bit hardware. However, this approach introduces the risk of losing out on performance because even when a reasonably smart code-translator maps the 64-bit applications on to the 16-bit processor, it has been observed that there is non-negligible penalty both in terms of the dynamic code size (about 3.9x) and execution time in cycles (2.2x).

Thus kicks in the importance of the role of the compiler and harnessing the power of compiler optimizations. The compiler needs to not only translate but also aggressively optimize to reduce the negative impacts of the narrow ISA. Further, the narrow ISA presents itself as a completely new play-doh for compiler optimizations – the narrow stream is inherently more parallel; it has more computations of finer granularity. A hardware-based aggressive code optimizer will be limited in scope and above all, hard to implement. It also defies the purpose of achieving a simple execution core.

Hence, the goal of this thesis is to design a software layer that can assuage this performance penalty while mapping 64-bit programs on to the 16-bit datapath based hardware. More specifically, this thesis focuses on compiler optimizations targeting the problem of how to compile a 64-bit program to a 16-bit machine from the perspective of *Minimum Required Computations*. Given a program, the notion of Minimum Required Computations (henceforth, MRC) aims to infer how much computation is really required to generate the same (correct) output as the original program.

Approaching perfect MRC is an intrinsically ambitious goal in that it requires oracle predictions of program behavior. Towards this end, the thesis proposes and evaluates three heuristic-based optimizations to closely infer the MRC. The optimizations are performed by the software in the form of a compiler / dynamic optimizer. The perspective of minimum required computations unfolds into a definition of *Productiveness* – if a computation does not alter the storage location, it is non-productive and hence, not necessary to be performed. In this research, the concept of MRC has been applied to different granularities of the data-flow as well as control-flow of the programs.

Three profile-based code optimization techniques have been proposed :

1. *Global Productiveness Propagation* (GPP) applies the concept of productiveness at the granularity of a function.
2. *Local Productiveness Pruning* (LPP) applies the same concept but at a much finer granularity of a single narrow computation.
3. *Minimal Branch Computation* (MBC) is a profile-based, code reordering optimization technique which applies the principles of MRC for conditional branches.

The primary aim of all these techniques is to reduce the dynamic code footprint of the narrow ISA programs. The first two optimizations, namely the Global Productiveness Propagation (GPP) and the Local Productiveness Pruning (LPP), perform the task of *code pruning*. Both these techniques make use of profiles to speculatively prune the non-productive (useless) computations. Further, these two optimization techniques perform backward traversal of the optimization regions to embed checks into the non-speculative slices, hence, making them self-sufficient to detect a mis-speculation dynamically.

The Minimal Branch Computation (MBC) optimization is a use case of a broader concept of *reordering narrow backslices*. The idea behind MBC is to reorder the backslices containing narrow computations such that the minimal necessary computations to generate the same (correct) output are performed in the *most-frequent case*; the rest of the computations are performed only when necessary.

With the proposed optimizations, it can be concluded that there do exist ways to smartly compile a 64-bit application to a 16-bit ISA such that the overheads are considerably reduced. A combination of the foregoing optimizations together with a classical code scheduling algorithm has the potential to reduce the dynamic code size penalty from 3.9x to 2.68x and the performance penalty from 2.2x to 1.38x.

To my parents and my big joyful family, with love.

Acknowledgements

My first and most sincere acknowledgments go to my advisors – Enric Gibert, Jesús Sánchez, and Antonio González, for giving me an opportunity to research with the ARCO (Architecture and Compilers) Group at the Department of Computer Architecture at the UPC. They have been great mentors throughout and have greatly helped in shaping my professional and academic skills. The lessons I learnt from them have brought about a great change in my outlook and will continue to do so throughout my career. This thesis wouldn't have been what it is without their insights, feedback and guidance. Heartfelt thanks to Enric and Jesús with whom I truly enjoyed working during all these years. I cannot imagine having worked on a thesis without you two.

I also wish to thank my big joyful family. They have always been full of encouragement and have always extended their support at all times unconditionally. Acknowledging their contribution to my thesis is the least I can do. My most warm acknowledgments to my doting parents who have strengthened this thesis and my life with their love, and encouragement to always strive for the best. I have learnt much of my perseverance which was instrumental to the completion of this thesis from them. The debts of gratitude are too numerous to particularize. My most adorable nephew and nieces with their shining bright eyes and toothy smiles (with an added or deleted tooth every now and then) have always kept me going.

I am also greatly thankful to my husband Anshuman for having reminded me untiringly to be objective about both the technical and non-technical aspects of my thesis. He has always been a reminder to not lose focus, even in the state of crisis. Thanks to Anshuman also for painstakingly reviewing several drafts of my submissions and the thesis document multiple times, for cooking several delicious meals for me while I slept/worked and for being there always.

The stay at (DAC) UPC, Barcelona shall always remain the most memorable one of my life. My colleagues at D6-113, with whom I shared my office as a PhD student, all my friends, and my colleagues at UPC with whom I chatted over several meals and coffees have also greatly added value to my thesis. Some of the many joys of doing a PhD – sharing the classic PhD in-jokes, sharing woes of graduate life and above all the 'collaborative procrastination sessions'; all of those would have been so incomplete without you all.

Contents

List of Figures

LIST OF FIGURES

List of Tables

1

Introduction

1.1 Motivation

Processor architectures have evolved from 4-bit wide datapath to 64-bit datapath. Architectures with 64-bit wide datapath can be seen not only in the general-purpose computing domain (Intel's Dual Core, AMD Turion 64) but also in the embedded domain (Toshiba TX4925, NEC UPD301x, Intel Atom, AMD Geode and AMD Alchemy to name a few). This increase in the bitwidth of the datapath offers many advantages. Firstly, wider datapath implies a larger memory-addressing capability which is used by many of the state-of-the-art applications. Secondly, wider datapath increases the computational capabilities and enables higher precision math. Further, these advantages benefit the integer as well as floating-point computations.

However, many of our contemporary applications (integer, network and multimedia processing) do not need wide datapath in the hardware [4, 8, 17, 36, 47, 53]. These applications tend to have a large percentage of subword computations in the form of narrow data-types and masking operations. Such type of computations, which do not need the total available datapath of the processor, have traditionally been identified as *narrow computations*. Even for SPEC2000 integer suite of programs, which are the representative CPU-intensive general-purpose applications, it has been accredited by previous research [17] that as high as 40% produced results need only 16-bit datapath. The same has also been corroborated by our experiments (results in Section 2.4.3.1).

The observation that a large percentage of computations do not need the available wide datapath in the hardware has been well-acknowledged. It is one of the reasons why contemporary ISAs like x86 / IA64 continue to support byte/word extensions. Another dimension that is used to exploit narrow computations is that of the SIMD instructions (AMD's 3DNow!, Intel's

MMX, SSE and SSE2) which already exist in many existing CPUs. SIMD instructions exploit data-level parallelism, whereby multiple subword computations are packed together to execute as one large quanta of work.

Note that, an increase in the datapath width entails structural changes in almost all the 'processing' elements – register files, pipeline datapath, ALUs, data bypass logic and interface to memory (data and address buses) amongst others. All this additional processor real estate has implications on the overall hardware complexity in terms of *area, cycle-time and energy*.

Oblivious to the actual bitwidth requirement of a computation, most of the contemporary processor designs inadvertently exercise the *complete* datapath of the processor. Hence, as the datapath width increases, we not only pay in terms of cost borne and power consumed by the additional datapath, but above all, in terms of *under-utilized hardware complexity*.

Computational Inefficiency. This leads us to the issue of *computational inefficiency* – i.e. usage of more resources like power and computational logic than necessary. Some of the previous research has investigated on this issue of computational inefficiency and has exploited the high prevalence of *narrow computations* for reducing hardware-complexity [5, 30, 37, 56], designing for energy-efficiency [4, 8, 9, 47], and obtaining performance gains [4, 33, 36]. Some of these approaches have been elaborated upon in the section on literature review (Section 1.4). But in summary, these approaches vary mainly in the degree of innovations brought about in hardware and/or software to achieve the goals.

The aim of this thesis, however, is to approach this problem of computational inefficiency, which is inherent in our traditional computer systems, in a more holistic manner. Our approach draws inspiration from the synergistic viewpoint of improving the processor architecture via innovations in the hardware, and using the software to support these innovations. Towards this end, in this thesis we propose to redesign the hardware for making the *common case* computationally efficient by reducing the bitwidth of the datapath and the processing elements to 16-bits only. This architecture, as explained in detail later in Section 2.1, is referred to as the *Narrow Bitwidth Architecture*. We believe that narrow bitwidth architectures offer a promising solution for future low-power, low-cost design requirements; both of which are important design goals not only in embedded but also in general-purpose computing domains. Narrow bitwidth architecture is unique in that although the datapath is squeezed to 16-bits, it continues to support 32-bit / 64-bit computations (and hence, the current software stack). This is made possible by a non-conventional memory interface design which combines a 16-bit data interface with a 64-bit address interface.

The task of mapping the current software stack on to this narrow bitwidth architecture is laid down on the software in the form of dynamic optimizer / compiler. The central issue in

mapping applications on to the narrow bitwidth architecture is the risk of losing out on performance. This is because even when a reasonably smart code-translator maps 64-bit applications to 16-bit processor, it has been observed that there is non-negligible penalty (more details in Section 2.2.3) both in terms of the dynamic code size (about 3.9x) and execution time in cycles (2.2x).

1.2 The Thesis

The focus of this thesis is to develop a hardware-software ecosystem which attacks this problem of computational inefficiency due to the bitwidth requirements of computations in a holistic manner. This thesis lays down the proposition that the hardware can be redesigned to be a strictly 16-bit datapath execution core (integer datapath only). This redesign transforms the hardware to an extremely simple, low-cost, low-complexity execution core. The key to establish the viability of this redesigned hardware (the narrow bitwidth architecture) is to design an effective software-based translation system.

Hence, the goal of this thesis is to develop an effective software layer that maps the 64-bit applications on to the 16-bit datapath hardware. More specifically, the software layer must *efficiently* :

- **Translate** : The generated code must be equivalent, both in semantics and behavior, to the original 64-bit application code.

- **Optimize** : By harnessing the power of optimizations, the software layer must not only translate but also aggressively optimize in order to reduce the negative impacts of the narrow ISA.

1.3 Contributions

In this thesis, we focus on compiler optimization techniques to generate optimized narrow ISA code from 64-bit ISA programs. The key to assuage the negative impacts of the narrow ISA is to reduce the dynamic code size. This is because a 64-bit instruction is decomposed into a semantically equivalent series of several narrow ISA computations to be executed on the narrow bitwidth architecture (details later in Section 2.2.3). Hence, we approach this task of compiling a 64-bit program to the narrow ISA program with the perspective of *Minimum Required Computations*. Given a program, the notion of minimum required computations (henceforth,

MRC) aims to infer the minimum necessary computations required to generate the same correct output as the original program. Clearly, inferring true MRC from a program requires oracle-information and achieving perfect MRC is an intrinsically ambitious goal.

Towards this end, we have developed three code optimization techniques to approximate the notion of MRC. Some of the proposed techniques are speculative and are used to *prune computations aggressively*, while embedding checks into the non-speculative slices of narrow ISA computations making them self-sufficient to detect mis-speculation. The thesis also explores another optimization technique which *reorders narrow computations* such that the most-probable minimal necessary computations are executed first; rest are required only if the former do not suffice. All the proposed techniques are profile-based and are aimed at reducing the dynamic code size of the narrow ISA programs.

The following sections describe the main contributions of this thesis in deeper detail.

1.3.1 Global Productiveness Propagation (GPP)

Consider an abstract region of connected series of computations, which delivers outputs via few register locations or state changes in memory. Given this region, the fundamental question that *Global Productiveness Propagation* (henceforth, GPP) aims to resolve is – what is the minimum number of computations required to generate the same (correct) set of outputs for each dynamic execution of the region ? In other words, GPP capitalizes on the observation that many of the outputs of the region remain the same as that at the input of the region.

GPP is a profile-guided, speculative optimization technique that infers the *minimum required data-flow* by pruning narrow computations that are most-probably useless (non-productive). More precisely, GPP speculatively prunes the static backward slices of selected narrow computations: computations that result in the same value in their respective storage location as that at the input of the region. As the pruning is speculative, there is need to devise means to detect the unassumed cases causing mis-speculation. For this purpose, GPP relies on its *Assertion Rules Generator* pass which essentially involves backward-traversal of pruned static backward slices. It embeds checks into the static code stream, which enable the code to detect mis-speculations dynamically.

Conceptually, GPP can be applied on any type of abstract region – superblock, loop, or a function. In this thesis, we have applied GPP on the granularity of a function. In a relaxed, perfect setup (details provided later in Section 2.4.3.3), applying the concept of productiveness at the granularity of a function can potentially reduce the dynamic code size of the narrow ISA programs by 25% on an average. In a realistic setup, however, the overall gains achieved by GPP are up to 6.6% reduction in the committed stream and 4.5% reduction in the number

of cycles for a 1-issue, in-order narrow processor, when an average of 60% of the code has been optimized via GPP. The achieved gains of GPP are far from the upper bound and our investigation has identified the key roadblocks.

1.3.2 Local Productiveness Pruning (LPP)

Local Productiveness Pruning (henceforth, LPP) applies similar principles as GPP, only at a much finer granularity of single narrow computation. It is a profile-based, speculative optimization technique that prunes individual narrow computations while merging the assertion-requirements by following producer-consumer data-flow relationships across a defined region. Similar to GPP, LPP also uses the *Assertion Rules Generator* to make the static code stream self-sufficient to detect the mis-speculations.

The difference in approach of GPP and LPP, however, lies in demarcating the notion of optimization regions vs. atomic regions. GPP assumes a whole function as both the optimization region as well the atomic region. This allows GPP to prune comparatively large connected chains of computations. LPP, on the other hand, assumes a single narrow computation as the optimization region whereas a somewhat larger granularity of connected chain of computations (basic block, or superblock) as the atomic region. Allowing atomic regions to be larger than the optimization region is useful in lowering the overheads of the LPP optimization.

Assuming advance, perfect knowledge of output data values written by all narrow integer computations, our experiments reveal that up to 48% of the dynamic computations (all integer computations) are non-productive. Evaluating LPP as a static optimization reveals that LPP can reduce the dynamic code stream by 20%, with around 15% reduction in number of cycles, and an overall code coverage of about 60%. We believe that the achieved gains of LPP are close to the upper bound of the disposable potential; nevertheless, the key roadblocks are duly identified.

1.3.3 Minimal Branch Computation (MBC)

Minimal Branch Computation (MBC) optimization remains to be a use-case of a broader strategy which applies the notion of a *lazy computation model* to reduce the dynamic code footprint of the narrow ISA : the narrow chunks of data are generated only when required, deferred otherwise. One possible way of achieving this (lazy) computational model is by reordering the backslices containing narrow computations such that the *minimal necessary computations* to generate the same (correct) output are performed in the *most of the times*; the rest of the computations are performed only when necessary.

Minimal Branch Computation optimization is a profile-based, code reordering optimization which is based on the philosophy of reordering computations around conditional branches such that those computations which are *most-probably sufficient* [1] to generate the correct value(s) of the required condition codes are placed (and executed) first. The rest are executed only if the former were insufficient.

Our evaluations suggest that narrow ISA programs typically spend 20% of the total number of computations in *only* determining the direction of conditional branches (average across our choice of workloads in the narrow ISA). Further, another fundamental property of these computations is that they collectively contribute to generate few bits of information (known as flags); these bits represent *properties* of data values. Mode importantly, these properties of data values can sometimes be inferred without the knowledge of the precise data value. This forms the underlying motivation to redefine the notion of *Minimum Required Computation* for conditional branches, and hence the optimization heuristic of Minimal Branch Computation. MBC can prune an average of 3.12% of the dynamic narrow computations across the set of workloads.

1.4 Related Work

As mentioned previously, narrow computations have been traditionally defined as those computations that do not require the full data width of the processor. This section annotates the previously studied approaches of exploiting narrow computations. These approaches have varied primarily in the extent of exploiting the narrow computations by software or hardware.

1.4.1 Hardware-Oriented Approaches

First, we visit the hardware-only approaches in literature. These approaches devise changes in the hardware structures to best exploit the narrow computations. Hence, the application-stack remains oblivious to what the hardware performs underneath.

Narrow computation conscious pipeline design [8] reduces the register, logic and cache activity in the processor by appending data, addresses and instructions by two or three extension bits to indicate the significant byte position. By doing so, up to 30-40% reduction in the dynamic activity can be achieved. Various configurations ranging from byte serial pipeline to full-width pipeline stages with operand gating were studied; each provides benefits but at the cost of an increased CPI.

[1] profile-based learning

Packing variable-width operations to increase the effective issue width of the processor has also been proposed [36]. The *Multi-Bit-Width (MBW) Microarchitecture* predicts the data widths of simple single-cycle integer computations (those that can execute on 'simple' integer ALU) and shares the datapath amongst multiple narrow computations in a MIMD like approach. The data width of an instruction is defined as the maximum of its input operand widths and its output value width. The width of a data value is defined as the position of the first zero bit such that all bits in more significant positions are also zero. It achieves an overall speedup of 7% for SPECint2000 benchmarks over an out-of-order processor with no bitwidth prediction and dynamic subword instruction packing mechanism as the baseline.

Brooks and Martonosi [4] propose hardware mechanisms that dynamically recognize and capitalize on the *narrow bitwidth* instances. Two hardware-based optimizations are proposed, both of which require little additional hardware, but neither of these requires compiler support. The first optimization uses clock-gating to reduce power consumption by disabling certain functional units. It has been observed that the power consumed by the integer execution unit can be reduced by 54.1% for SPECint95 suite. The second proposed optimization improves performance by dynamically recognizing, at issue time, opportunities for packing multiple narrow operations into a single ALU. An average speedup of 4.3%-6.2% (depending on the processor configuration) can be achieved using the second technique.

1.4.2 Software-Oriented Approaches

A fundamental limitation of the hardware-only approaches is that hardware allows a limited visibility of the application's behavior : only those computations currently being executed can be exploited. A software-based approach soothes this limitation by allowing a more global and larger vision of the program space. This is the prime motivation behind some of the previous proposals (detailed next in the upcoming paragraphs) that harness static, compiler analysis to best exploit the narrow computations.

A bitwidth aware register allocation algorithm [56] tackles the issue of the inefficient use of wide register files. The proposed algorithm can reduce register requirements by 10% to 50%. Explicit Bit-Section Instruction Extensions [22, 35] have also been exploited to pack multiple subword variables into a single register. Some embedded processor ISAs already allow explicit bit-section referencing. The reduction in instruction counts for the modified functions varies between 4.26% and 27.27%. Similarly, the reduction in cycle counts for modified functions varies between 0.66% and 27.27%.

Static analysis *Bitwise* [55] is an abstract interpretation that computes data ranges required for both pointers and integers in a program. This is achieved by both forward and backward propagation of static information in the program data-flow graph. The proposal is integrated

with the DeepC Silicon Compiler. By taking advantage of bitwidth information during architectural synthesis, it reduces silicon real estate by 15% to 86%, improves clock speed by 3% to 249%, and reduces power by 46% - 73%.

Özer et al. [43, 44] propose a stochastic bitwidth estimation technique for compact and low-power custom processors. By following a simulation-based probabilistic approach, a stochastic bitwidth estimation technique has been introduced which estimates the bitwidths of integer variables using extreme value theory. The estimation technique is also empirically compared to two compile-time integer bitwidth analysis techniques. The experimental results show that the stochastic bitwidth estimation technique dramatically reduces integer bitwidths and, therefore, enables more compact and power-efficient custom hardware designs than the compile-time integer bitwidth analysis techniques.

All the foregoing research stress on two main aspects. One, there exists significant amount of narrow computations and future general-purpose and reconfigurable architectures should strive to capture a portion of these gains. Second, a compiler can play a significant role in detecting and exploiting narrow bitwidth computations.

1.4.3 Hardware-Software Collaborative Approaches

Hardware-software collaborative approaches combine the best of both worlds : hardware is designed for efficiency and software ensures that the efficacy of the hardware is exploited to its best potential.

Razdan and Smith [48, 49, 50] propose a static analysis to infer narrow data widths for use with a tightly-coupled configurable functional unit. Their analysis is bit-wide abstract interpretation over the bit positions of each variable in an internal representation of the program, with forward and backward passes to characterize the generation and the use of bit positions. Using the aforementioned bitwidth-specific optimization technique with other optimizations, 22% performance gain on the SPECint92 benchmark suite has been observed with a modest hardware investment (a single combinational PFU).

Program In Chip Out, PICO [28] is a research effort by HP labs to design a system for automatically generating customized hardware. PICO-NPA [51] is one offshoot of PICO, that automatically generates non-programmable accelerators (NPAs) for a hot loop written in C. The overall aim is also to synthesize cost-effective hardware systems. Towards this end, exploiting *user-defined integer bitwidths* is an important contributor [37]. Hence, the PICO-NPA is a very specific design which exploits user-defined bitwidths to synthesize VHDL-code and then it applies operation-scheduling in a width-conscious manner to achieve cost-effective, application-specific, non-programmable accelerators.

BitValue Inference algorithm [5] works intraprocedurally to figure out the *'useful'* bits. It is a *data-flow analysis* to discover bits which are independent of the program inputs (constant bits) and bits which do not influence the program output (unused bits). For example, a bit which is later masked with zero is unused and hence, not useful. Using forward and backward data-flow analyses, BitValue Inference algorithm reveals that around 36% of the computed bytes are thrown away. This algorithm achieves up to 20-fold reductions in the size of the synthesized hardware, when integrated with a compiler for a reconfigurable hardware logic (which has the advantage of supporting non-standard data widths).

Applying the same definition of '*usefulness*' as done in the BitValue Inference algorithm [5], but on a data-word granularity, Canal et al. [9] propose a static optimization technique known as Useful Value Range Propagation (UVP). Using a backward and forward propagation of 'useful' value ranges achieves more number of narrow operations than the 'conventional' value ranges. Conventional value ranges are determined solely on user-specified data-types and variable instantiations. On the other hand, 'useful' value range additionally exploits masking operations, shift operands, and other implicit value range specifications through static analysis of the programs. Using this technique, the number of 64-bit instructions which are originally 51% of the total number of instructions are reduced to 42% for SPECInt95 programs. Also, the overall energy savings in the processor which implements power-gating are close to 6% (on an average) for the SpecInt95 suite (18% for the functional units, and around 15% for instruction-queues, rename buffers, register file and the result buses).

Further, they also propose a profile-driven optimization technique known as Value Range Specialization (VRS). VRS is similar in its approach to the UVP technique, except that the former uses profiling techniques to determine the value ranges. The VRS technique *specializes* hot code by using on these dynamic *value range* profiles. It involves a cost-benefit analysis which prioritizes energy-savings expected from the specialization of a certain candidate. VRS basically duplicates the regions of code that are affected by the specialization, and then inserts tests to dynamically select the region that will be executed: either the specialized or the non-specialized one. The proposed technique achieves an overall $energy * delay^2$ reduction of 14% for the SpecInt95 set (the $energy * delay^2$ benefits are over 20% for data-intensive structures like FUs). Note that, both UVP and VRS are software approaches, and operand-gating at each pipeline stage is required from the hardware to collaboratively exploit narrow bitwidth computations.

Lastly, speculative bitwidth management by the compiler has also been proposed [47]. The strategy is to exploit high *bitwidth locality* between individual instructions of a single basic block to create narrow-width regions by a compiler. Hardware enhancements for mis-speculation recovery management are exploited. The ISA is also enhanced with a hardware-exposed datapath-width reconfiguration instruction. This instruction is embedded by the com-

piler which provides a hint for the hardware to predict the execution width of the subsequent regions. It achieves energy savings of 17% in the datapath's dynamic energy and 22% in the register files' static energy by *clock-gating* and *byte-slice* register file. Byte-slice register file organizes the register file data into multiple sets of 8-bit slices, some of which can be turned off in a low-power mode [19]. Note that, creating narrow bitwidth regions has the advantage of reducing the overhead of clock-gating on a cycle-by-cycle basis (the latter is assumed by Canal et al. [8, 9]).

1.4.4 In Perspective

The first and foremost difference between all the above-mentioned work and this thesis is in the fundamental approach. Given the high percentage of narrow computations, furthered by our definition of *productiveness*, we envision a hardware/software collaborative approach towards exploiting narrow computations. The hardware is designed for computational efficiency for executing the most common case, i.e., the 16-bit datapath computations. This is called the narrow bitwidth architecture, which essentially combines a 16-bit datapath with 64-bit addressing capability (explained in deeper detail in Chapter 2). The software is entrusted with the task of efficiently mapping the 64-bit applications on to the narrow bitwidth architecture based processor.

This thesis views the computations with a different standpoint – *Everything is Narrow*, i.e., all computations can be decomposed into a set of 16-bit instructions which collaboratively maintain the same semantic meaning as the 64-bit counterpart. This is unlike the traditional definition of narrowness where narrow computations are defined as those computations that do not need the available bitwidth the hardware. Henceforth, in this thesis, the term *narrow computations* is used to unequivocally refer to the strictly 16-bit datapath computations.

This thesis proposes three optimizations (two of them being speculative, and the third non-speculative) to generate optimized narrow ISA programs. Unlike the previous works [5, 37, 55], which use conservative user-defined data widths, the proposed optimizations in this thesis use value profiles (and hence, are more aggressive). The only exception is the work by Pokam et al. [47]. However, speculative bitwidth management [47] uses the definition of sign-extension to prune chunks of wide computations. In contrast, the optimizations proposed in this thesis (Global Productiveness Propagation and Local Productiveness Pruning) exploit a new definition called *productiveness*).

The definition of value range [5, 9] to uncover narrow computations is also more aggressive than the conventional user-defined bitwidths. Two of the proposed optimizations in this thesis (LPP and GPP) optimize a region using the definition of *productiveness*. We show in Section

4.4.2 and Section 2.4.3 that the definition of productiveness is more aggressive than that of value range, and hence, the former provides more opportunities.

Further, in many of the previous proposals, the hardware remains wide [4, 9, 36, 47]. In one of the works, a special instruction is used to enable clock-gating speculatively for a narrow bitwide region [47]. Other proposals reduce the dynamic activity in the processor by making use of hardware-based operand-gating. Hence, in contrast to [9, 47], our work does not require hardware enhancements like clock-gating.

Finally, some more related work which is very specific to the respective discussion in each chapter is embedded at the relevant points throughout the thesis.

1.5 Organization of the Document

The thesis document is organized as follows.

First, Chapter 2 provides an overview of the narrow bitwidth architecture from both the hardware and software perspectives. The chapter describes how the micro-architecture of the narrow bitwidth architecture looks like. Then the narrow ISA, which forms the software interface of the narrow bitwidth architecture to the outside world, is described. Also described is the Narrow Translation Scheme, which is used to map the 64-bit operations to the equivalent set of 16-bit narrow ISA computations. Chapter 2 also motivates the role of the compiler in this thesis and proposes the overall perspective taken in this thesis to reduce the dynamic code footprint of the narrow ISA.

Chapter 3 describes the simulation infrastructure with detailed configurations. It also describes the workloads and the metrics used to evaluate the proposed optimizations.

The three proposed optimizations are organized as Chapter 4 (Global Productiveness Propagation), Chapter 5 (Local Productiveness Pruning) and Chapter 6 (Minimal Branch Computation). All proposed optimizations are described in detail with selected algorithms and performance evaluations. For each optimization, the thesis tries to present a clear idea on not only the best achievable potential, but also what is eventually achieved and the observed roadblocks, if any.

Finally, Chapter 7 provides a summary of the thesis and some interesting future directions related to this work.

2

Narrow Bitwidth Architecture: A Hardware/Software Perspective

This chapter describes the design of the narrow bitwidth architecture and also its software interface to the outside world (the Narrow ISA). The fundamental design goal for the narrow bitwidth architecture (Section 2.1) is to provide a low-cost, low hardware-complexity, simple execution core. It is a strictly 16-bit integer datapath, in-order processor which is best at executing the *common case*, i.e., the narrow computations (16-bit computations) efficiently. The narrow bitwidth architecture is unique in that although the datapath is squeezed to 16-bits, it continues to support 32-bit / 64-bit computations (and hence, the current software stack). This is made possible by a non-conventional memory interface design which combines a 16-bit data interface with a 64-bit address interface.

In this chapter, we evaluate the narrow bitwidth architecture briefly and suggest that it has the potential to offer significant hardware and power savings. Next, the narrow ISA, which is a RISC-like ISA for performing integer computations on the narrow bitwidth architecture, is outlined in Section 2.2. The chapter then evaluates the behavior of the narrow ISA, specifically the increased dynamic code footprint that is characteristic of the narrow ISA programs, when compared against the 64-bit wide ISA programs.

As the dynamic code footprint increases significantly, the central issue is how to alleviate the negative performance impact. This chapter introduces and emphasizes the importance of a compiler in such an ecosystem. Section 2.3 outlines some key opportunities that have been identified in this direction. In this thesis, the task of reducing the negative performance impact of the narrow ISA is assigned to the software in the form of dynamic optimizer / compiler. The

chapter is concluded with some preliminary evaluations of the identified opportunities which form the basis of the proposed optimizations.

2.1 Micro-Architecture

In the proposed narrow bitwidth architecture, all 'processing' elements, viz., the execution units, the data bypass logic, the register file and the data interface to memory are squeezed down to a data width of 16-bits. This redesign of the hardware furnishes significant area and power savings and hence, an increase in the computational efficiency. It also exposes the opportunity to scale frequency further as the execution core is extremely simple.

2.1.1 Processor Datapath

Figure 2.1 shows the implementation of a MIPS-like data path [23] of a simple five-stage pipeline. The use of a five-stage pipeline in this section is for illustration purposes only. The specific micro-architecture design and configuration used in this thesis for evaluating the proposed code optimizations is detailed in Chapter 3. More importantly, in Figure 2.1, those datapath signals that may remain 64-bits are shown in black, while those that are scaled down to 16-bits are shown with gray dotted lines.

Processing Elements. With regard to the register file, the narrow bitwidth architecture incorporates a design which is a naive extension of a 64-bit register file. In other words, the register file simply has the data interface reduced to 16-bit and the number of register names are effectively 4 times that in the 64-bit design.

Although each 16-bit wide register is an independent register and it is treated as such, there is a strong conceptual binding between a 64-bit ISA register and the corresponding four 16-bit narrow ISA registers. Thus, we often use the term 'i^{th} chunk' of a 64-bit ISA register to refer to the appropriate 16-bit register that holds the particular 16-bit portion of the 64-bit value. This design implies that all reads/writes to the register file are 16-bit wide only. For example, we often use the naming convention of 16-bit register names as rax_0, rax_1, rax_2 and rax_3 to describe the 4 registers used to map the value of the 64-bit register rax.

Bypass logic is reduced by a factor of 4 because instead of forwarding 64-bit values, only 16-bit values need to be forwarded. Similar reduction can be seen in other computational resources like integer ALU's. The internal registers (A, B, Imm, IR, L, ALU output) are also reduced to 16-bit data width.

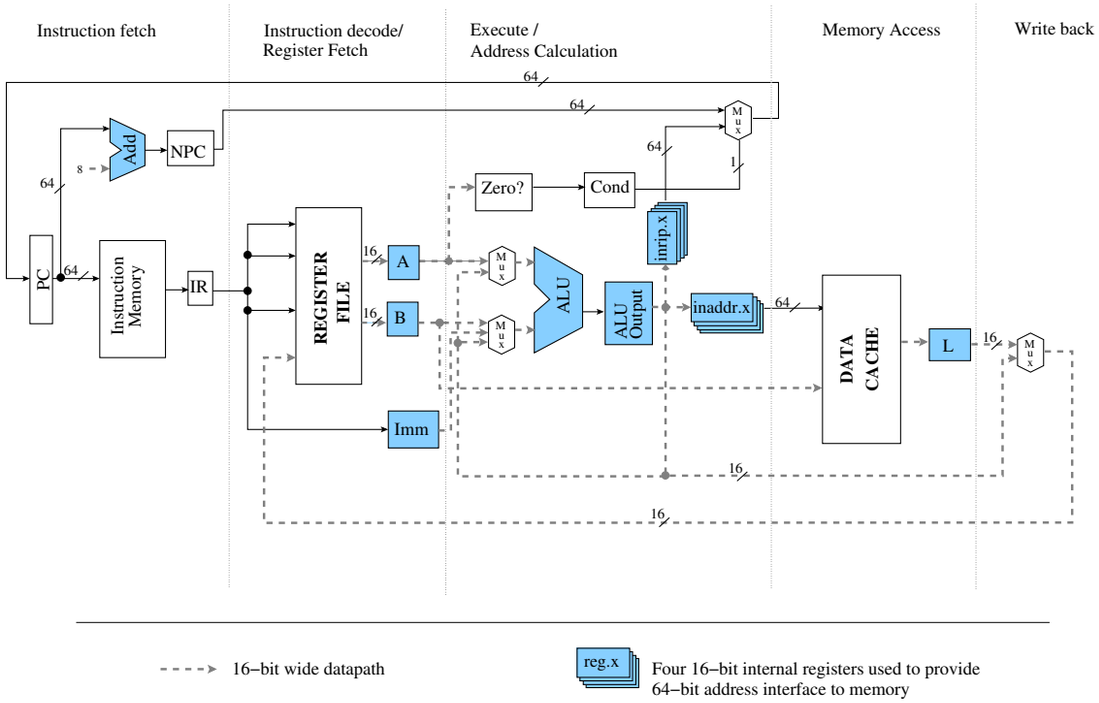

Figure 2.1: The hardware redesign opportunities using a MIPS-like datapath – 64-bit vs. 16-bit wide implementations

Memory Interface. Many 16-bit data commercial microprocessor designs have been used widely in the embedded domain. The narrow bitwidth architecture advocated in this thesis, however, differs from the existing (e.g. Intel 8086) or proposed 16-bit datapath designs with respect to an important design point : the memory interface. As previously mentioned, the narrow bitwidth architecture combines a 16-bit integer datapath in an in-order pipeline with a 64-bit wide address path (see Figure 2.1). This is done to support large memory addressing capabilities which are required by many current execution environments.

Figure 2.1 shows the two supplementary registers added to the narrow bitwidth architecture for enabling the 64-bit memory interface – (i) *inaddr*, which is a 64-bit internal register used to enable 64-bit data interface to memory, and (ii) *inrip*, which is another 64-bit internal instruction pointer register used to enable the 64-bit address interface to memory for instruction accesses. For each access to the instruction or data memory (64-bit address), additional instructions generated by the software flow through the pipeline to update the four individual 16-bit chunks of these registers incrementally to reconstruct the 64-bit values. More details on the same are discussed in the upcoming Section 2.2.2.

As stated previously, Figure 2.1 only shows key pipeline structures to illustrate what a 16-

bit pipeline has to offer as compared to its 64-bit counterpart. Although not shown, hardware structures for effective branch prediction (branch predictor, branch target buffer, and the RAS) do form an integral part of the instruction fetch stage. Branch resolution takes place at the execute stage of the pipeline, following which a mis-prediction, if any, can be handled.

Considering the scalar pipeline of Figure 2.1, every cycle a new narrow ISA instruction is fetched using the predicted / correct 64-bit PC value. The instruction bytes hence loaded into the IR register are then decoded. Next, 16-bit register values are read from the register file per instruction and a 16-bit ALU operates on these input operands to generate 16-bit output data value. The computed result is available via the bypass network, and is eventually written back to the register file in the next stage of the pipeline.

A processor based on the narrow bitwidth architecture described above is referred to as the *narrow processor* in this thesis. Section 2.2 explains how 64-bit computations are mapped on to the narrow ISA so that they can be executed on the narrow processor. In the narrow processor, interrupts may be masked until the four individual chunks of a 64-bit register are updated. Hence, exceptions (interrupts or traps) can be allowed only at the boundary of a bundle of narrow ISA computations corresponding to the same 64-bit wide instruction. This approach is similar to how contemporary Intel processors mask interrupts at the x86 macro-op boundary. Although they crack CISC instructions into simpler micro-operations that more closely resemble RISC instructions (using the hardware decoder), exceptions and interrupts are managed at the user instruction boundaries.

2.1.2 Brief Comparison with 64-bit Wide Architecture

A comparison of the area and power benefits of a 16-bit datapath processor versus a 64-bit datapath processor is not the focus of this thesis. Nevertheless, we derive some trends based on previous research work to highlight how important the area and power benefits of the narrow bitwidth architecture can be.

Area. An interesting analysis of *potential savings in a 16-bit data/instruction microprocessor* was performed in previous research [59]. Compared to its 32-bit counterpart (ARM7), the 16-bit THUMB instruction set microprocessor requires only 40% gate count, 51% power consumption and can be clocked at 160% clock frequency. The proposal also reduces the register file size by 21% in the 16-bit implementation. It affirms the benefits of a 16-bit design over a 32-bit. More importantly, these findings, when extrapolated, also indicate that the hardware and power savings with respect to a 64-bit counterpart will only be proportionally higher.

ALUs and Data-Bypass. The logic complexity of the data bypass logic has been previously shown to be proportional to the square of the issue width and linear to the number of pipeline stages after the first result-producing stage [1, 45]. The logic complexity of the *data bypass circuit* is also expected to be linear function of n – $\Theta(n)$, where n is the bitwidth of the data being transferred. Hence, if the rest of the pipeline structures are the same, the hardware cost of bypassing 16-bit data values must be 1/4 of that of bypassing 64-bit values.

With regard to the FUs, let us unrealistically assume that all the logical and arithmetic operations (addition, subtraction, and other harder operations like shift, multiplication etc.) can be implemented with the same logic complexity as a CSCSA adder circuit - i.e., Θ(nlogn) [12]. Since a realistic implementation of complex FUs like shifter, rotate etc. will only be *harder* than an adder, the area occupied by the 16-bit ALUs must be at least 1/6 (16log16 / 64log64) of the area occupied by 64-bit ALUs. On an in-order chip like Intel Atom, this implies $\sim$ 5-7% area reductions, as ALUs occupy a relatively small space on the chip [58].

Memory Array Structures. In the narrow bitwidth architecture, the bitwidth of the accessed values from the memory structures close to the processor (register file and L1 Data cache) is limited to 16-bits. This allows the data-arrays in these structures to be banked and/or sub-banked as only 16-bits are accessed per request.

Some previous researchers have studied the opportunities of register file redesign by exploiting narrow computations. *bit-partitioned register file* (BPRF) [30] and *byte-slice register files* [47] demonstrate that a finer granularity (the split data-storage) allows a more efficient use of the register file. Multi-banking has also been exploited to reduce the dynamic activity in the register files using extension bits in *Very low power pipelines* design [8]. When the register file is accessed, first the low order data bytes and the extension bits are read. Depending on the values of the extension bits, additional register bytes may be read during subsequent clock cycle(s). Though we do not exploit these narrow register file array structures in the narrow bitwidth architecture, note that all of these aforementioned techniques (BPRF, sliced register file, multi-banked register file) fit gracefully to our proposed narrow bitwidth architecture.

We present preliminary analytical comparisons to compare simple non-banked 64-bit and 16-bit register file designs using CACTI 6.5 [42]. The results and the studied configurations are illustrated in Table 2.1. The number of registers in 64-bit register file suitable for an in-order pipeline are assumed to be 32 (each of 64-bit size). Similarly, the number of registers in the 16-bit register file suitable for an in-order pipeline in the narrow bitwidth architecture are assumed to be 128 (each of 16-bit size).

It has been observed that a 16-bit register file is a better design than a 64-bit register file when both the designs have the same number of read / write ports. This holds true especially for the metrics of area and energy per access for various design objectives – delay, dynamic power,

Table 2.1: Register file design evaluation : comparing 16-bit vs. 64-bit data cells

[16-bit RF Configuration] : RF Size = 128 (32∗4) entries of 2B each, 8 read/4 write ports, 1-bank, 7 tagbits, Technology = 32nm, **Read out bits = 16-bits**								
[64-bit RF Configuration] : RF Size = 32 entries of 8B each, 8 read/4 write ports, 1-bank, 5 tagbits, Technology = 32nm, **Read out bits = 64-bits**								
Design Objective	Access-time (ns)		Cycle-time (ns)		Area (mm^2)		Energy Per Access (nJ)	
Out_W	16-bits	64-bits	16-bits	64-bits	16-bits	64-bits	16-bits	64-bits
Delay	0.166	0.165	0.133	0.133	0.027	0.048	0.001	0.002
Dynamic Power	0.225	0.188	0.185	0.185	0.035	0.078	0.001	0.002
Area	0.195	0.165	0.138	0.133	0.025	0.048	0.001	0.002
ED^2	0.167	0.167	0.136	0.136	0.027	0.048	0.001	0.002

area and energy delay square product (ED^2). As shown in Table 2.1, when the register file designs are optimized for ED^2, a 16-bit design achieves 42% reduction in energy per access, and occupies 43% less area together with no impact on cycle-time / access-time.

In the data caches, a brief evaluation using CACTI 6.5 [42] reveals that sub-banking at 16-bit granularity of a 32KB, 4-way associative L1 data cache is effective (refer to Table 2.2). Different cache design objectives have been studied – delay, dynamic power, area and energy delay square product (ED^2). As shown in Table 2.2, when the cache is optimized for energy delay square product, sub-banking at 16-bit granularity achieves 50% reduction in energy per access. Such a cache design also occupies 27% less area with negligible effect on cycle-time / access-time.

2.2 Narrow ISA

This section describes the interface of the narrow bitwidth architecture to the software stack, also referred to as the narrow ISA. The narrow ISA is a RISC-like ISA and highlights the different standpoint of viewing computations that this thesis adopts – *Everything is Narrow* : all 64-bit computations can be decomposed into a set of 16-bit computations which perform the semantically equivalent operation as the former. The upcoming sections describe how the narrow ISA unfolds both opportunities and challenges.

The design of the narrow ISA is highly influenced by the x86 / IA64 [25, 26, 27] ISAs. This will be apparent in the rest of the thesis as it uses the same register names, absolute branch

Table 2.2: DCache design evaluation : comparing 16-bit vs. 64-bit data interface

[16-bit DCache Configuration] : Cache Size = 32KB, Banks = 4, line size = 64 B, associativity = 4, Read/Write Ports = 2, Technology = 32nm, Sub-banking at 16-bit granularity								
[64-bit DCache Configuration] : Cache Size = 32KB, Banks = 4, line size = 64 B, associativity = 4, Read/Write Ports = 2, Technology = 32nm								
Design Objective	Access-time (ns)		Cycle-time (ns)		Area (mm^2)		Energy Per Access (nJ)	
Out_W	16-bits	64-bits	16-bits	64-bits	16-bits	64-bits	16-bits	64-bits
Delay	0.263	0.268	0.198	0.198	0.216	0.291	0.007	0.015
Dynamic Power	0.354	0.268	0.341	0.198	0.184	0.249	0.007	0.014
Area	0.351	0.338	0.278	0.262	0.176	0.246	0.008	0.015
ED^2	0.263	0.268	0.198	0.198	0.182	0.249	0.007	0.014

addressing, conditional branch opcodes, and status flag bits amongst others. Further, the RISC-like operations composing the narrow ISA are semantically very close to the micro-operations implemented in the open-source simulator and virtual machine for x86 / x86-64 ISAs [62].

2.2.1 An Absolute Narrow Paradigm

Previous proposals [4, 5, 8, 9, 36, 47] have identified narrow (bitwidth) computations as those operations whose bitwidth precision requirements are less than the width of the available datapath. For example, an *add rax = rbx, 0x4* is a narrow computation needing only 16-bit wide path if both *rax* and *rbx* hold the value of a variable declared *short int*. Moreover, throughout the thesis document, we use the notation – opcode dest = src1, src2 to denote an operation.

Theoretically, each wide (64-bit) computation can be broken down into an *equivalent set* of narrow computations, in which the 64-bit value is computed as four independent 16-bit chunks. For example, consider a 64-bit wide add operation – *add rbp = rsp,176*, which adds a constant to register *rsp* and writes the result into *rbp*. This can be decomposed into a sequence of semantically equivalent narrow 16-bit RISC operations as –

$$add\ rbp_0 = rsp_0, 176;$$
$$addc\ rbp_1 = rsp_1, 0;$$
$$addc\ rbp_2 = rsp_2, 0;$$
$$addc\ rbp_3 = rsp_3, 0;$$

where register name R_i refers to the i^{th} chunk (16-bit data) of the equivalent 64-bit register register R as commented before. *addc* has the same semantics as an *add* operation, except

that it consumes the carry flag of the previous operation and adds it to the result. Moving forward, the entity that performs the task of decomposing each 64-bit computation into a set of equivalent 16-bit narrow computations is referred to as the *Narrow Translator* in the rest of the thesis document.

Definition 2.1 (**Chunk**). A *chunk* is a 16-bit data location. This is the output unit of a narrow computation in the narrow ISA. $Chunk_0$ refers to the least significant chunk, i.e., data value from bit 0 to 15. $Chunk_1$ refers to the next chunk in order, i.e., data value from bit 16 to 31. $Chunk_2$ refers to next chunk in order, i.e., data value from bit 32 to 47. $Chunk_3$ refers to the most significant chunk, i.e., data value from bit 48 to 63.

2.2.2 Wide to Narrow Translation Scheme

Intuitively, all integer operations OP_{int} can be decomposed into a semantically equivalent set of narrow operations : OP_{int}^{N0}, OP_{int}^{N1}, OP_{int}^{N2} and OP_{int}^{N3}, each representing the set of operations needed to generate the respective 16-bit chunk (*N0, N1, N2* and *N3*). Each of these RISC-like computations that write a 16-bit data (with flags if applicable) constitute the narrow ISA. The number of operations in this semantically equivalent set of narrow operations may vary from four (e.g. for add/sub and logic operations on 64-bit operands) to tens of narrow operations (e.g. in case of integer shift/rotate operations on 64-bit operands).

Narrow ISA Flags Semantics. The design of the narrow ISA is highly influenced by the x86 / IA64 [25, 26, 27] ISAs. This is because the baseline ISA in our simulation infrastructure is x86-64 based. With respect to flags, many x86 arithmetic instructions modify some or all of the status flag bits. Hence, it is important for the narrow translator to generate semantically equivalent code not only with respect to data values generated but also flag values. The narrow translator ensures correct state generation with respect to the *five flags relevant to normal execution* : Zero, Parity, Sign, Overflow and Carry.

The foregoing flags can be cumulatively calculated by the set of narrow computations. For example, zero flag (ZF) can be calculated as the logical *or* of the zero flags for the composite narrow chunks. Carry flag (CF) can be calculated by propagating the carry flag of each lower significant chunk forward to the next more significant chunk. Sign flag (SF) is updated correctly by the most significant chunk, if the carry flag has been propagated correctly. Parity flag can also be calculated by accumulating the parity of each chunk corresponding to the same wide 64-bit operation.

This accumulation of flags to generate the final x86 equivalent flag values is performed by special opcodes like addc (add and carry), subc (subtract and borrow), shlc, shrc etc. The opcode names are terminated by a *c* only to ensure a uniformity in the opcode names; each of these opcodes combines all the relevant flags from the previous computation. The flag dependences between narrow ISA computations can be managed explicitly by exploiting the well-known ZAPS rule [62] : any instruction that updates any of the Z/P/S flags updates all three flags, so in reality only three flag entities need to be tracked: ZPS, O, C.

Hence, selected arithmetic / logical narrow ISA computations generate a 16-bit data value together with the requisite flag values.

x86 Merging Rules and the narrow ISA. In the x86 ISA, the size of an operation (henceforth, referred to as opsize) is normally indicated by its register operands. For example, the use of RAX, EAX and AX/AL/AH indicate a 64-bit, 32-bit and 8/16-bit computation respectively. Our implementation of narrow translator abides by the x86 sizes and generates less than the worst-case translation whilst following the *x86 merging rules*[1] according to the opsize.

Definition 2.2 (**x86 Merging Rules**)**.** x86 compatible ALUs implement operations on 1, 2, 4, or 8 byte quantities. Given the operation size, an operation *Op rd = ra, rb* computes the result as follows :

- 8-bit : Low byte of rd is set to the 8-bit result; higher 7 bytes of rd are set the same as the corresponding bytes of ra.
- 16-bit : Lower 2 bytes of rd are set to the 16-bit result; higher 6 bytes of rd are set the same as the corresponding bytes of ra.
- 32-bit : Lower 4 bytes of rd are set to the 32-bit result; higher 4 bytes of rd are cleared to zero.
- 64-bit : All 8 bytes of rd are set to the 64-bit result.

The upcoming paragraphs provide more details of the narrow translations generated for the most common operation classes. Relevant details on how the x86 merging rules are applied are also given. For the rest of this thesis, we use the following terminology : register name R_i refers to the i^{th} chunk (16-bit data) of the equivalent 64-bit register register R. R_0 holds the least significant chunk. Similarly, immediate name Imm_i refers to the i^{th} chunk of the equivalent 64-bit immediate, Imm_0 specifies the least significant chunk.

[1] briefly mentioned in [62]

Table 2.3: Template based translation for arithmetic / logical operations. Op indicates the operation. Opc indicates the narrow ISA opcode which performs the operation and accumulates the associated flags

Wide Operation (64-bit)	Equivalent set of Narrow Operations		
	opsize = 8/16-bit	opsize = 32-bit	opsize = 64-bit
	Template I : destination register same as input register		
Op regD = regD Imm	Op $regD_0$ = $regD_0$ Imm_0	Op $regD_0$ = $regD_0$ Imm_0 Opc $regD_1$ = $regD_1$ Imm_1	Op $regD_0$ = $regD_0$ Imm_0 Opc $regD_1$ = $regD_1$ Imm_1 Opc $regD_2$ = $regD_2$ Imm_2 Opc $regD_3$ = $regD_3$ Imm_3
	Template II : destination register different from the input register(s)		
Op regD = regA regB	Op $regD_0$ = $regA_0$ $regB_0$ mov $regD_1$ = $regA_1$ mov $regD_2$ = $regA_2$ mov $regD_2$ = $regA_3$	Op $regD_0$ = $regA_0$ $regB_0$ Opc $regD_1$ = $regA_1$ $regB_1$ mov $regD_2$ = zero mov $regD_2$ = zero	Op $regD_0$ = $regA_0$ $regB_0$ Opc $regD_1$ = $regA_1$ $regB_1$ Opc $regD_2$ = $regA_2$ $regB_2$ Opc $regD_3$ = $regA_3$ $regB_3$

Arithmetic / Logical Operations. Two selected templates for the arithmetic / logical operations are shown in Table 2.3. The first sample shows a template where both the source and destination operands are the same. Op indicates the operation. Opc indicates the narrow ISA opcode which performs the operation and accumulates the associated flags. Following the x86 merging rules, such an operation may be translated into 1, 2, or 4 narrow operations if the 64-bit operation size is specified to be 16-bit, 32-bit or 64-bit respectively. In each case, the appropriate register and immediate chunk data are specified, and flags are accumulated accordingly by the use of the aforementioned opcodes.

The second sample shows a template where the source and destination operands are different from each other. Following the x86 merging rules, such an operation is always translated into 4 narrow operations irrespective of whether the operation size is specified to be 16-bit, 32-bit or 64-bit.

Shift Operations. The shift operations are one of the most costly in terms of number of generated narrow operations. This operation class includes the opcodes like shift left (shl), shift right (shr), rotate right (ror), rotate left (rol) etc. Two examples of shl operation are shown in Table 2.4. Translations for other opcodes like shr, ror, rol can be derived from the same.

The shift operations lead to a high number of narrow operations. This is because the computed values always overflow the 16-bit register data boundaries (chunks) and need to be buffered before the higher / lower chunks are updated. The buffering of the (extra) overflown bits from previous chunk in case of shl and shr is performed by few additional opcodes

Table 2.4: A sample translation for the shift left operation. shlc indicates the narrow ISA opcode which performs the shift operation and accumulates the associated flags. shlext indicates the narrow ISA opcode which shifts and buffers the data

Wide Operation (64-bit)	Equivalent set of Narrow Operations		
	opsize = 8/16-bit	opsize = 32-bit	opsize = 64-bit
	Template I : destination register same as input register		
shl regD = regD Imm	shl $regD_0$ = $regD_0$ Imm_0	shlext $tempD_1$ = $regD_0$ Imm_0 shl $regD_0$ = $regD_0$ Imm_0 shl $regD_1$ = $regD_1$ Imm_0 or $regD_1$ = $regD_1$ $tempD_1$	shlext $tempD_1$ = $regD_0$ Imm_0 shl $regD_0$ = $regD_0$ Imm_0 shlext $tempD_2$ = $regD_1$ Imm_0 shl $regD_1$ = $regD_1$ Imm_0 or $regD_1$ = $regD_1$ $tempD_1$ shlext $tempD_3$ = $regD_2$ Imm_0 shl $regD_2$ = $regD_2$ Imm_0 or $regD_2$ = $regD_2$ $tempD_3$ shl $regD_3$ = $regD_3$ $tempD_1$ or $regD_3$ = $regD_3$ $tempD_3$
	Template II : destination register different from the input register		
shl regD = regA Imm	shl $regD_0$ = $regA_0$ Imm_0 mov $regD_1$ = $regA_1$ mov $regD_2$ = $regA_2$ mov $regD_3$ = $regA_3$	shlext $tempD_1$ = $regA_0$ Imm_0 shl $regD_0$ = $regA_0$ Imm_0 shl $regD_1$ = $regA_1$ Imm_0 or $regD_1$ = $regA_1$ $tempD_1$ mov $regD_2$ = $regA_2$ mov $regD_3$ = $regA_3$	shlext $tempD_1$ = $regA_0$ Imm_0 shl $regD_0$ = $regA_0$ Imm_0 shlext $tempD_2$ = $regA_1$ Imm_0 shl $regD_1$ = $regA_1$ Imm_0 or $regD_1$ = $regA_1$ $tempD_1$ shlext $tempD_3$ = $regA_2$ Imm_0 shl $regD_2$ = $regA_2$ Imm_0 or $regD_2$ = $regA_2$ $tempD_3$ shl $regD_3$ = $regA_3$ $tempD_1$ or $regD_3$ = $regA_3$ $tempD_3$

called shlext and shrext respectively. For example, as shown in Table 2.4, to generate $regD_1$, first shlext shifts $regD_0$ and buffers up the overflowing bits beyond the default lower 16-bits in $tempD_1$. The final value of the output register $regD_1$ is calculated as the logical *or* of the buffered bits in $tempD_1$ together with the $regD_1$ shifted left.

Branches. The datapath to update the PC also remains 16-bit wide. A 64-bit conditional branch operation which probes a condition code (cond) and updates the instruction-pointer register (rip) depending on the outcome – br rip = cond, Imm (where Imm is the absolute target address), is translated to five narrow operations. Table 2.5 shows the equivalent set of narrow computations that include four mov operations to update an internal register *inrip*, followed

by a branch operation. The 64-bit value co-operatively held by the four 16-bit inrip.x registers describes the taken branch address.

Table 2.5: Translation for a branch operation template. Branches are always 64-bit sized operations

Wide Operation (64-bit)	Equivalent set of Narrow Operations
br rip = cond, Imm	mov $inrip_0$ = Imm_0 mov $inrip_1$ = Imm_1 mov $inrip_2$ = Imm_2 mov $inrip_3$ = Imm_3 br rip = $cond$, $inrip$

For indirect branches, in which the taken address is not expressed by an immediate, but by a register (as it needs to be loaded from memory most of the times), the four mov operations will copy a register value instead.

Memory Operations. As mentioned previously, the *data interface* to memory is squeezed to be 16-bit wide, while the *address interface* remains 64-bit wide.

Hence, a load or store operation may be translated into eight narrow ISA instructions, as shown in Table 2.6. Each of the first four of these instructions (addition operations) compute a chunk of the address, hence updating an internal register *inaddr* that provides the 64-bit address interface to the memory hierarchy, while the other four are the individual 16-bit loads/stores. Note that, if a following ld/st uses the same address, it would be translated into just the 4 latter instructions, since the address would still be in the internal register *inaddr*.

Table 2.6: Translation for a load operation template. Similar translations are generated for store operations

Wide Operation (64-bit)	Equivalent set of Narrow Operations		
ld rax = Mem[rbx,Imm]	opsize = 8/16-bit	opsize = 32-bit	opsize = 64-bit
	add $inaddr_0$ = rbx_0, Imm_0	add $inaddr_0$ = rbx_0, Imm_0	add $inaddr_0$ = rbx_0, Imm_0
	addc $inaddr_1$ = rbx_1, Imm_1	addc $inaddr_1$ = rbx_1, Imm_1	addc $inaddr_1$ = rbx_1, Imm_1
	addc $inaddr_2$ = rbx_2, Imm_2	addc $inaddr_2$ = rbx_2, Imm_2	addc $inaddr_2$ = rbx_2, Imm_2
	addc $inaddr_3$ = rbx_3, Imm_3	addc $inaddr_3$ = rbx_3, Imm_3	addc $inaddr_3$ = rbx_3, Imm_3
	ld rax_0 = Mem[$inaddr$]	ld rax_0 = Mem[$inaddr$]	ld rax_0 = Mem[$inaddr$]
		ld rax_1 = Mem [$inaddr$+0x2]	ld rax_1 = Mem [$inaddr$+0x2]
		mov rax_2 = zero	ld rax_2 = Mem [$inaddr$+0x4]
		mov rax_3 = zero	ld rax_3 = Mem [$inaddr$+0x6]

Mul/Div Operations. Multiply and Divide operations are assumed to be handled by special FUs (and hence, are not translated into narrow computations). It has been observed that on an average, Mul/Div operations remain less than 0.21% of the committed 64-bit instructions in our workloads (with an exception of parser with 1%).

Floating-point Operations. A semantically equivalent set of narrow computations can also be constructed for floating-point (fp) computations, but the size of this set may be huge. For example, a simple double-precision (IEEE 754 standard) fp add operation involves : extraction of exponents (11 bits each) and significands (52 bits each), aligning the significands based on exponents, and lastly an addition of the significands and exponents. This makes the number of operations shoot up to 25 odd operations (without including the cost of rounding/normalizing the result if needed).

Such a huge amount of code explosion suggests that, although there may be a potential for realizing narrow computations for fp code as well, the current definition of doing computations at a 16-bit boundary does not perpetuate gracefully for the fp operations. Hence, this thesis concentrates only on the integer operations. For fp computations, we assume traditional 64-bit wide execution units and datapath.

Conclusions. In this section, we have outlined the narrow translation scheme with several examples of each operation class of a typical RISC-like ISA. Table 2.7 summarises the size of the semantically equivalent set of narrow computations for the selected 64-bit operation classes.

Overall, the narrow ISA mostly consists of the traditional RISC-like 16-bit operations with few additional opcodes e.g., shlc, shrc (as shown in the *Additional Related Opcodes* column in Table 2.7). Many ISAs are already equipped with byte and word size operations like load byte, add halfword etc in x86. As another example, ADC (Add with Carry) opcode already exists in the x86 ISA.

The translation scheme (to map 64-bit programs to the narrow ISA programs) implemented in this thesis applies several simple heuristics to avoid the generation of the worst-case translations. Our implementation of narrow translator abides by the x86 operation sizes (opsize) and generates less than the worst-case translation (e.g. see shr/shl; and also compare rows 1 and 2 with row 3 of Table 2.7) whilst following the *x86 merging rules*[1] according to the opsize.

[1] briefly mentioned in [62]

Table 2.7: Narrow translator conversion ratios – Size of semantically equivalent sets of different type of 64-bit opcodes

Wide Operation (64-bit)	**Number of (16-bit) Narrow Operations in the Semantically Equivalent set for a specified size of a wide x86 (64-bit) operation (size)**			**Additional Related Opcodes**
	opsize = 8/16-bit	opsize = 32-bit	opsize = 64-bit	
add / sub when source register is different from destination register	4	4	4	addc/subc
mask / xor / and	4	4	4	maskc
add / sub / logical when source register is same as destination register	1	2	4	
br	5	5	5	
ld	8 or 4[a]	8 or 4[a]	8 or 4[a]	
st	5 or 1[a]	6 or 2[a]	8 or 4[a]	
shr / shl	4	6	11	shrc, shlc, shrext, shlext
div / mul / fp / sse[b]	1	1	1	
[a] depending on whether address needs to be generated or not. [b] we assume these operations go to special FUs like vector units etc (non narrow datapath).				

2.2.3 Preliminary Evaluations

Impact of the narrow ISA. A realistic code-translator (the narrow translator) for the narrow bitwidth architecture as outlined in Section 2.2, that decomposes 64-bit instructions into an equivalent set of 16-bit computations, results in ∼3.9x increase in the number of computations *dynamically* executed (see Figure 2.2). The execution time in cycles increases by a magnitude of ∼2.2x in an in-order, 16-bit wide datapath, 4-issue width processor with respect to its 64-bit counterpart[1]. The precise simulator configurations used for wide and narrow processors are detailed later in Section 3.1.

[1]Although 16-bit design can potentially operate at a higher frequency, we have conservatively assumed the same frequency for both

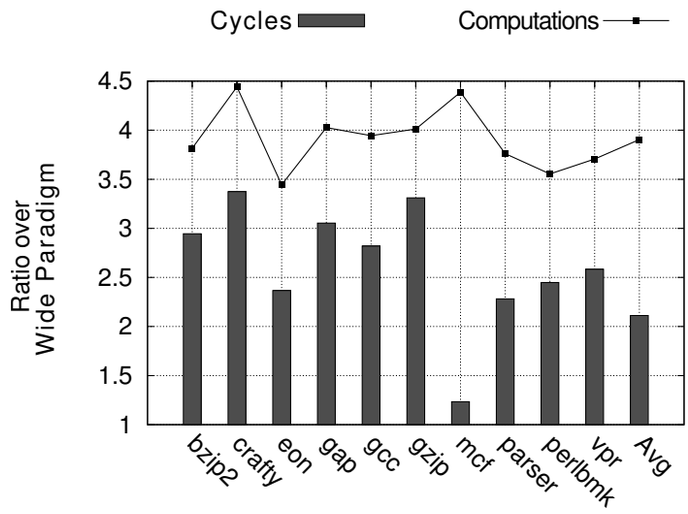

Figure 2.2: Impact of the narrow ISA : The figure shows the dynamic code size (computations) and the execution time (cycles) impact by comparing narrow (16-bit) vs. wide (64-bit) paradigm using the narrow translation scheme outlined in Section 2.2

Impact on the Static Code Size. A realistic estimate of static code size of the narrow ISA programs requires precise specification of the encoding of the narrow ISA instructions. Certain classes (like arithmetic, logical operations) of narrow ISA computations can potentially be encoded in lesser bits than their 64-bit counterpart because the size of the immediate field is reduced to 16-bits rather than 64-bits. However, a narrow ISA instruction will need an additional two bits (per register name) for specifying each register name, as the number of registers in the narrow ISA are 4x that of its 64-bit counterpart.

However, an insight into the overall impact on the static code size of the narrow ISA programs may be important, especially in embedded systems' domain. In absence of precise instruction encodings, Figure 2.3 presents the first approximation of the impact of the narrow ISA on the static code size. For each program, it shows the ratio of the absolute number of static narrow ISA instructions to the absolute number of static 64-bit computations. On an average, the static narrow ISA programs lay out about 3.9 times more instructions than the static 64-bit programs. Further evaluations of the impact on the static code size due to the proposed optimizations in this thesis are provided in Chapter 6.

Dynamic Opcodes Distribution. For each benchmark, Figure 2.4 illustrates the breakdown of the different categories of the committed operations in the 64-bit program (shown in the *BENCHMARK*-wide bar) and the narrow ISA program (shown in the *BENCHMARK*-narrow bar), both for the same 200 million x86 user instruction commits. The classification includes

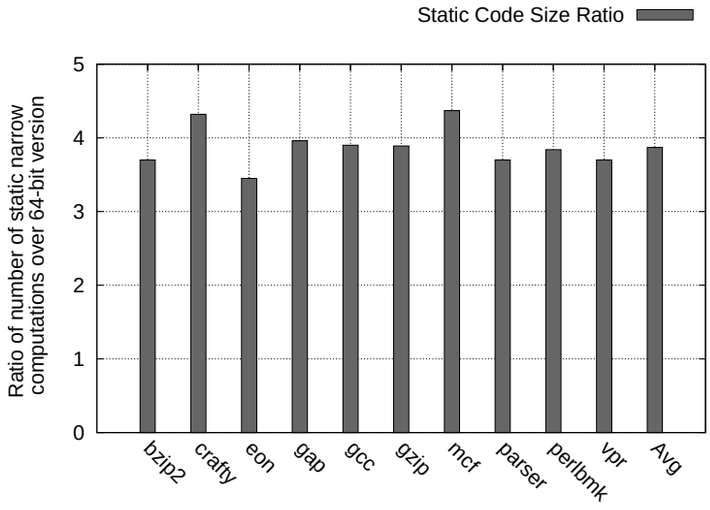

Figure 2.3: Impact on the static code size

– (i) *logic* operations : logical operations and move operations, (ii) *addsub* operations : addition operations and subtract operations, (iii) *addr-gen* operations : add / sub operations which update the *inaddr* register (refer to Table 2.6), (iv) *branch* operations : conditional and unconditional branch operations, (v) *memory* operations : load and store operations, (vi) *br-addr-gen* operations : mov operations to update the *inrip* register (refer to Table 2.5 for translation of branch opcodes), (vii) *shift* operations, (viii) *fp-n-mul* operations : floating-point and multiply operations, and (ix) *others* : rest of the infrequent operations like bitscan, sel etc.

The following can be observed from Figure 2.4 :

- The average (dynamic) code size ratios for *addsub* operations, and *memory* operations are 2.84, and 2.81 respectively. This shows that the narrow translator does not always generate the worst-case translation of four operations. It abides by the x86 sizes if applicable and generates the minimal required operations per 64-bit operation.

- The average code size ratio for *shift* operations is 5.12. The shift class of operations include simple shift by small offset, complex shift, rotate left and rotate right operations. Shift operations introduce more overhead in the narrow ISA as they require more narrow operations per wide operation than any other opcode. However, the dynamic weight of shift operations is comparatively lower.

- Narrow ISA has more *addr-gen* (address generation) operations. The addr-gen operations generate an address which is eventually used by a memory operation. Such operations may exist at the x86 ISA level (like updates to stack pointer or memory operations

from absolute addresses) or originate from *some* specific memory operations : those memory operations where the immediate offset to the memory address cannot be encoded in 16-bits (refer to Table 2.6 for more details). On an average, the 16-bit version of the program has about 4.9x addr-gen operations than the 64-bit version.

- Lastly, Figure 2.4 also shows the dynamic weight of the *br-addr-gen* (branch address generation) operations. Note that these operations do not exist in the 64-bit counterpart. These operations exist only in the narrow ISA. These operations are of the type – mov $inrip_x$ = Imm_x (as shown in Section 2.2.2 for translation of branch opcodes) and are responsible for updating the taken branch address in an internal 64-bit instruction-pointer register (namely, *inrip*).

Conclusions. The implemented narrow translator scheme uses several heuristics to ensure that the narrow code-translations offer a reasonable baseline for further analysis and proposals. The negative impact of the narrow ISA remains large and calls for investigation for further techniques to assuage it.

2.3 Role of the compiler

The analysis presented in Section 2.1.2 suggests that the area and power savings of the narrow bitwidth architecture can be significant. Further, on the software front, the narrow ISA not only provides an ideal platform to implicitly support short-precision computations, but also maintains compatibility with wide computations. These benefits, of course, do not come without a price since this approach introduces the risk of losing out on performance. Using the translation scheme outlined in Section 2.2.2, it has been observed that there is large penalty both in terms of the dynamic code size (about 3.9x) and execution time in cycles (2.2x) when mapping 64-bit applications to the narrow processor.

One could possibly envisage a hardware-based aggressive code optimizer to address this issue of performance degradation. However, it will be limited in scope and, above all, hard to implement, as it may require non-trivial data-flow analyses and book-keeping to improve performance. Hence, it defies the focus of this thesis which is to achieve a simple hardware execution core.

On the other hand, a software-based solution may allow harnessing the power of compiler optimizations. The compiler can not only *translate* but also aggressively *optimize* to reduce the negative impacts of the narrow ISA. Moreover, the narrow ISA presents itself as a completely new play-doh for compiler optimizations – the narrow ISA computation stream is inherently

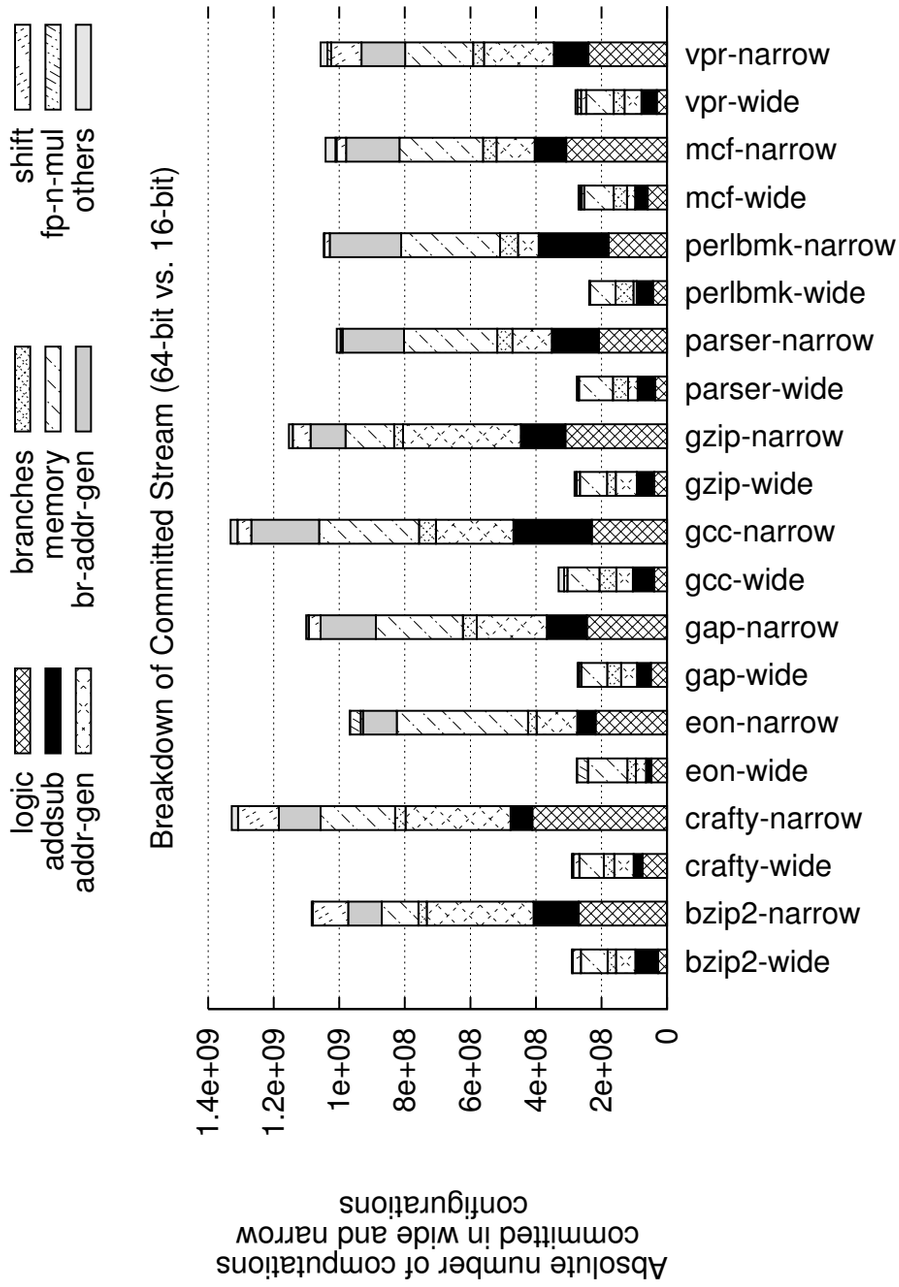

Figure 2.4: Breakdown of the narrow and wide committed stream into operation classes - The distribution of all the committed computations into operation classes to provide more details on the behavior of the implemented narrow translator Scheme

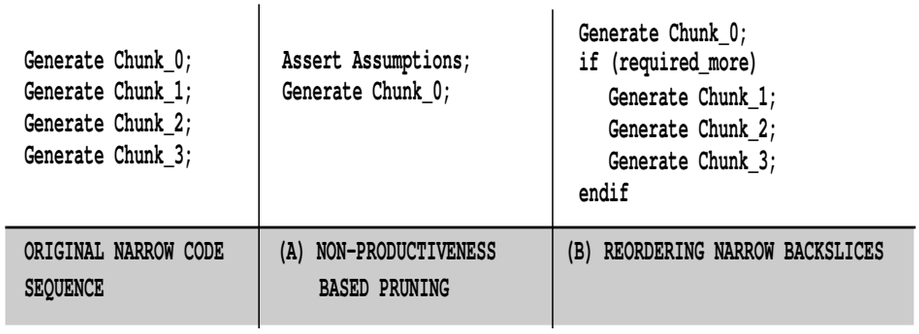

Figure 2.5: Overview of the two main compiler optimization philosophies adopted in the thesis

more parallel; it has more computations of finer granularity. Lastly, a compiler can apply optimizations across a larger scope of instructions.

Thus, this thesis focuses on compiler optimizations targeting the task of how to compile a 64-bit program to a 16-bit machine in order to alleviate the negative performance impact. More specifically, this thesis investigates on code optimization techniques with a perspective of *Minimum Required Computations*. Given a program, the notion of *minimum required computations* (henceforth, MRC) aims to infer the minimum set of narrow ISA computations which are required to generate the *same (correct) output* as the original 64-bit wide program. Needless to say, achieving *perfect* MRC is an intrinsically ambitious goal in that it requires oracle predictions of program behavior. This is because it may not be possible to know whether a computation is required or not without actually performing the computation.

This thesis uses two main profile-guided heuristics to approximate the notion of MRC – one of them prunes useless computations (*Non-productiveness based Pruning* in Section 2.3.1), while the other reorders narrow computations (*Reordering Narrow Backslices* in Section 2.3.2). Figure 2.5 shows a diagrammatic overview of these two main philosophies with dummy sample code sequences. Both the set of heuristics are profile-based and are designed with a strong focus to minimize the dynamic code footprint of the narrow ISA. However, they differ in the degree of speculation.

2.3.1 Non-productiveness based Pruning Techniques

The heuristic of *Non-productiveness* is an approximation of the notion of MRC. Informally, if a computation does not alter the destination storage location, it is non-productive (useless) and therefore, not necessary to be performed. A formal definition of productiveness together with some preliminary evaluations providing more insight into the motivation for the pruning techniques is provided in the upcoming sections.

In this thesis, we apply the concept of Non-productiveness as a learning-based inference (and hence, it is speculative). The proposed Non-productiveness based pruning techniques perform two main tasks – eliminate such non-productive narrow computations, and embed the static productive slices with checks to make them self-sufficient to detect mis-speculation dynamically. As shown in Figure 2.5, Non-productiveness based pruning techniques generate atomic regions with speculatively pruned computations together with the required assertions-like computations to ensure correctness.

The two main code optimizations based on these principles are Global Productiveness Propagation (described in Chapter 4) and Local Productiveness Pruning (described in Chapter 5). The pruning techniques work at different granularities of the data-flow of the programs. More specifically, we have applied the definition of productiveness at the granularities of a single instruction and functions, while varying the notion of atomic regions from a basic block, superblock and a function.

2.3.2 Reordering Narrow Backslices

Code reordering has been proposed in traditional systems for achieving better resource-utilization, hiding memory latencies, improved cache performance amongst others. The aim of the reordering technique proposed in this thesis, however, is to reduce the dynamic code size. The main ideology is to apply the notion of a *lazy computation model* : the narrow chunks of data are generated only when required, deferred otherwise. One possible way of achieving this (lazy) computational model is by reordering the backslices containing narrow computations such that the *minimal necessary computations* to generate the same (correct) output are performed in the *most-frequent case*; the rest of the computations are performed only when necessary.

In this thesis, we propose and evaluate a particular use case of the broader concept of reordering narrow backslices : the use case of reordering narrow backslices around *conditional branch computations*. We call this transformation the Minimal Branch Computation optimization and is described in Chapter 6. This technique is based on the philosophy of reordering computations such that those computations which are *most-probably sufficient*[1] to generate the correct value(s) of the required flag(s) are placed (and executed) first. The rest are executed only if the former were insufficient (refer to Figure 2.5).

2.3.3 Additional Support for Optimizations

As the proposed optimizations are speculative, it is necessary to devise means to detect mis-speculation and a mechanism of recovery. The former is performed in software by means

[1] profile-based learning

of assertion-like instructions and the latter is performed by the hardware. Hence, following additional hardware support is requisite for the optimizations proposed and evaluated in this thesis :

- **New Opcodes for Assertion** : For the Non-productiveness based pruning techniques, few additional *assertion* opcodes are exploited to detect mis-speculation in the dynamic narrow code stream. The assertion opcodes are essentially compare operations which trigger a pipeline fault if the comparison of the indicated entities fails. More specific detail is provided in the respective chapters.

- **Speculative Execution Support** : On an event of misspeculation (i.e. failure of an assertion), we rely on the hardware to ensure that the speculative state does not corrupt the program state by providing some mechanisms to ensure separation of state. In some cases, the speculative regions can be potentially big in size, e.g. a complete function execution may need to be speculative. In these cases, the basic architectural mechanism for speculative execution can be very similar to that of transactional memory [10, 14, 38, 52, 54]. For the rest of the optimizations, where the atomic regions are mainly basic blocks or superblocks, the architectural mechanism for speculative execution can be similar to that used by the Transmeta Crusoe / Efficeon Processors [16, 29]. The latter mainly supports check-pointing the committed register state, and implementation of a store buffer. After a rollback, safe, correct code (generated as explained in Section 2.2) is executed.

  The execution model in wake of speculatively optimized regions is as follows. At the entry of a speculatively optimized region, the register state is checkpointed. Execution continues whilst keeping the speculation contained using the above-mentioned hardware support. On the event of a mis-speculation, a fault is triggered causing a pipeline flush and a correct program state is recovered using the previously checkpointed state. The execution then resumes from the beginning of the region using the correct, non-optimized version of the code. More details on the execution flow are provided in Chapter 3.

  The existence of speculatively optimized regions also impacts the notion of precise exceptions. An exception condition inside a speculative region is treated in a similar manner as a mis-speculation. Hence, after a rollback, safe correct code is executed until the excepting computation is reached.

2.4 Productiveness : Definition and Preliminary Evaluations

We have motivated the case for investigating on compiler optimizations for the narrow bitwidth architectures and have proposed two main compiler optimization philosophies. The rest of the chapters in the thesis formally define, formulate and evaluate the optimization techniques based on these philosophies. However, as the concept of productiveness is used for two main optimization techniques, we define and investigate on specific key aspects of productiveness in this chapter itself.

2.4.1 Background Definitions

This section describes the terminology used in this document.

Definition 2.3 (**Optimization Region**). An *optimization region* is the section of code on which an optimization is applied. Practical examples of an optimization region include basic block, superblock, function or any abstract region of code. Any reference to a *region* in the remainder of this thesis will be to an *optimization region* in general.

Definition 2.4 (**Storage Location**). A *storage location* is a register or memory location used to store data. Further, the *value* of a storage location is the *content* of the storage location, and not its *identifier*. The identifier of a storage location of a computation *i* is denoted by S_i.

Definition 2.5 (**Last-writers**). A computation is a *last-writer* to a storage location if it is the *last* instruction that writes into a given storage location in a given control-flow path from entry to exit of the region. Note that, there can be multiple last-writers to a given storage location, if the region has multiple control-flow paths from entry to its exit(s).

Definition 2.6 (**Dead Last-writers**). A last-writer computation is a *dead last-writer* to a storage location if (even though it is the *last* instruction that writes into a given storage location in a given control-flow path from entry to exit of the region) there is always a writer after the exit of the region that overwrites the storage location before it is consumed.

Definition 2.7 (**Input State of a Region**). The *input state* of a region is the set of values of all the last-writers' storage locations (all storage locations written to by the last-writers of the region R) *before* entering the region.

Definition 2.8 (**Output State of a Region**)**.** The *output state* of a region is the set of values of all the last-writers' storage locations (all storage locations written to by the last-writers of the region R) *after the exit* of the region. Both the input and the output state of the region must be understood in context of a single dynamic instance of the region.

Definition 2.9 (**Program Dependence Graph**)**.** It is a connected, directed graph containing information of both data and control dependences of the computations of a region R [18]. Any reference to a *graph* of a region in the remainder of this thesis will be to its program dependence graph. It is often denoted as PDG(R).

Definition 2.10 (**Backslice of a Computation**)**.** Backslice of a computation is the connected chain of computations which consists of computations that potentially affect its output directly or indirectly. This is computed by traversing the PDG(R) backwards from the computation of interest until specific context-based conditions are not violated. For example, the backward traversal of a computation may terminate only at the entry of the PDG(R) is reached, or may terminate only when computations of specific properties are encountered. Both the control and data dependences are traversed.

2.4.2 Defining Productiveness

Consider a region of one or more computations. The region communicates its output(s) via the last-writer storage locations. On a related note, if the region consists of a single instruction, the latter itself is the (only) last-writer.

Definition 2.11 (**Dynamic Non-productive Last-writer**)**.** A dynamic instance of a last-writer of a region, is called *non-productive* if the value of its storage location (S_i) in the *output state* of the region is the same as the value of S_i in the *input state* of the region.

Definition 2.12 (**Dynamic Productive Last-writer**)**.** A dynamic instance of a last-writer of a region, is called *productive* if the value of its storage location (S_i) in the *output state* of the region is different from the value of S_i in the *input state* of the region.

Definition 2.13 (**Static Non-productive Last-writer**)**.** In the static representation of the region, a last-writer is *non-productive* if the value of S_i at the beginning of the region is the

same as the value of S_i in the *input state* of the region *most of the times*. More precisely, it is a profile-based inference based on a configurable threshold of the last-writer's Dynamic Non-productiveness, and hence, speculative.

Definition 2.14 (Static Productive Last-writer). In the static representation of the region, a last-writer is *productive* if the value of S_i at the beginning of the region is different from the value of S_i in the *input state* of the region *most of the times*. More precisely, it is a profile-based inference based on a configurable threshold of the last-writer's Dynamic Productiveness, and hence, speculative.

In the rest of the thesis, unless otherwise stated, we use the word *Non-productiveness* to directly refer to *Static Non-productiveness*; conversely, the word *Productiveness* is used to directly refer to *Static Productiveness*.

2.4.3 Preliminary Evaluations

This section presents some experiments that evaluate the following aspects of the definition of Productiveness :

1. How does the definition of Productiveness compare with the previously proposed methods of defining narrowness ?
2. What can the definition of Productiveness achieve in the best-case ? Here the intention is to gain an understanding by allowing the system to have a *perfect, advance knowledge* of how computations will contribute towards the output state of a region.

Together, these evaluations bring out the benefits and innovation in the definition of Productiveness. The upcoming results presented in this chapter underline the fact that the definition of productiveness is *more aggressive* than any of the previously proposed strategies. Further, the use of run-time knowledge of the observed values offers a notion of a *bound on that achievable by compile-time analyses in the best case*.

2.4.3.1 Productiveness vs. Previous Approaches

The gamut of ways defining narrowness by previous research has been –

1. *User-Specified Data Widths* : derived from the data type of the variables in the high-level program [35, 37, 56],

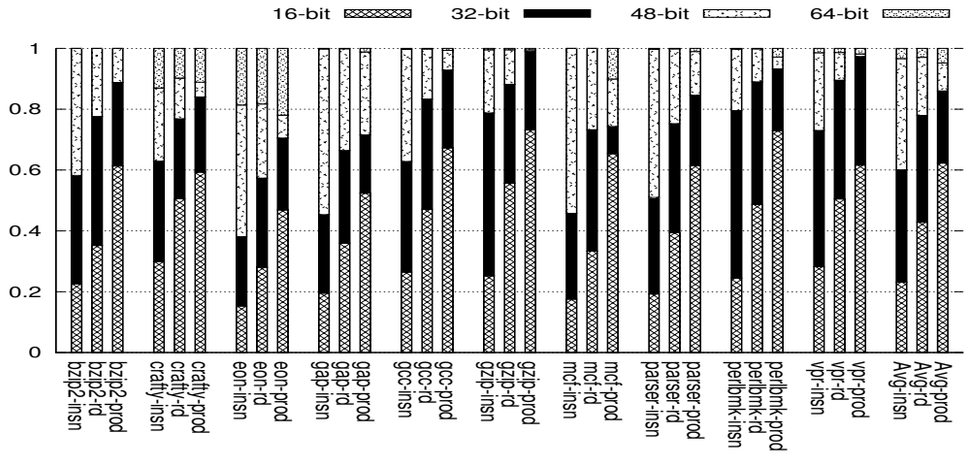

Figure 2.6: Comparing various proposals around the concept of narrowness. The figure shows histogram distribution of the actual datapath required by 64-bit computations for different definitions

2. *Dynamic Data Widths* : calculated as the maximum of the data width / sign-extension of the dynamic operands of each instruction (hence, inferred from the values flowing in the pipeline [4, 8, 36, 47]), and

3. *Usefulness* of data [5, 9, 55].

An interesting comparison of some of these definitions from the perspective of a compiler has been performed previously [53] using a mix of kernels from the Raw benchmark suite and Honeywell ACS suites [57] together with SPEC95 suite. The study compares the definitions of sign-extension and the usefulness (together with some of its variations), with and without backward propagation of the foregoing data properties. The study concludes that – (i) the number of bits that can be saved apart from the high-order prefix ones (i.e. sign-extension) is small, and (ii) back-propagation of operand use information (inferred with either sign-extension or usefulness) is also of limited use. These inferences must be understood in the context of compile-time analyses, instead of purely dynamic definitions like dynamic sign-extension [4, 8, 47].

Hence, it seems fair to compare the new proposed definition of Productiveness against Dynamic Sign-extension. Figure 2.6 compares the following three configurations by considering a single 64-bit computation in isolation –

(i) *BENCHMARK-insn* : These bars show the dynamic data width of the instruction (maximum of the sign-extension data width of all input operands and the output value). This is the same definition as exploited in previous literature [4, 8, 36, 47].

(ii) *BENCHMARK-rd* : These bars show the dynamic data width of only the destination register (more aggressive than *BENCHMARK-insn*).

(iii) *BENCHMARK-prod* : These bars show the dynamic productiveness (Definition 2.12) which measures the change in each destination register.

Only integer instructions (both register and memory instructions) have been included for all the three configurations for the first 200m x86 user commits of each workload (using ref *input* data-set). Our findings corroborate with previous research [17, 53] and highlight the following aspects –

1. The number of narrow computations uncovered by traditional definitions like sign-extension is also significant even in contemporary general-purpose applications. Upto 22% computations need only 16-bits to compute the result (see *Avg-insn* bar), and a 32-bit datapath suffices for yet another 38% of the computations.

2. Our definition of productiveness is more aggressive than previously exploited definitions of narrow computations. Upto 62% of the computations need only 16-bit datapath to compute the result, and a 32-bit datapath suffices for yet another 25% of the computations (see *Avg-prod* bar).

3. This implies that 62% of 64-bit computations, some of which may have been writing non-zero data in higher-significant bits (compare *Avg-rd* bar vs. *Avg-prod* bar), *change* only 16-bits of the destination storage location.

2.4.3.2 Dynamic Non-productiveness : Instruction as Region

Having established that productiveness is a more aggressive definition than previously proposed definitions, we now measure its two further aspects –

1. It is important to observe the effect of productiveness on the smallest possible unit of optimization in our HW/SW ecosystem, i.e., a single narrow computation in isolation. This is essentially similar to *BENCHMARK-prod* bars in Figure 2.6, but measured after integrating the narrow translator (which decomposes 64-bit computations into a set of equivalent 16-bit computations).

2. What is the *temporal locality* of an instruction's productiveness ? The rationale is to measure the run-time variance of the property of productiveness. The degree of variance may affect how the property is to be exploited by compiler analyses.

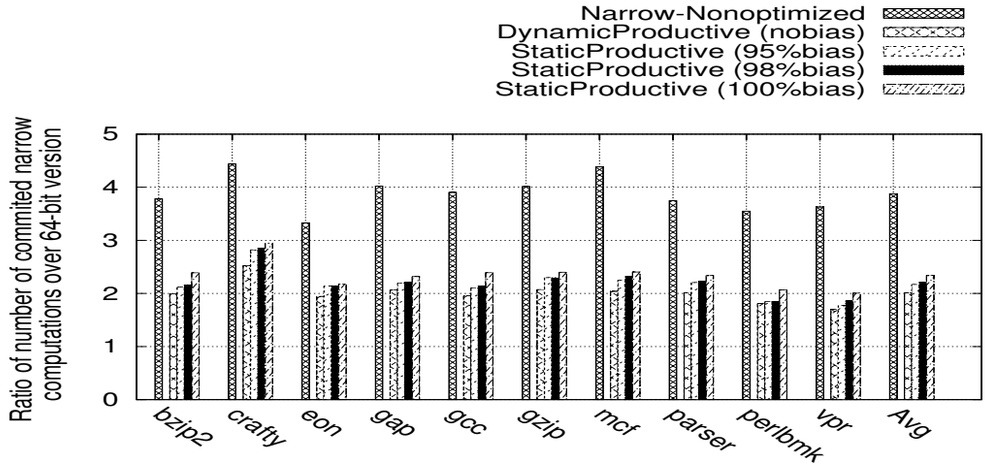

Figure 2.7: Dynamic Productiveness with instruction as a region : best case and sensitivity analysis

Figure 2.7 illustrates how the definition of productiveness can impact the narrow computation stream, assuming advance, perfect knowledge of output data values written by all narrow integer computations. All integer narrow computations are considered for the first 200m commits of x86 user instructions. There are five different configurations shown in 2.7 :

1. *Narrow-Nonoptimized* indicates the ratio of narrow computations over the number of the original 64-bit RISC like operations (same as that shown in Figure 2.2). For example, gzip experiences 4 times more narrow computations than the number of original 64-bit RISC-like operations.

2. *DynamicProductive (nobias)* accounts for all productive narrow computations.

3. *StaticProductive (95%bias)* accounts for the amount of dynamic narrow computations that are productive at least 95% of the total number of times they were executed.

4. *StaticProductive (98%bias)* accounts for the amount of dynamic narrow computations that are productive at least 98% of the total number of times they were executed.

5. Finally, *StaticProductive (100%bias)* accounts the amount of dynamic instructions that are productive all the 100% of the total number of times they were executed.

Comparing different configurations in Figure 2.7, it can be observed that –

1. The dynamic overhead of the narrow ISA can be reduced by around 48% (from 3.9x to 2x as seen in *DynamicNonProductive (nobias)*) by applying the definition of Dynamic Non-productiveness on an individual instruction.

2. The dynamic overhead of the narrow ISA can be reduced by around 44% (from 3.9x to 2.17x as seen in *DynamicNonProductive (95%bias)*) by applying the definition of Dynamic Non-productiveness on an individual instruction.

3. The variance of the data-property of productiveness is admissible.

In Chapter 5, we propose Local Productiveness Pruning, which is an optimization technique aiming to tap this disposable potential.

2.4.3.3 Dynamic Non-productiveness : Function as Region

Using a single instruction as a region (as done in the previous section) allows the compiler to work on a region of very fine granularity. Programs, however, are generally modularized into smaller tasks known as functions. Intermediate state generated by the functions (temporal state in registers) is not available outside the function and is not required to be generated unless it affects the output state of the function.

This forms the rationale of studying the heuristic of productiveness on a whole function as region. Using a function as a region may facilitate a more global data-flow analysis. Informally, the application of the definition of productiveness on a function as a region attempts to achieve the following : if the last-writer of a region is non-productive, those computations contributing directly or indirectly are also non-productive by association, and hence, may be considered useless too. The rest of the computations of the region will be referred to as the *Global Productive Computations* for the upcoming experiment.

To get a notion of the available potential of using function as a region, Figure 2.8 shows the experimental results for a subset of the hot functions [1] from selected benchmarks. Each point on the line graph corresponds to a statistically sampled dynamic execution of the corresponding function. A total sample of 100 dynamic executions of the selected functions is generated from the first 200m user instructions commits of each benchmark. The different points in each line graph indicate :

1. *Narrow-Nonoptimized* indicates the ratio of narrow computations (generated by the narrow translator) over the total number of the original 64-bit RISC-like operations. For example, getRLEpair function in bzip2 experiences 3.7 times more narrow computations than the number of original 64-bit RISC-like operations.

2. *DynGlobalProductive* indicates the ratio of the global productive narrow computations over the number of the original 64-bit RISC like operations. The number of global productive narrow computations is measured by accounting for all productive last-writers in

[1] hot functions are defined in detail in Chapter 3

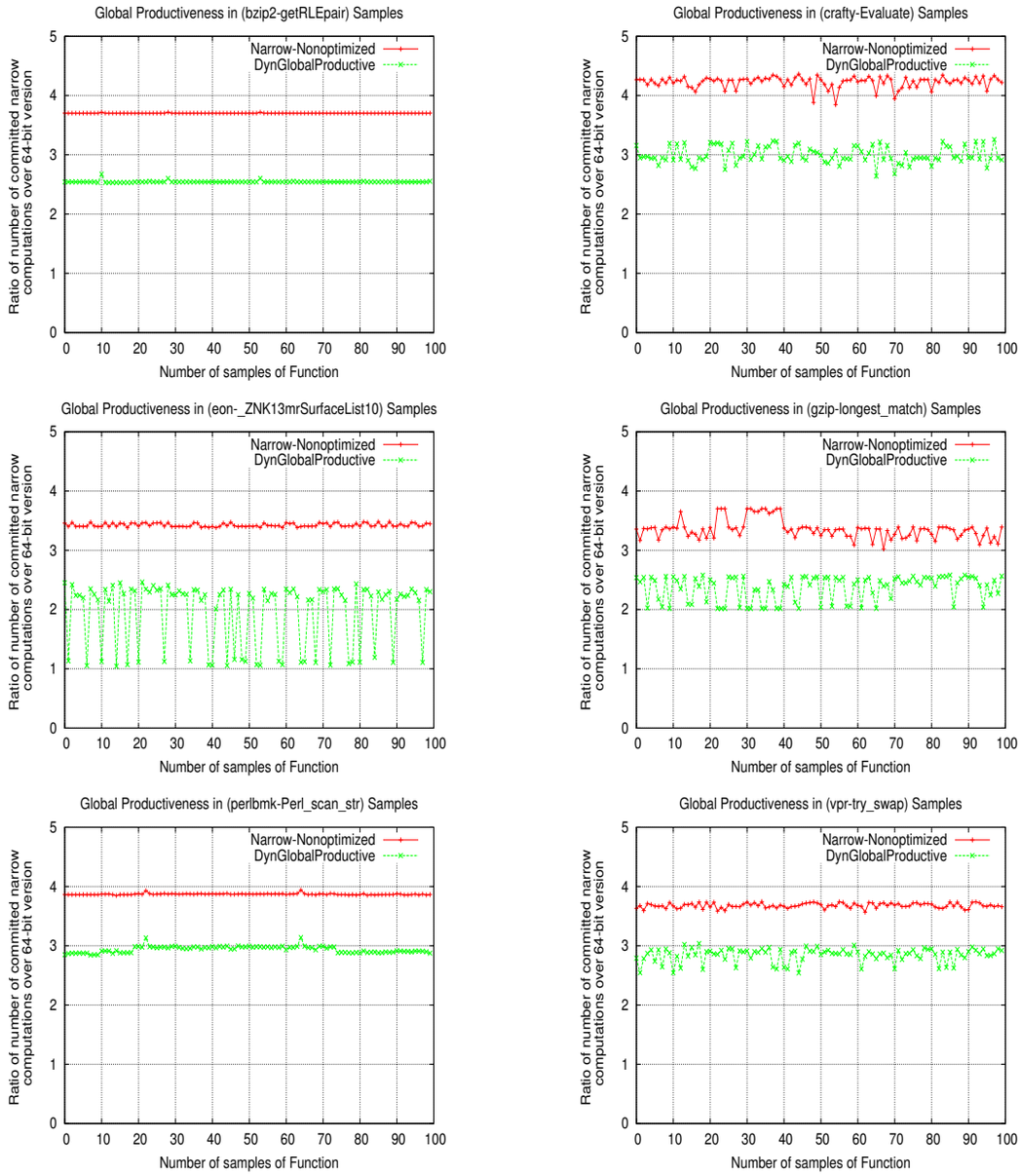

Figure 2.8: Dynamic global productiveness with function as a region : A hundred samples from the dynamic executions of a subset of functions each from SPECint 2000 benchmarks

the respective dynamic execution, together with their backslices. The rest of the computations, that is, the dynamic non-productive last-writers and those computations in their backslices which are not already are a part of the global productive narrow computations are deemed useless. All branches and their backslices are deemed useful.

Applying the concept of productiveness at the granularity of a function can potentially reduce the dynamic code size impact from 3.6x [1] to around 2.7x (25% reduction) on an average. In Chapter 4, we propose Global Productiveness Propagation, which is one possible formulation of an optimization technique capable of capturing this disposable potential.

[1] This average is for the hundred selected dynamic executions of few functions only, and hence, can be different from the observed average of 3.9x in Section 2.2.3

3

Methodology

This chapter details the experimental framework and the evaluation methodology adopted in this thesis. Section 3.1 elaborates on the overall experimental framework, the simulation infrastructure for optimizing and executing narrow ISA applications, and also the benchmarks with their respective training and the input data-sets. The chapter concludes at Section 3.2 with a description of the baseline ecosystem, against which the proposed optimization techniques are compared throughout the rest of the thesis.

3.1 Experimental Framework

3.1.1 Simulator Infrastructure

There are two key components of the simulation infrastructure – the compiler/optimizer framework (named, CodeAnalyzer, described in detail in the following section) and the narrow processor simulator.

3.1.1.1 CodeAnalyzer : The Optimization Framework

The CodeAnalyzer is a tool written in C/C++ which performs the main role of a *trace analyzer* and *optimizer*. Key functionalities of CodeAnalyzer include creating data-flow and control-flow graphs, gathering statistics of program behavior, creating regions like superblocks if required, and profiling and maintaining other book-keeping required for optimizations. Figure 3.1 gives an overview of the important components of the CodeAnalyzer. A brief discussion of some of the important components shown in the same follows next.

Narrow Translator. The CodeAnalyzer supports the translation of a 64-bit RISC-like ISA program into a semantically equivalent 16-bit narrow ISA program. The narrow translator is written as a wrapper utility which uses the PTLsim's parsing and code-cracking support for mapping x86 to 64-bit RISC-like micro operations [62]. The implemented narrow translation scheme to map these 64-bit RISC-like operations to the narrow computations has already been outlined in Section 2.2.2.

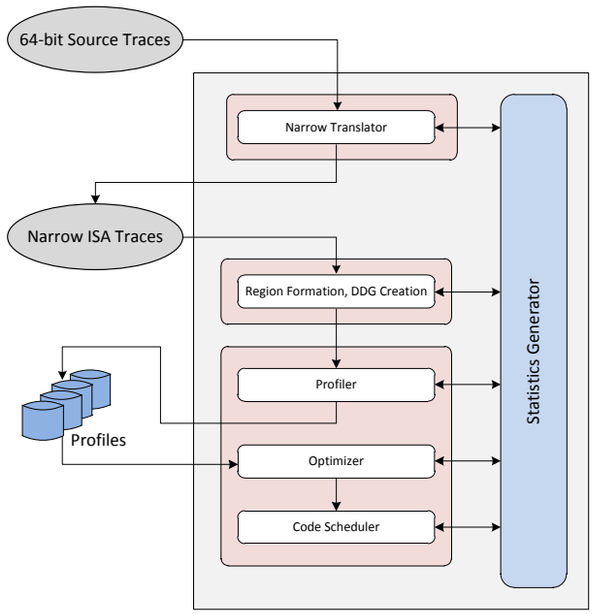

Figure 3.1: CodeAnalyzer : Basic workflow and components - CodeAnalyzer is a tool chain written in C/C++ and performs the main task of profiling and optimization in our evaluation infrastructure

Region Formation. The proposed optimizations have been applied and evaluated on different type of regions, ranging from a single instruction to basic block, superblock and a complete procedure. For selected evaluations, superblocks [24] are built to investigate on how the compiler may benefit from larger atomic regions. The generated superblocks are single-entry, single-exit regions which are created by chaining hot basic blocks and converting the internal branches into asserts (similar to REPLAY [46]). To form a superblock, hot basic blocks are continually added to a new superblock unless the overall commit probability of the region falls below 90% or the next hot basic block to be added is already the start point of another superblock.

Lastly, regarding procedures, we only evaluate non-recursive functions in this thesis (more details in Section 3.1.4). It remains a caveat of our infrastructure and can be improved upon in future work. Hence, all the proposed techniques can be extended to recursive functions as well, given adequate support in the infrastructure.

Modeling Dependences. Register-based data dependences are computed by trace inspection which accounts for control dependences as well. Memory dependences are equally crucial for any data-flow analyses. Often an effective data-flow analysis requires a smart memory dependence analysis using some form of static pointer/alias analysis [11, 32]. CodeAnalyzer, however, models memory dependences conservatively. Both register-based and memory-based true (RAW), anti (WAR) and output (WAW) dependences are modeled. Control dependences are also duly modeled.

Profiler. All the optimization techniques proposed in this thesis are profile-guided. The specific profiles required for each optimization technique are described in the respective chapters.

Optimizer. The optimizer remains an integral part of the CodeAnalyzer. It can optimize both off-line (static optimizer) and on the fly (dynamic optimizer). In case of dynamic optimizer, the CodeAnalyzer's Optimizer works in coalition with the simulator to optimize the applications. In a static optimizer based model, on the other hand, CodeAnalyzer generates the optimized narrow ISA programs off-line, much like a traditional compiler.

The optimizer takes as input the PDG(R) [18], i.e., the program dependence graph of the region R to be optimized and the requisite profile information of the region R to generate the optimized narrow ISA computations stream.

Scheduler. Instruction Scheduling is an important code optimization technique to obtain high performance and better resource utilization on a parallel (pipelined or superscalar) in-order processor. We implement a simple list-scheduling algorithm – *Earliest-start Time Slack* (abbreviated as ETS) based Scheduling. The ETS heuristic based scheduling focuses on two targets – Stall time, and Critical path. The ETS Instruction Scheduling is based on a classical greedy list-scheduling algorithm [21]. ETS scheduling is performed at the basic block level with the target of reducing the overall execution time.

ETS scheduling is evaluated in Chapter 5 after code pruning optimization to illustrate that some of the proposed optimizations not only reduce the dynamic code footprint, but also act as enablers for obtaining further gains from classical compiler optimizations like code scheduling. For this brief study, ETS scheduling is applied at the granularity of a basic block.

Table 3.1: Simulator configurations - Both for the wide (64-bit datapath) and the narrow (16-bit datapath) processors

Parameter	In-Order 16-bit pipeline
L1 DCache	32KB (4-way), 64 Bytes line size, replacement policy : pseudo LRU
L1 ICache	Perfect
L2 Cache	1024KB (16-way), 64 Bytes line size, 8 cycles, replacement policy : pseudo LRU
Memory Latency	250 cycles
Branch Predictor	Combined predictor with Gshare semantics with 64K 2-bit counters, 16 bit global history, and a bimodal predictor of 64K entries with 2-bit counters. Branch Target Buffer (1024 sets, 4-way). RAS of 1024 entries
Frontend, Dispatch, Writeback, Commit Width	4 instructions
Frontend Cycles	5 cycles
Issue Width	4
Functional Units	2 simple integer ALU, 1 complex integer ALU, 2 load/store units, other FP units
Start Point of Simulation	Hand-picked to skip initialization phase

3.1.1.2 The Narrow Processor

For the evaluation of the optimizations, we model an in-order processor of an issue width of up to four instructions per cycle and compare the performance of two versions of the programs in narrow ISA : *with* and *without* the proposed optimizations. As the main focus of the optimizations is dynamic code footprint reduction, we believe that an out-of-order pipeline execution model is not necessary as it may only provide more insight beyond the focus of the thesis.

The cycle-accurate timing model for executing the narrow ISA programs is based on PTLsim [62]. PTLsim is a cycle-accurate simulator for x86 / x86-64 ISA. Porting the simulator for the narrow ISA allows fetching, decoding, renaming, executing and committing individual narrow computations, when the narrow processor based on the narrow bitwidth architecture is simulated.

Table 3.1 shows the values for the most important configuration parameters for all the

Table 3.2: Benchmarks – training and input data-sets, and command line arguments

Name of the program from SPEC2000 int	**Training Data-Set (Profiling)**	**Input Data-Set**
bzip2	input.compressed 8	input.source 1
crafty	< crafty.in > crafty.out	< crafty.in > crafty.out
eon	chair.control.kajiya chair.camera chair.surfaces chair.kajiya.ppm ppm pixels_out.kajiya	chair.control.kajiya chair.camera chair.surfaces chair.kajiya.ppm ppm pixels_out.kajiya
gap	-q -m 128M -l data	-q -m 192M -l data
gcc	-quiet cp-decl.i -o cp-decl.S	-quiet integrate.i -o integrate.S
gzip	input.combined 32	input.source
mcf	input/mcf.in	ref/mcf.in
parser	2.1.dict -batch < ref.in	2.1.dict -batch < ref.in
perlbmk	-I./lib perfect.pl b 3 m 4	-I./lib splitmail.pl 1 5 19 18 1500
vpr	net.in arch.in placed.out routed.out -nodisp -place_only -init_t 5 -exit_t 0.005 -alpha_t 0.9412 -inner_num 2	net.in arch.in placed.out routed.out -nodisp -place_only -init_t 5 -exit_t 0.005 -alpha_t 0.9412 -inner_num 2

evaluations. For some experiments which required the evaluation of the narrow bitwidth architecture against the wide bitwidth architecture (the 64-bit counterpart), same configurations as mentioned in Table 3.1 were used for both. We believe this is justified and does not benefit neither wide nor narrow processor in any configuration because both the simulated pipelines exercise in-order execution.

Further, the *latency of operations* also remains the same in both the simulated wide and narrow bitwidth architectures, in spite of the fact that frequency can be scaled in case of narrow processor [59]. In other words, although 16-bit design can potentially operate at a higher frequency, we have conservatively assumed the same frequency for both.

3.1.2 Benchmarks

The impact of this thesis remains mainly around integer computations. As previously commented in Section 2.2.2, floating-point computations do not tend themselves gracefully to the notion of a 16-bit datapath boundary. Hence, for all the experiments in this thesis, we use

the integer programs from the SPEC2000 suite of benchmarks [15] as the workload. Table 3.2 lists the individual programs from the SPEC2000 integer suite of benchmarks used for the evaluation of the proposed techniques in this thesis. It also shows the command-line arguments together with the data-set used each in the profile-phase (training data-set) and the cycle-accurate evaluation run (*ref* input data-set). The integer applications of the SPEC2000 suite are compiled with *gcc -O3* (for a x86-64 machine with -mfpmath=sse).

The start point of each benchmark is determined by manual inspection of the respective source code. These precise points of interest are communicated to the simulator using special nop like instruction. More information on this is available online in the form of patches at the PTLsim SPECcpu 2000 Benchmark webpage for perusal [20]. In general, they are placed before the start of the main loop to allow the execution of the most representative parts of the program.

3.1.3 Metrics

The two main metrics used for measuring the performance of the proposed optimizations are – *number of cycles* and *number of committed narrow computations*. The *number of cycles* are measured as the total number of cycles taken to commit 200m x86 user instructions (Table 3.4). The *number of committed narrow computations* is measured as the absolute number of narrow computations committed to complete 200m x86 user instructions.

As the optimizations are speculative, we also measure the *failure-rate* of speculative assumptions. The failure-rate is measured as the ratio of the total number of failure events (hence, requiring rollbacks) with the total number of committed atomic regions. Only one failure event per dynamic atomic region is counted, as it is sufficient to trigger a recovery mechanism.

Lastly, a direct effect of having failures / rollbacks in a speculative execution environment is *squashing of useless work*. At relevant points in the thesis, the distribution of the *dynamically committed stream* split between successfully committed optimized regions, successfully committed non-optimized regions and the work squashed due to failures is also appropriately demonstrated.

3.1.4 Hot Regions

In order to bound the profiling and optimization overheads in the software layer, it is important to detect representative portions of the programs. To detect hot regions, we measure the number of committed x86 instructions from each function in the profile-phase of the application. Then, those functions that account for the maximum number of committed x86 instructions are chosen. Table 3.3 lists the detected hot functions. Needless to say, the represented share of the

Table 3.3: Hot regions and expected code coverage (x86 instructions)

Benchmark	Representative Non-Recursive Functions	Expected Coverage (x86 instrs)
bzip2	fullGtU, qSort3, simpleSort	29.0%
crafty	Evaluate, EvaluatePassedPawns, MakeMove, Attacked, UnMakeMove, NextMove, Swap	77.32%
eon	_ZNK13mrSurfaceList10, _ZNK6mrGrid9shadowHit, _ZNK6mrGrid10viewingHit, _ZNK10mrMaterial9shadowHit, _ZNK13mrSurfaceList9shadowHit, _ZN10ggSpectrum3SetEf, _Z25ggRayXZRectangleIntersect, _ZNK19mrCookPixelRenderer	90.28%
gap	strcmp, NewBag, FindIdent, __memchr, Resize	38.35%
gcc	note_stores, propagate_block, memset, memcpy	30.2%
gzip	longest_match, send_bits, ct_tally, updcrc, compress_block	63.12%
mcf	refresh_potential, primal_bea_mpp	99.60%
parser	xalloc, match, power_prune, form_match_list, xfree	41.1%
perlbmk	_IO_getc, Perl_scan_str, Perl_sv_gets, Perl_my_bcopy, _int_malloc, tokeq	85.50%
vpr	try_swap, update_bb, my_irand	99.1%
Avg	**Avg Percentage Optimized**	65.35%

chosen functions will reflect a different share in the committed stream in the cycle-accurate run of the programs, which is shown as the expected coverage in Table 3.3.

Lastly, it is important to note that due to a limitation of our optimizer, we exclude the recursive functions. This impacts the overall coverage achieved by the optimizations evaluated throughout the thesis.

3.2 Baseline Ecosystem

The baseline ecosystem, against which all the proposed optimizations have been evaluated, consists of a narrow processor with a software layer that can map the 64-bit applications to the narrow ISA. This mapping is performed by the narrow translator which implements the

translation scheme outlined in Section 2.2.2. The dynamic code footprint of the narrow ISA is 3.9 times more than that of the wide 64-bit ISA programs. The narrow processor implements an in-order 16-bit datapath pipeline and fetches, decodes narrow ISA computations, and executes and writes-back 16-bit data.

The baseline narrow processor also implements hardware support for speculative execution. Although the *mis-speculation detection* is performed using assertion-like opcodes, the *mis-speculation recovery* is handled via means of hardware. The complexity of the additional hardware support for speculation recovery is related to the size of the speculative regions. When the size of the speculative regions can be potentially big (such as in the Global Productiveness Propagation technique which assumes a whole function execution to be speculative), hardware support similar to that of transactional memory [10, 14, 38, 52, 54] may be required. For the rest of the optimizations (viz. Local Productiveness Propagation and Minimal Branch Computation in Chapter 5 and Chapter 6 respectively), where the atomic regions are mainly basic blocks or superblocks, the architectural mechanism for speculative execution can be similar to that used by the Transmeta Crusoe / Efficeon Processors [16, 29].

In practice, the implemented baseline narrow processor allows a sufficiently large capacity (up to 4096 stores) in the commit record in order to accommodate the different type of atomic regions used for evaluating different optimizations. Subsequent stored values to the same addresses in a speculative atomic region are merged into the same commit record. Hence, except mcf, most of the benchmarks do not require such a large capacity in a commit record.

3.2.1 Overall Execution Model

Broadly, this thesis evaluates the proposed profile-guided optimization techniques in two different models – static or dynamic optimization model. One of the well understood key distinctions between the static and dynamic optimization models is the overhead of profiling and optimization on the application's run-time. The edge of dynamic optimization techniques, however, lies in the accuracy and reach of its predictions of the run-time behavior.

The proposed optimizations have been evaluated as static and / or dynamic optimizations, wherever applicable :

1. Global Productiveness Propagation (Chapter 4) as a dynamic optimization,
2. Local Productiveness Pruning (Chapter 5) as both dynamic and static optimization,
3. Minimal Branch Computation (Chapter 6) as a dynamic optimization.

We comment on why a specific model has been chosen for an optimization in the upcoming respective chapters.

Table 3.4: Dynamic vs. Static optimization model configurations

Parameter	Configuration Value
Static Optimizer	
Committed User Instructions (Profile-phase to perform value profiling with *training* data-set)	200m
Committed User Instructions (Cycle-accurate phase with *ref* input data-set)	200m
Dynamic Optimizer	
Committed User Instructions (Profile-phase to perform value profiling with *ref* input data-set)	200m
Committed User Instructions (Cycle-accurate phase with *ref* input data-set)	200m

In a profile-guided static (or the dynamic) optimizer model, the compiler (or the optimizer) detects, profiles and finally optimizes only the *hot code* of programs. The rest of the program is executed as non-optimized narrow stream, the overheads of which are undoubtedly reflected in the final results illustrated in all the chapters. Profiling and optimizing only the most representative regions of a program is important to reduce the compilation time (or optimization overheads in case of a dynamic optimizer). The lengths of the profile phase and the cycle-accurate phase for evaluation purposes are shown in Table 3.4.

Dynamic Optimizer Model. The basic workflow to evaluate a dynamic optimization is illustrated in Figure 3.2. The execution begins with hot region detection followed by value profiling [6] until the first 200m (after skipping the program initialization phase) with *ref* input data-sets. Next, the optimization is triggered to generate code for the narrow processor.

The advantage of a dynamic optimizer model is that the program is optimized on the fly, hence, the profile-based learning is more precise (as its on the same input as the current run). However, such a model entails higher costs because the run-time of the application must bear the time and space overheads of profiling and optimization. Global Productiveness Propagation being sensitive to the profile information, has been evaluated only as a dynamic optimization.

Static Optimizer Model. The basic workflow to evaluate a static optimization is illustrated in Figure 3.2. First, requisite profiles are gathered by running the programs for the first 200m (after skipping the program initialization phase) with *'training'* data-sets. Next, the compiler optimizes and generates code for the narrow architecture.

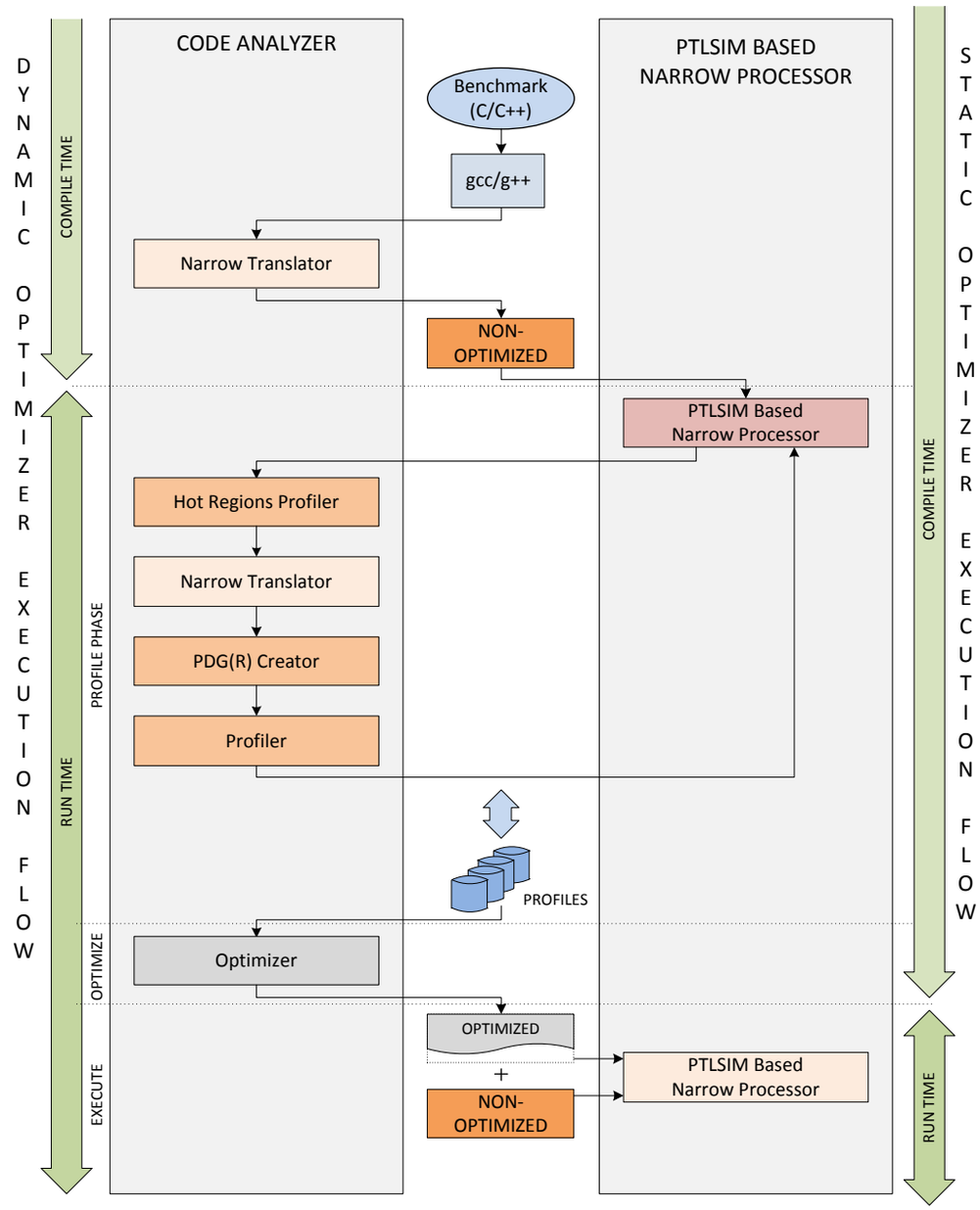

Figure 3.2: Showcasing different execution flows through the developed infrastructure - On the left is the Dynamic Optimizer Execution Flow where the run-time bears the overheads of profiling and optimization. On the right hand side is the Static Optimizer Execution Flow where the compile-time is used for profiling and optimization.

Mis-speculation Recovery. The cycle-accurate model, executes the optimized regions, wherever applicable, and is run for 200m user instruction commits. The rest of the program is executed as non-optimized stream of computations. In case of assertion failures, the instructions committed from the atomic region are squashed, and the hardware support (as outlined previously) restores correct program state. Subsequently, the execution continues from the beginning of the optimized region with translated, safe, and correct code (non-optimized narrow ISA stream).

In a dynamic optimizer model, non-optimized narrow computation stream may be available via a callback to the dynamic optimizer. However, for evaluation purposes, this thesis assumes that the non-optimized narrow computation stream is included in the binary in both the static and the dynamic optimizer model.

4

Global Productiveness Propagation

The two broad techniques proposed in this thesis are – *non-productiveness based code pruning* and *reordering narrow backslices* (introduced in Section 2.3). This chapter describes the first optimization technique known as Global Productiveness Propagation (henceforth, GPP). GPP is a specific technique that exploits the non-productiveness based code pruning strategy to reduce the dynamic code footprint of the narrow ISA programs. It provides the means to apply the principles of minimum required computations on a code region potentially containing complex control-flows.

Programs of utility typically consist of possibly interdependent chains of computations which generate one or more outputs (writes to register or memory storage locations). These computations that write the outputs have been referred to as the last-writers in this thesis (Definition 2.5). Ideally speaking, only those computations that are producers of values consumed by the last-writers directly or indirectly are useful. Thinking further along the lines of the minimum required computations, a consequent question to ask is – how many of these last-writers are themselves non-productive ? In other words, how many of these last-writers are updating the storage locations to the same value as known at the beginning of the region ? If indeed they are non-productive, neither the last-writers nor their backslices are required for generating the same (correct) output.

Now having stated the rationale behind GPP, our experiments suggest that there does exist a non-negligible number of the last-writers which remain non-productive (detailed evaluations are provided in Section 4.4). On an average, about 33% of the 64-bit last-writers have *non-productiveness*[1] in the narrow dimension, i.e. at least one chunk of the generated value remains the same as the input location.

[1] Static Non-productiveness

Hence, the motivation behind GPP is to provide the means to exploit this non-productiveness in a code region potentially containing complex control-flows. This chapter begins by establishing a formal definition of GPP. Next, it describes the overall flow of the optimizer, followed by the design of the optimization (Section 4.2). GPP is then evaluated as a dynamic optimization technique (Section 4.4). Finally, the observed roadblocks preventing GPP to achieve its full potential are evaluated.

4.1 Definition

Informally, GPP is a profile-based optimization technique that *speculatively* prunes the static backslices of *selected* narrow computations : computations that result in the same value (in their respective storage location) as that at the input of the region.

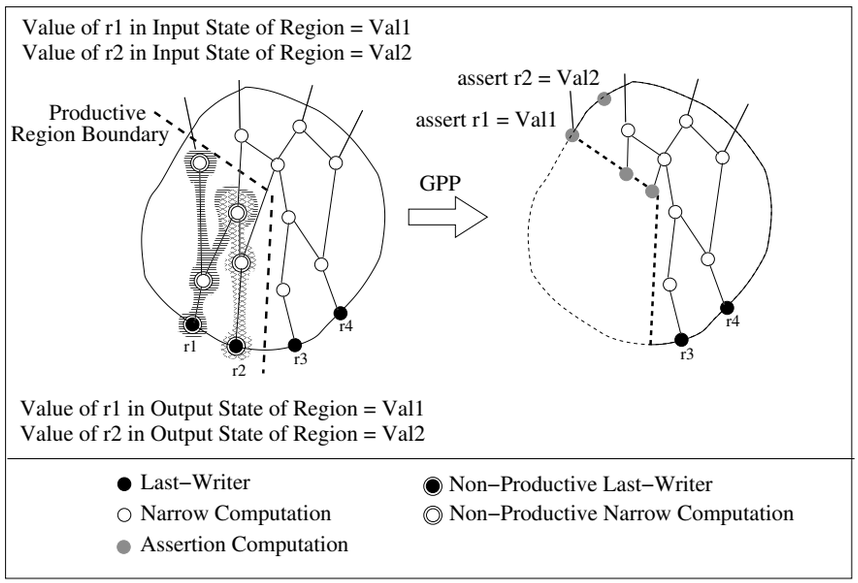

Figure 4.1: The rationale and mechanism of GPP on an abstract region - GPP prunes non-productive last-writers and their backslices and places assertion-like instructions either at the beginning of the region or at the boundary with the productive region to dynamically contain its speculation

Figure 4.1 diagrammatically shows the overall mechanism of GPP in an abstract region. The last-writers of the region are shown in solid black circles. Further assume that the last-writers to storage locations r1 and r2 are non-productive and that values in the storage locations remain Val1 and Val2 respectively. Hence, the GPP transformation prunes these non-productive

last-writers and their backslices (excluding those computations which are already in the backslice of some other productive last-writer). The pruned computations (non-productive) are shown with concentric circles.

Non-productiveness is a profile-based inference, and hence, such a pruning remains speculative. To manage this speculation, GPP reverse-engineers these pruned backslices placing assertion-like instructions in the rest of the code. GPP always generates self-sufficient code which can detect *unassumed* cases by itself. In the event of a successful pruning, the assertion instructions are placed either at the beginning of the region or at the boundary with the productive region. Refer to Figure 4.1 for the assertion instructions marked in gray.

Hence, GPP can be formally defined as follows.

Definition 4.1 (Global Productiveness Propagation). *Global Productiveness Propagation* on a region is an optimization technique that marks for *inclusion* :

- productive last-writers of the region, and
- the backslices of all productive last-writers of the region.

The backslice (Definition 2.10) comprises of all the narrow computations by following both the control and data dependences. Henceforth, the above-mentioned set of computations will be referred to as the *GPP set of productive computations*. On a parallel note, GPP marks for *exclusion* :

- the non-productive last-writers of the region, and
- those computations (in the backslice of the non-productive last-writers) which are not included in the GPP set of productive computations.

4.2 Description

4.2.1 Overview

Figure 4.2 gives an overview of the overall workflow of the GPP optimization with its component passes. As with any profile-guided optimization, there are two main steps – profiling and optimization.

In the first step of the optimization process, representative regions are detected. Next, the optimizer creates its auxiliary structures like data dependence graph and control dependence graphs of these representative narrow ISA program regions. Then, the profiler kicks in and

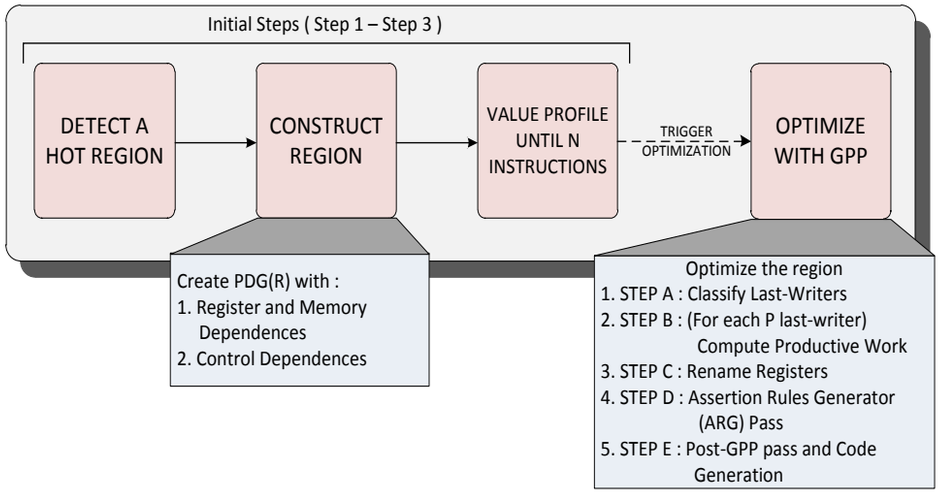

Figure 4.2: Global Productiveness Propagation : An overview with the component passes

collects statistics on the application behavior in context of GPP. This concludes the profiling phase after which the optimization is triggered. Finally, the optimized narrow ISA regions are created and buffered for future execution.

4.2.2 Initial Steps

As the first step of the optimization process, *hot regions* are detected. Hot regions are those regions of code that contribute the maximum number of committed user instructions in the commit trace of a program. The selected routines and their coverage have been outlined in Section 3.1.4.

Next, the Program Dependence Graph [18] (Definition 2.9) of the region is created (henceforth, denoted by PDG(R)). GPP requires cognizance of both register dependences (computed statically) and memory dependences (modeled conservatively). Control dependences are also duly modeled in the program dependence graph of the region. More details of graph creation and modeling data dependences can be revisited in the description of CodeAnalyzer in Section 3.1.1.1.

Subsequently, the profiler kicks in and collects statistics on the application behavior in context of GPP. This *profile phase* is performed until a configurable number[1] of instructions are committed. GPP relies on two different profiles : (i) Productiveness Profiles, and (ii) Value Profiles.

[1] this number is 200m x86 user instructions in our evaluations and is the length of the profile-phase

Productiveness Profiles. To perform GPP on a region, a productiveness profile is required for each last-writer operation of the region. For each (static) last-writer instruction, the productiveness profile indicates how many times the instruction is *dynamically productive* (Definition 2.12) together with the total execution count of the instruction.

Recall that, a dynamic instance of a last-writer computation is dynamically productive if it drives a change to the value of its associated output storage location (Definition 2.11) in the output state of the region (as compared to the input state of the region). Thus, to perform productiveness profiling for the last-writers to register storage locations of a region, the associated values in the registers are buffered at the entry of the region. At the exit of the region, the value in the last-writer's register locations is compared to that at the entry of the region. Using these two set of values, the productiveness profile record (for each static last-writer) counts the following two properties :

(i) the number of times the last-writer is dynamic productive, and

(ii) the total execution count of the last-writer.

Indeed, the cost of productiveness profiles is directly proportional to the dynamic number of executions of the region being profiled and also the static number of narrow last-writer computations in the respective region.

Value Profiles. The concept of value profiling has been introduced and exploited in previous research [6, 7]. GPP requires value profiles of the component edges of the PDG(R). Hence, value profiling for data dependence edges, as used in this thesis, can be understood as an extension of value profiling for instructions as proposed in aforementioned previous research.

The component edges in PDG(R) reflect the data-flow relationships between the narrow computations while accounting for the complex control-flows that may arise in the region being optimized. In other words, a data-flow edge between two computations reflects a dynamic flow of value from the producer to the consumer. An edge is uniquely represented for the purpose of profiling by hashing a tuple of three different values : producer computation PC (program counter), consumer computation PC, and the register name. Value profiling keeps the top ten most-used values for each edge. More precisely, GPP relies on *most-frequent value* profiles of the data-flow edges. In our model, we keep :

(i) the ten most-frequently occurring values of each edge, and

(ii) the total execution count of each edge.

If, for any edge, an overflow case is seen, the least-frequent value is removed in order to profile for the frequency of the new value. This approach may introduce imprecise value profiles for those edges which experience highly fluctuating data values in bursts (each new value may kick out the previous least-frequent value which can otherwise be the most-frequent value). Hence, the cognizance of the total execution count of the edge is useful in determining the overall *execution-count-bias* of the most-frequently occurring value.

Although, we profile all the edges, the overhead of most-frequent value profiling can be greatly reduced by eliminating some candidate edges, e.g., flag edges, and edges which are exclusively in the address-generating backslice etc.

At the end of the profile phase (of say N instructions), the final step of the overall flow, i.e. 'GPP Optimization', is performed. The next section describes the GPP optimization (Figure 4.2) in more detail.

4.2.3 GPP Optimization

The overall steps of the GPP optimization are presented in Algorithm 1. The pre-requisite for GPP is that the hot regions(s) to be optimized be represented in the form of a program dependence graph (PDG(R), containing both the control and data dependences). The PDG(R) can then be used to infer the region's last-writers.

The first step for the GPP optimization is to infer productive vs. non-productive last-writers (Step A in line 1). Then, the algorithm marks all the productive last-writers and their backslices as productive (Step B in lines 2-4). The main task now for GPP is to *safely* prune the non-productive last-writers and their backslices. Hence, Step D forms the critical part of the optimization as reflected in lines 6-23 of Algorithm 1. GPP generates self-sufficient code by embedding assertion-like instructions to detect the mis-speculations dynamically. An assertion is of the form – assert(storage location == value). The assertion opcodes are essentially compare operations which are used to detect mis-speculation in the dynamic narrow code stream. This task of embedding assertions in the program dependence graph is performed by the Assertion Rules Generator Pass (detailed in Section 4.2.4).

Next, a post-GPP pass (detailed in Section 4.2.5) is performed which carries out a simple cost-benefit analysis. It also checks for the corner cases, if any, encountered in the Step D of the optimization. Finally, optimized code with embedded asserts is generated. Next, we elaborate upon these aforementioned steps of the GPP optimization.

Step A : Classify last-writers. As the first step of the optimization, we infer the productive and non-productive last-writers of PDG(*R*). Recall that although the last-writers can be determined statically, the notion of *productiveness* of a last-writer is profile-based. In the current

context, productiveness refers to static productiveness (Definition 2.14) and is inferred using the productiveness profile of the last-writer. In other words, a last-writer which does not change the value of the storage location (as compared to its value in the input state of the region) most of the times is termed as non-productive.

ALGORITHM 1: **GPP_Optimize_Region**

```
   // Step A : Classify last-writers
1  Get productive and non productive last writers of the region;
   // Step B : Compute productive work
2  for each productive last writer do
3      mark backslice productive;
4  end
   // Step C : Rename Registers
5  Rename selected registers;
   // Step D : Assertion Rules Generator Pass.  Bottom-up ARG
6  for each nonproductive last writer do
       // Some nplws may be in the backslice of prod lw
7      if last writer is not productive then
8          if last writer is dead then
               // No Assertion Required
9              Propagate Void Assertion Up;
10         else
11             Get MostFreq Output Value from ValueProfile for the last writer;
12             if mostfreq output value of last-writer is biased then
                   // Initialize assertion to propagate up
13                 assertionA1 ← assert(destination register == mostfreq output value);
14                 if last-writer is not a store operation then
                       // Assert for mostfreq output value at entry
15                     insert assertionA1 at the beginning of the region;
16                 end
17                 Propagate assertions up until either beginning of region or prod region boundary is
                   reached;
18             else
19                 mark backslice productive;
20             end
21         end
22     end
23 end
   // Step E : Post-GPP pass and Code Generation
24 PostProcess Contradictions;
25 Generate Optimized Code;
```

At this juncture, it is beneficial to identify the *dead last-writers* as well. Recall that dead last-writers (Definition 2.6) are those last-writers which are always followed by a write to the storage location after the exit of the region (before a read if any). Thus by definition, dead last-writers, either productive or non-productive, do not impact the final program state. With regard

to GPP, identifying dead last-writers allows simpler handling if they are non-productive (lines 8-10 of Algorithm 1). Note that, detecting dead last-writers is not required for correctness of the algorithm. Figure 4.3 shows the overall classification of the last-writers performed for GPP.

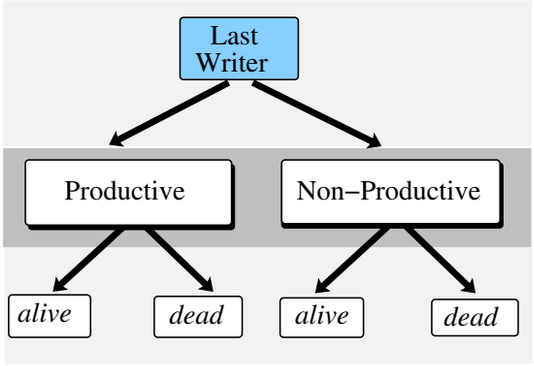

Figure 4.3: Step A – Classify last-writers - First, the last-writers are classified into productive or non-productive based on productiveness profiles. Next, they are classified as dead or alive

Step B : Compute Productive Work. After annotating the PDG(R) with productive and non-productive last-writers, the next step of computing the productive work is performed. This step marks the following as productive computations –

(i) backslices of productive last-writers, and

(ii) complete control-flow [1] (and fp arithmetic with their backslices) in the PDG(R).

Step C : Rename Registers. To ensure correctness, some registers may need to be renamed. This pass renames those intermediate writes in the productive backslices of the region, which write to the same storage location as a pruned non-productive last-writer. In other words, the issue here is that when a non-productive last-writer (and its backslice) is pruned by GPP, no intermediate computation must update the storage location with the same name, since they are not the last-writers.

Step D : Assertion Rules Generator (ARG) Pass. This is the final and critical step of GPP that *reverse-engineers* the pruned backslices and inserts the assertions. This pass is integral to the correctness of GPP because it *enables* the code to detect, by itself, the *assumption failures* at

[1] This set of instructions can be reduced by performing a more accurate analysis and adding control-flow needed just for the productive last-writer slices. This refinement is postponed for future work.

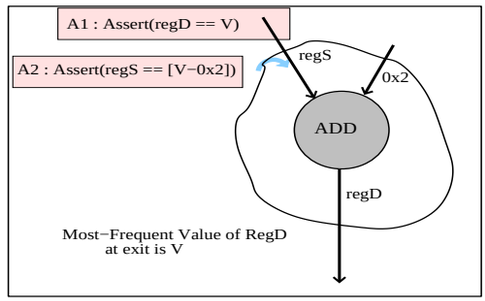

Figure 4.4: Deriving rules for a simple region

run-time. In the ARG pass, the PDG(R) is traversed in a bottom-up manner to derive inferences based on opcodes of computations. The next section gives further details of this pass.

Step E : Post-GPP pass and Code Generation. One of the key aspects of the bottom-up ARG pass is the process of merging assertions by exploiting producer-consumer relationships. This allows GPP to prune more computations than embed assertions in most of the cases. To ensure the cost-effectiveness of GPP further, the algorithm includes a post-processing pass as well. This post-processing pass mainly performs a *cost-benefit* analysis to improve the efficacy of the GPP optimization. For sake of brevity, we only comment on it briefly :

- **Redundant Assertions** have been observed in some cases. This is especially true when the bottom-up propagation places assertions on the live-in edges of the region, which are often repetitive. We perform a dominator and post-dominator analysis to remove redundancy in these cases.

- **Assertion Cost** in our model is not only the number of times it executes, but also the amount of work lost when it fails. Placing assertions while keeping the cost-benefit trade-offs in mind is beyond the scope of this work. However, we do implement some simple heuristics, like, not placing asserts at the beginning of region for those non-productive last-writers which are low on execution count bias with respect to the region (in other words, the unlikely exits of the region).

4.2.4 Assertion Rules Generator (ARG) Pass

The Assertion Rules Generator (ARG) Pass is used to make the GPP set of productive computations self-sufficient to detect the assumption failures. Before elaborating on the step-by-step working of the ARG pass, we first present some basic principles that it applies.

Reverse-Engineering One Computation. Consider the region shown in Figure 4.4 consisting of a lone operation: add *regD = regS, 0x2* (note that the destination register is different from the source register). As the region consists of only one computation, the add operation itself is the last-writer to regD. Let's assume that this computation generates a value of V in most of its dynamic instances. If this computation is non-productive (hence, can be removed speculatively from the region), all dynamic instances must satisfy the following conditions:

- Firstly, value of storage location *regD* in the input state of the region (Definition 2.7) must be V (as shown by assertion A1 in Figure 4.4). This assertion must be placed at the beginning of the region when the last-writer is removed.

- Secondly, value of storage location *regS* before the computation executes must be (V - 0x2) (as represented by assertion A2 in Figure 4.4).

Note that, assertion A2 is derived as assert(*regS* == [V-0x2]) because the operation being pruned is an addition operation. Also note how assertions A1 and A2 are related to each other : assertion A2 on the input storage location regS is what is required to be true for the output storage location to satisfy the assertion A1. As far as both assertion A1 and A2 are satisfied for each dynamic instance of the computation, the speculative removal of the computation is safe and correct. This process of reverse-engineering each computation is achieved by back-propagating the assertion requirements through the associated computation.

Table 4.1 specifies non-exhaustively the associated assertion A1 and assertions A2 for selected templates of addition and mov narrow computations. The template associated with the computation shown in Figure 4.4 is template 2 in Table 4.1. Note that, back-propagation through some operations (e.g. template 3, template 4, template 5 etc.) does not need the most-frequent value profile of the input operands [1]; it is needed only in case of add template 6.

Similarly, one can derive rules for other arithmetic operations. For the sake of brevity, we specify (non-exhaustively) the rules for only add, mov and memory operations in Table 4.1 and Table 4.2 respectively. To conclude, reverse-engineering one computation forms the basis for generating asserts. Evidently, removing one instruction by placing two or three assertion instructions is not efficient. The following sections explain the techniques we have used to further propagate and collapse assertions.

[1] nevertheless, we profile the input operands and do perform a sanity check with the most-frequent value profile in these cases

Table 4.1: Non-exhaustive template-based assertion rules for opcodes – add, mov. Assertion rules for sub operation are similar to add operation

Id	**Instruction Template**	**Assertion A1**	**Assertions A2**
1.	add Ra_0 = Rb_0, $[Imm_0 = 0x0]$	assert (Ra_0 == V)	assert(Rb_0 == V)
2.	add Ra_0 = Rb_0, $[Imm_0 = 0xc]$	assert (Ra_0 == V)	assert(Rb_0 == [V - 0xc])
3.	addc Ra_i = Ra_i, $[Imm_i = 0x0]$	assert (Ra_i == V)	assert (cfin == CV)
4.	addc Ra_i = Rb_i, $[Imm_i = 0x0]$	assert (Ra_i == V)	assert(Rb_i == [V - CV]), assert(cfin == CV)
5.	addc Ra_i = Rb_i, $[Imm_i = 0xc]$	assert (Ra_i == V)	assert(Rb_i == [V - 0xc - CV]), assert(cfin == CV)
6.	addc Ra_i = Rb_i, Rc_i	assert (Ra_i == V)	assert (Rb_i == XX1), assert (Rc_i == XX2), assert(cfin == CV)
7.	mov Ra_i = Rb_i	assert (Ra_i == V)	assert(Rb_i == V)

CV is the carry flag value. cfin indicates the carry-flag input to the computation.
XX1, XX2 are the profile-based most-frequent values of the respective storage location before the execution of the operation. A sanity check (that the profile based asserted values satisfy the operation) is performed for arithmetic operations.

Table 4.2: Assertion rules table for memory operations – load, store

Instructions	**Assertion A1**	**Assertions A2**
ld Ra_i = Mem[$inaddr$]	assert (Ra_i == V)	assert ([ld Mem[$inaddr$]] == V), assert (Ra_i eq V)
st Mem[$inaddr$] = Ra_i	assert (Mem[$inaddr$] == V)	assert ([ld Mem[$inaddr$]] == V)$\otimes$, assert (Ra_i == V)

[$\otimes$] If WAR dependence chain is found, the load based assertion can be removed.

Bottom-Up ARG Pass. The bottom-up ARG pass (lines 6-23 in Algorithm 1) essentially applies the notion of reverse-engineering computations to each non-productive computation in the backslice of the non-productive last-writer in a cohesive manner.

Recall that a non-productive last-writer does not alter the value of its storage location S_i : the value of S_i remains the same[1] in the output state of the region as well as the input state of the region. Hence, to remove the non-productive last-writer from the region, the bottom-up ARG pass essentially performs three basic steps:

1. Initializes assertion A1 for the non-productive last-writer to assert(S_i eq V).

2. Places assertion A1 for non-store based last-writers at the beginning of the region (similar to Assertion A1 in Figure 4.4). This is essentially placing the first test to see whether our profile-based assumption (that the computation is a non-productive last-writer) will be true or not (lines 14-16 of Algorithm 1). Stores are excluded because the storage location is memory-based and the exact location is known only after executing the address-generating backslice.

3. For each non-productive computation C in the backslice of the non-productive last-writer, do the following :

   (i) Perform set union of all the assertions on the successor edges of the computation C. This action, if successful, must always culminate in a single assertion. It is termed as *merging assertions* in the rest of the thesis. For example, in the case of the abstract computation C in Figure 4.5, a set union of assertion A1, assertion A2 is successful if (Val == X). Let's call the merged assertion as A3. In the contrary case of (Val != X), a contradiction is signaled and merging assertion fails (handling of which is explained in the upcoming section).

   (ii) Propagates backward the merged assertion A3 by reverse-engineering the computation C according to the rule-book partially specified in Table 4.1, and Table 4.2. Hence, for this purpose, now assertion A3 is synonymous to assertion A1 for the task of lookup from the rule-book (column 2). This is because assertion A3 represents the condition to be satisfied for computation C to be non-productive dynamically.

   (iii) Perform the foregoing two steps continually for each non-productive narrow computation in the backslice of the non-productive last-writer until one of the following events is seen :

[1] most of the times

- Entry of the region is reached. At this point, the assertion is embedded on the live-in edge (symbolic edge connecting the computation C with the entry node of the region).
- A productive region boundary is reached. At this point, the assertion is embedded on the edge connecting the computation C with the parent productive computation.

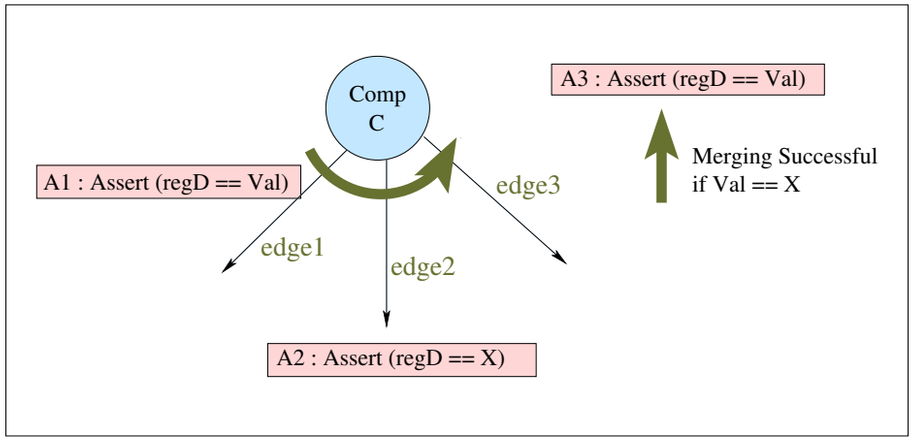

Figure 4.5: Backward propagation of assertions through an abstract computation C. This is termed as merging assertions

4.2.5 The Issue of Contradictory Profiles

This section highlights some of the core issues of the bottom-up Assertion Rules Generator and explains how each of them is handled. First, Figure 4.6 shows the three theoretical degenerate possibilities of a producer-consumer relationship graph –

1. Single Producer Single Consumer (SPSC)
2. Single Producer Multiple Consumers (SPMC)
3. Multiple Producers Single Consumer (MPSC).

Lastly, Multiple Producers Multiple Consumers (MPMC) is a combination of the above three. Next, we define the notion of *contradictory* requirements. Two assertions for a storage location are defined as contradictory when their asserted values are not equal to each other. Note that, in presence of control-flow, existence of such tuples is possible (as we detail next). Such tuples, if left undetected, may lead to a non-zero probability of the two assertions getting executed together, and hence, all their combined executions will always fail.

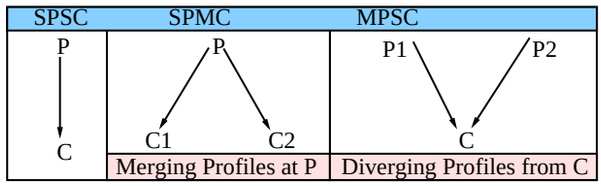

Figure 4.6: Possible types of flows – Single Producer Single Consumer (SPSC), Single Producer Multiple Consumer (SPMC) and Multiple Producer Single Consumer (MPSC)

Single Producer Single Consumer (SPSC). Clearly, SPSC is the simplest case for a bottom-up analysis. As there is a unique path from producer to consumer, we will never have any issue of contradictory value profiles.

Single Producer Multiple Consumers (SPMC). SPMC presents points of *merging* of value profiles in context of the ARG bottom-up traversal. The issue with *merging* profiles at the producer P is that if the most-frequent value of edges P-C_1 and P-C_2 are contradictory, propagating two different assertions up is seemingly difficult. Further dissecting SPMC as shown in Figure 4.7, the following types of data-flows between producer and consumer are possible :

- Type I : Two consumers of a producer lie on mutually exclusive paths such that none of the consumers is a post-dominator.
- Type II : Only one of the consumer is a post-dominator.
- Type III : Both the consumers are post-dominators.

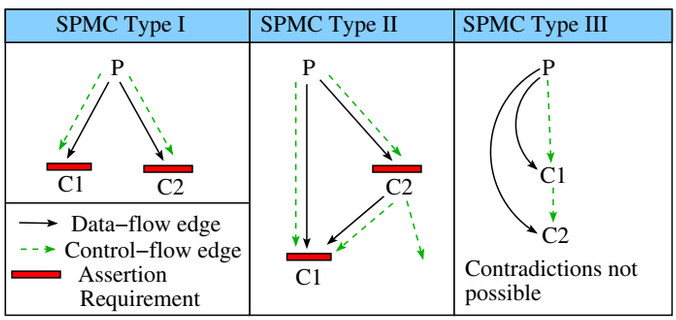

Figure 4.7: SPMC Classification

Both SPMC Type I and SPMC Type II may lead to failure in merging assertions, if contradictory profiles are seen. In these scenarios, the ARG pass handles the points of contradictory profiles *conservatively* by marking the point of contradiction as useful (marking P and

its backslice productive) and embedding the assertions above the consumers. More aggressive techniques like duplication of data-flow are indeed possible, but have not been explored in this thesis. Lastly, in the SPMC Type III data-flow, contradictions are not possible, because both the consumers C1 and C2 are post-dominators of producer P, and hence will consume the same value always.

Axioms. Following must be true in the context of the ARG pass and the existence of SPMC type data-flows. Ensuring these below-mentioned axioms in the functional implementation (implemented in line 17 and 24 of the Algorithm 1) of the ARG pass ensures its *correctness*.

- Amongst the possible SPMC flows, only SPMC Type I and SPMC Type II data-flows may have contradictory profile requirements. If no events of these types are seen, we must never encounter any contradictory profiles when *merging* assertions at any point.
- Two consumer edges in SPMC Type III data-flow must never have contradictory profile requirements at P. This is because both C_1 and C_2 will always execute if producer P executes. Hence, the most-frequent value profile of the edges P-C_1 and P-C_2 must be the same.

Multiple Producers Single Consumer (MPSC). Finally, MPSC presents points of *diverging* of value profiles in context of the ARG bottom-up traversal. An easy bottom-up propagation of assertion required to prune the consumer C in presence of *diverging* profiles from different sources is also difficult. To elaborate further, **MPSC** scenario may further be dissected into different types shown diagrammatically in Figure 4.8 :

- Type I : Consumer Node is a One-Input-Register-Operands Operation
- Type II : Consumer Node is a Two-Input-Register-Operands Operation

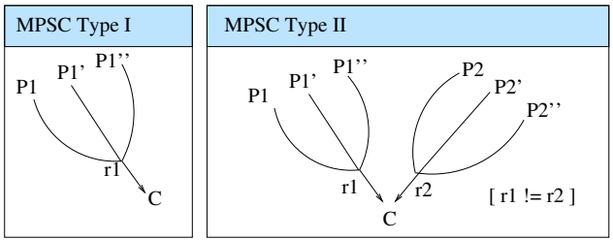

Figure 4.8: MPSC Classification

Note that, the different data-flow edges P_1-C, P_1'-C, P_1''-C *may arise* (see Figure 4.8 – MPSC Type I) if the consumer C is reachable from different control-flow paths. If the most-frequent profiles for the same input register, e.g., register r1 (in MPSC Type I) has three different profiles on the different paths – P_1-C, P_1'-C, P_1''-C, the profiles are termed as *diverging* in nature. In case of MPSC Type I, this is not an issue, and the ARG pass can safely continue in a bottom-up manner.

Similarly, due to the presence of complex control-flow in regions, there may exist scenarios like MPSC Type II shown in Figure 4.8. If the profiles of register r1 and r2 are diverging in nature, pruning consumer C is difficult without adding support to assert for *combination* of values : any combination of the control-flows P-Q, where P $\in$ [P_1, P_1', P_1''] and Q $\in$ [Q_2, Q_2', Q_2''] may occur at run-time. The ARG pass handles this scenario *conservatively* by marking this point of bifurcation (which is consumer C) as useful (hence, marking all its predecessors and their backslices productive) and embedding the required assertion below the consumer C.

4.2.6 Cost Analysis

Optimizing a region by GPP involves the sequential steps as shown in Figure 4.2 previously. The cost of optimizing code regions by GPP is the sum total of the cost of applying each of these individual steps. The cost of classifying last-writers (Step A) is proportional to the static number of last-writers of the region to be optimized. Computing productive work (Step B), in limit, is bounded only by the total number static computations in the region to be optimized (same as the total number of nodes in the PDG(R) denoted by '*n*'). Next, the cost of the ARG pass (Step D) is expected to be proportional to *nplw * (nnp + enp)*, where '*nplw*' denotes the static number of non-productive last-writer computations, and '*nnp*' denote the total number of nodes not marked productive by Step B and '*enp*' denotes the number of edges in the same portion of the PDG(R). Hence, as the ARG pass is the most expensive, the cost of performing GPP is expected to be bounded by *n * (n + e)* in the worst case.

4.3 Example : Walk-through

To put the overall optimization in context, let us walk through a small region consisting of a load-compute-store chain as shown in Figure 4.9. The only last-writers of the region are marked with concentric circles (the stores). Notice that in this region the writes to the higher three chunks store the same value to memory as that at the beginning of the region, when there is no carry generated from $Chunk_0$ to $Chunk_1$ (between the add operations). Hence, as **Step A**, we detect the productive last-writers (mem.0) and the non-productive last-writers (mem.3, mem.2, mem.1) based on the profile. Clearly, the productive work is the backslice of

mem.0 (**Step B**). **Step C** of renaming is not required as the non-productive last-writers update a memory-based storage location.

However, for some dynamic instances, we *may* need to store the higher chunk(s) as well. To be able to detect those cases at run-time, we perform **Step D** (the ARG pass) to insert appropriate assertions. Hence, beginning the bottom-up pass from a non-productive last-writer (e.g., mem.1), Table 4.1 gives the requisite two assertions – (a) load from memory and assert that value is V : A1 = *assert ([ld inaddr 0x2] == V)*, and (b) assert that the value being stored to memory is V : A2 = *assert (rax_1 == V)*. Assertion A1 propagates further up to encounter a WAR dependence with the ld operation and hence is nullified. A2, when propagated up, encounters the addc operation, which is again non-productive. This addc operation can be removed, and to ensure that A2 is satisfied, an *assert* on the previous operation (i.e. add) that there is no carry flag (refer to add template 3 in Table 4.1) is embedded.

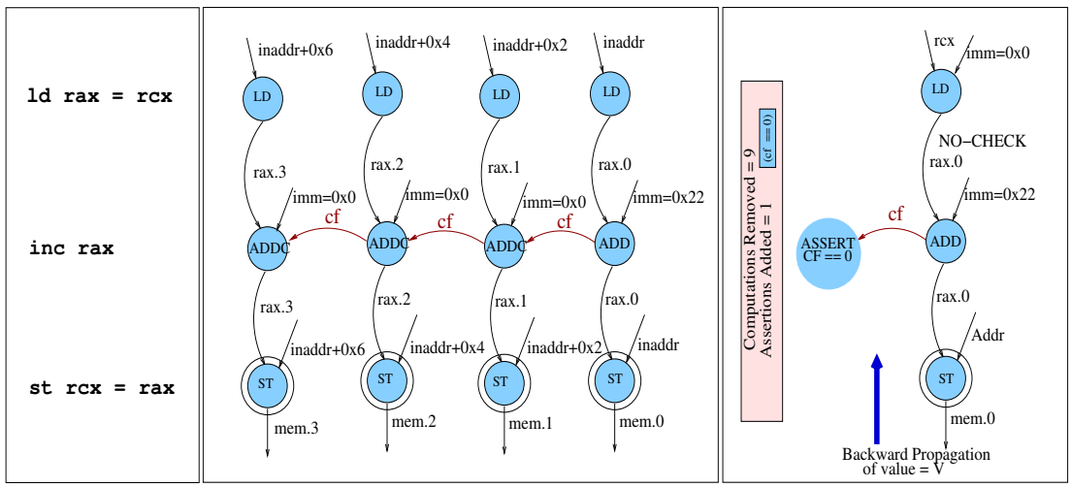

Figure 4.9: GPP at work – (left) Sample code, (right) Computations Removed vs. Assertions Placed

Note that to prune the backslices, cognizance of a WAR memory dependence is exploited. However, this aliasing information may sometimes be easily deduced at the compiler level (especially stack based communication, spill code, callee save and restore memory operations, and absolute address based memory operations).

Hence, in this example, GPP removes 9 operations and inserts 1 assertion, *assert (cf == 0)* after the add operation. Most importantly, we can observe that the computations which are removed by GPP cannot be removed by significance compression [4], UVP or VRS techniques [9] proposed previously. This is because the value of *rax* may as well be 64-bit *significant*([4, 8]) or *useful* ([5, 9]).

Discussion. GPP optimization technique, in the current form, has limited mechanism for capturing dynamic non-productiveness in a region (multiple specialized versions of the region may allow to tap dynamic non-productiveness in the true sense). Further, conceptually a region can be any piece of program code; and the notion of *productiveness* can be applied at other granularities too. However, in this thesis, we evaluate GPP considering functions as regions. Further, we consider only non-recursive functions (with their child functions inlined). Functions are a logical choice given the two-fold goals of achieving larger size of backslices and also minimal number of productive last-writers. Lastly, the dependence chain in Figure 4.9 has been chosen for sake of understanding. One of the realistic examples exploiting the data-property of *productiveness* is a chain of stack-address-increment and stack-address-decrement operations (as the higher chunks of the address will be *significant* and *useful* but not productive).

4.4 Evaluation

In this section, the performance of the proposed optimization is evaluated. First, we briefly revisit the experimental framework (already described in Chapter 3) in the context of GPP to aid the understanding of the upcoming evaluations. The performance of GPP is compared against the baseline narrow processor, which is an in-order 16-bit integer datapath processor combined with a realistic narrow translator.

Different aspects of GPP have been highlighted – reduction in computations, reduction in the number of cycles consumed to accomplish the same amount of work, assertion failure statistics and finally, classification of the dynamically committed narrow stream to understand the code coverage of the optimized regions. Lastly, in Section 4.4.4 we compare GPP's achieved performance against the disposable potential projected in a perfect environment. This study helps identifying the observed roadblocks, which moving forward, can aid in making GPP more effective.

4.4.1 Experimental Framework

GPP has been evaluated as a dynamic optimization, because preliminary evaluations indicate that it is sensitive to noise in profiles. Hence, the baseline assumed for these evaluations is a narrow processor with support for dynamic optimizations. Simplistically speaking, the evaluation of GPP *does not account for the overheads of profiling and optimization* of the narrow code stream.

Table 3.1 shows the simulation configurations for evaluating GPP on a narrow processor. We model an in-order processor of an issue width of up to four instructions per cycle and

compare the performance of a 'narrow processor with the GPP optimized code' against that of a 'narrow processor without such an optimization'.

The execution model profiles for first 200m x86 user instructions (after skipping the program initialization phase) and then triggers the GPP optimization. Table 3.3 shows the percentage of the committed stream which is optimized (Expected Coverage) by GPP. Regions have been evaluated with a most-frequent profile bias threshold of 90% and 95% (line 12 of Algorithm 1). This means that computations that are not biased 90% and 95% of their executions to a single value in their profile are considered productive.

The optimized regions are then used in the cycle-accurate processor model for the next 200m x86 user instructions. At the beginning of the execution of the optimized region, the system state is checkpointed. In case of an event of assertion failure in the optimized region, hardware support restores correct program state by copying the checkpointed system state. The execution then resumes with re-translated, safe, correct code (non-optimized stream of narrow ISA computations) of the region (i.e., function in this evaluation).

4.4.2 Productiveness of Last-writers

As previously mentioned in Section 1.4, another profile-based speculative optimization technique proposed in previous literature which is somewhat related to GPP is Value Range Speculation (VRS [9]). VRS is related to GPP as both the techniques aim to exploit narrow computations for reducing dynamic code footprint. The VRS technique *specializes* hot code by using on the dynamic *value range* profiles. It involves a cost-benefit analysis which prioritizes energy-savings expected from the specialization of a certain candidate. VRS basically duplicates the regions of code that are affected by the specialization, and then inserts tests to dynamically select the region that will be executed: either the specialized or the non-specialized one.

Overall, the optimization methods of VRS and GPP are very distinct from each other. Hence, we only compare how the trigger points of the two vary.

Figure 4.10 shows a study of *productiveness* of the dynamic 64-bit last-writer computations and provides a breakdown of the percentage of 64-bit last-writers which have only 'n' lower chunks productive, where n ranges from 0 to 3. 'all-chunks' indicates those operations which need to generate all the required chunks. Recall that the number of required chunks may not necessarily be four in number because the narrow translator does not always generate computations for all 4 chunks as outlined in Table 2.7 due to x86 opsize specification. It can be seen that –

- About 7% last-writers have all chunks as non-productive (see 0-chunk in Figure 4.10).
- 60% have all chunks as productive.

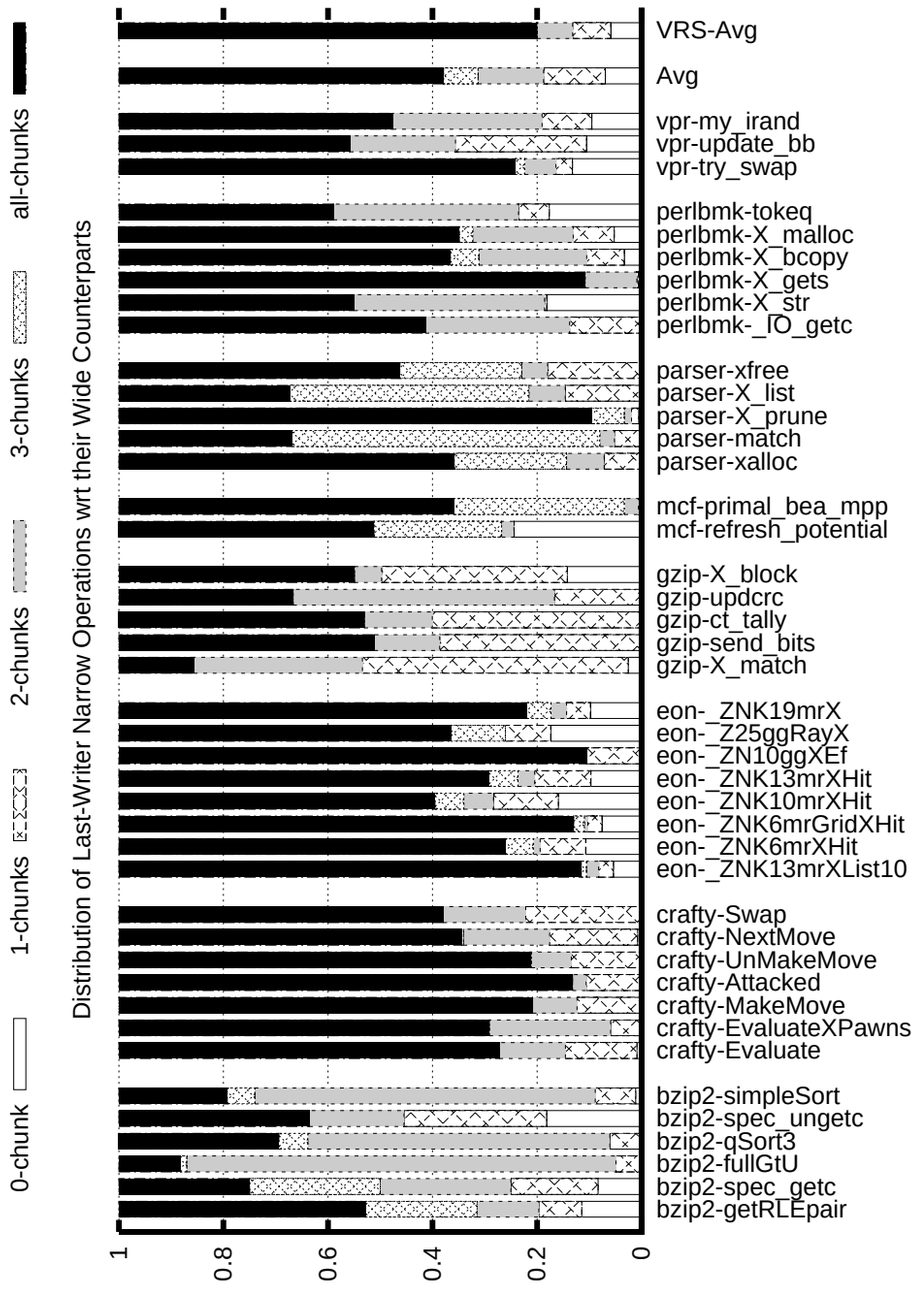

Figure 4.10: Distribution of productiveness of last-writers - Histogram distribution of all dynamic instances of last-writer 64-bit computations to compare GPP trigger points vs. the Value Range Speculation technique's trigger points

- The rest of the last-writers (sum of 1-chunk, 2-chunks and 3-chunks in the histogram), i.e., about 33% have *non-productiveness* in the narrow dimension.

The number of chunks required by VRS to optimize a trigger point A is calculated as the maximum of the number of chunks required to represent the range of values at A, i.e, sign-extended MinValue and MaxValue of A. As the right-most bar in Figure 4.10 illustrates, GPP's definition of *productiveness* is more aggressive and invades farther in the narrow dimension of the last-writers, and hence offers *2x more* trigger points than the VRS.

4.4.3 GPP Evaluation

As stated previously, this section compares the performance of a 'narrow processor with the GPP optimized code' against that of a 'narrow processor without such an optimization'. The performance metrics for GPP include – number of cycles , assertion failure statistics, and number of computations. Recall that one of the configuration parameters of GPP is – most-frequent value bias threshold (refer to line 12 of Algorithm 1). We evaluate two potential candidate values for the same – threshold=90% and threshold=95% to furnish some insights on the sensitivity of GPP to the most-frequent value bias threshold.

Reduction in Number of Cycles. Figure 4.11 shows the overall gains achieved by GPP in terms of reduction in the execution time in cycles, as measured for two different thresholds of 90% and 95%. Further, three different narrow processor configurations with respect to issue width are analyzed : issue widths of 1, 2, and 4. Both the experiments with the thresholds of 90% and 95% account for overheads of assertion failures : on account of an assertion failure, speculatively committed narrow computations are squashed, and correct program state is restored.

On an average, up to 4.5% reduction in the number of cycles for a 1-issue, in-order narrow processor can be achieved. A reduction of 2.7% and 2.1% in the number of cycles can be achieved in a 2-issue, and 4-issue configuration respectively. Mcf stands out with the lowest penalty in execution time in the narrow paradigm (refer to Figure 2.2), and low overall gains because it is mainly memory-bound. Vpr, as one of the model programs, achieves up to 9% reduction in the number of cycles.

Assertion Failures and Variation with different Thresholds. Non-negligible assertion failures exist only for crafty, parser and perlbmk (Figure 4.11). The primary causes of high failure rates in crafty for a threshold of 90% are :

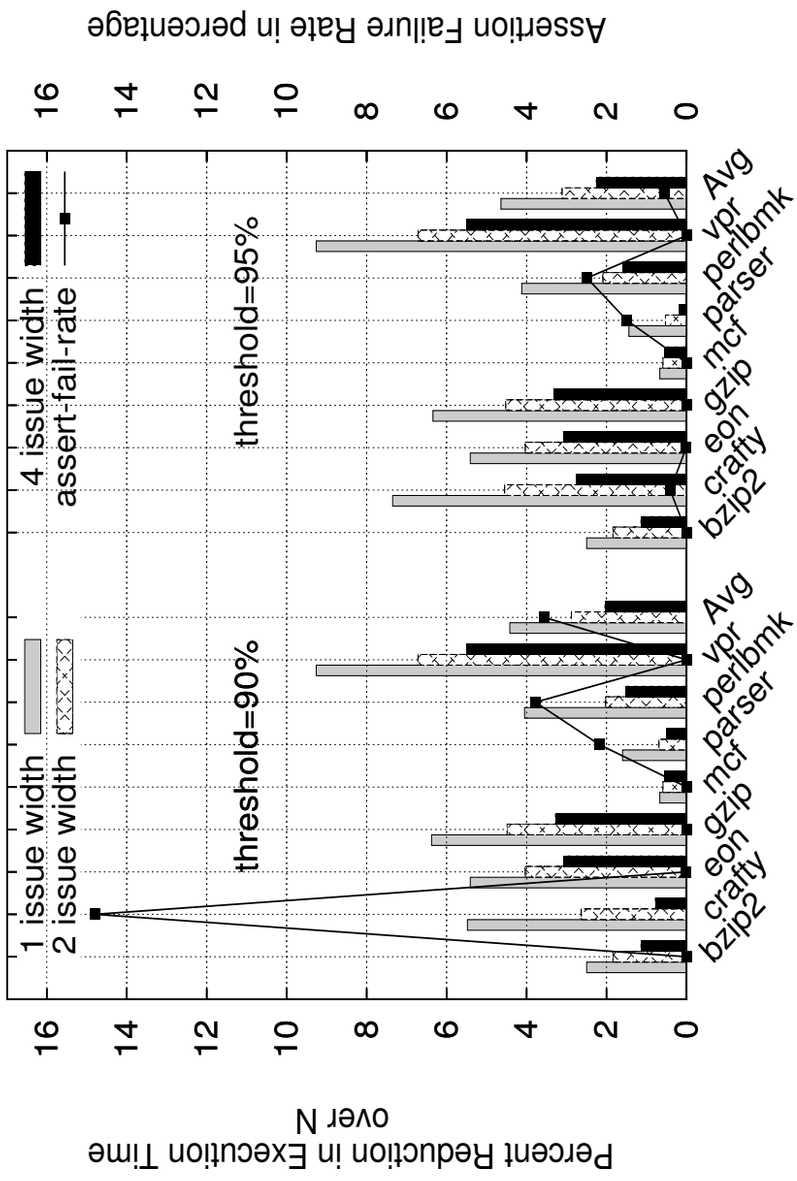

Figure 4.11: Gains achieved by GPP – Reduction in the number of cycles achieved by two configurations of GPP by varying the most-frequent value bias threshold : threshold=90% and threshold=95%

(i) Value profile phase change : GPP has been evaluated as dynamic optimization. This implies that the input data-set is the same for the profile-phase as well as the cycle-accurate phase. However, crafty experiences a phase change with respect to the value profiles. This accounts for some of the failures in the cycle-accurate phase.

(ii) Low most-frequent value bias threshold : A most-frequent value bias threshold of 90% is more speculative than a threshold of 95%.

Low assertion failure rates for bzip2, mcf, gzip, eon etc. indicate that the value profiles exploited to prune computations by GPP and to embed value-based assertions are very predictable. Hence, a most-frequent value bias threshold of 90% suffices. As can be seen, the assertion failure rates for crafty reduce greatly with threshold=95%, and hence, higher benefits are achieved both with respect to the number of cycles and the dynamic number of narrow computations (Figure 4.12).

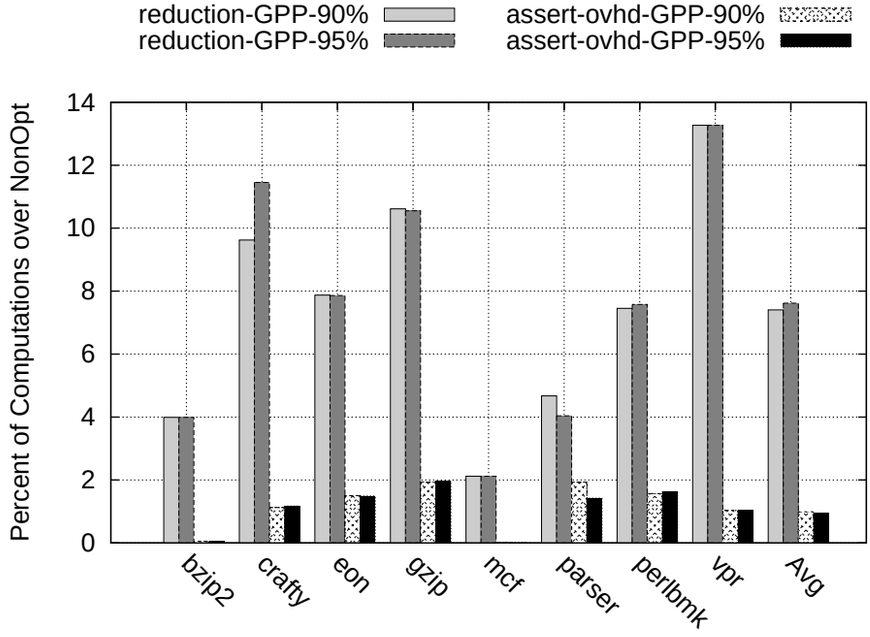

Figure 4.12: Benefits vs. Cost of GPP in terms of number of computations over baseline

Overall Gains : Computations Reduced vs. Overheads. Figure 4.12 shows the overall performance of GPP for both the most-frequent value bias thresholds of 90% and 95%. Two set of bars are shown for each program :

(i) *reduction-GPP* : Percentage of dynamic narrow computations removed by GPP with respect to the non-optimized dynamic narrow stream of computations.

(ii) *assert-ovhd-GPP* : Percentage of dynamic assertion computations added by GPP with respect to the non-optimized dynamic narrow stream of computations.

The assertions embedded by the ARG pass have very low run-time overhead : an average of less than 1% (they are non-zero in all the programs). Overall, 6.6% of the committed stream is reduced by GPP (after accounting for the assertion overheads). Benchmarks with relatively higher coverage (refer to Table 3.3) yield better results (vpr, gzip, crafty) than those with low coverage (bzip2, parser). Although about 90% stream is optimized in eon, it does not perform so well, as it has many fp ops embedded in the regions. It is interesting to note that vpr (with an overall coverage of $\sim$99%) achieves 12% reduction in the operation stream and $\sim$9% reduction in the number of cycles consumed in the configuration with an issue width of 1.

Dynamic Stream Classification. A deeper insight on code coverage achieved with GPP (with most-frequent value bias threshold of 95%) can be gained from Figure 4.13, which shows the breakdown of the committed narrow stream in the following categories –

1. *commit-success* shows the narrow operations from the optimized region when the regions commit successfully,

2. *squashed* shows the narrow operations lost due to assertion failures, including the assertions themselves, and

3. *non-optimized* shows the narrow operations committed from the non-optimized regions. This may be either due to originally untouched regions (as GPP selectively optimizes hot regions only) or due to an assertion failure in an optimized region causing a roll-back followed by execution and commit from safe, non-optimized version of the code.

Note that, overall 60% of the committed stream is optimized across benchmarks. Further, work squashed due to assertion failures is also minimal, indicating that assertion failures mostly occur close to the beginning of the region.

4.4.4 Observed Roadblocks

As previously discussed in Section 2.4.3.3, the disposable potential of GPP in an unrestricted, perfect scenario has been observed to be around 25% in terms of reduction in the number of committed narrow computations. However, the achieved reduction in computations is around

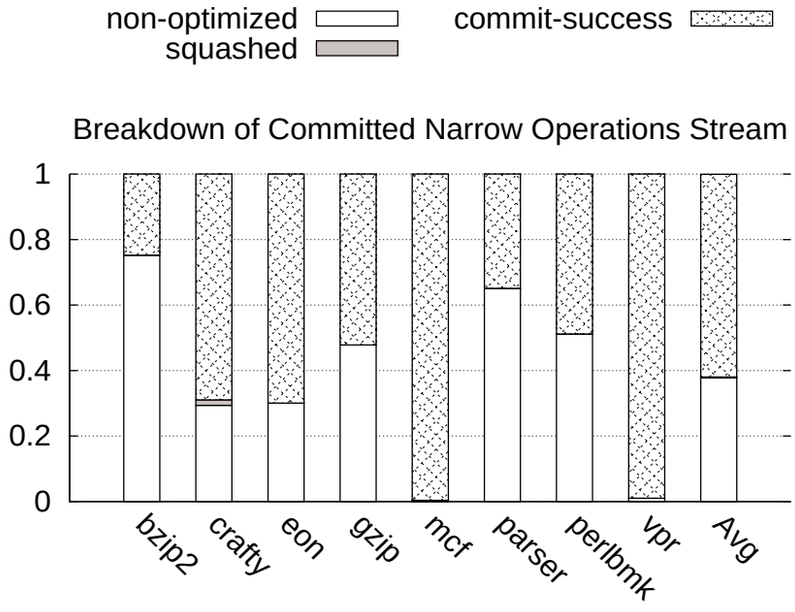

Figure 4.13: Breakdown of the committed narrow operations stream

6.6%. In this section, we analytically examine the pathologies leading to this difference between the expected vs. the achieved gains. Figure 4.14 shows various configurations simulated for the *same subset of selected functions* as those for the experiments to measure the disposable potential of a more Global Productiveness Analysis (results shown in Figure 2.8 in Section 2.4.3.3). For each function –

1. *PerfMem* bar shows the percentage reduction in the dynamic narrow computations with respect to the dynamic non-optimized narrow computations for the 100 statistically sampled dynamic executions of the respective function. For each dynamic instance of a function, oracle dependences (both register and memory) are assumed, and the *dynamic non-productive last-writers and their backslices* are pruned assuming no restrictions or overheads of any kind. Oracle data dependences are those register and data dependences which are observed only in that particular dynamic instance of the function; memory dependences are assumed to be perfectly disambiguated for each dynamic instance. Hence, *PerfMem* bars show the dynamic potential of GPP, if possibly 100 different dynamic versions of the code were to be considered (one for each of the 100 samples).

2. *RealMem-Expected* bar shows the percentage reduction in the dynamic narrow computations with respect to the dynamic non-optimized narrow computations when all the *static*

non-productive last-writers of the respective function and their backslices are pruned assuming no restrictions or overheads of any kind. However, memory dependences are now modeled conservatively. Hence, *RealMem-Expected* shows a more realistic potential of GPP, if a single static version of the GPP code (in wake of conservative memory dependences but assuming no overheads) were to be considered.

3. *RealMem-Achieved* bar shows the percentage reduction in the dynamic narrow computations with respect to the dynamic non-optimized narrow computations when all the *static non-productive last-writers of the respective function and their backslices* are pruned with realistic dependence modeling and with other restrictions induced due to the GPP optimization like most-frequent value bias threshold and contradiction handling (Section 4.2.5). However, no assertion overheads are modeled yet.

All the individual bars show the average ratios as measured for the first 200m x86 user instruction commits. Following insights can be gained from these evaluations :

1. The gap between *PerfMem* and *RealMem-Expected* holds because of two main reasons :
   - Memory dependences were modeled conservatively in the latter. This makes productive and non-productive backslices *share* more computations and hence, less computations are pruned by GPP.
   - Not all dynamic non-productive last-writers (trigger points of *PerfMem*) are static non-productive last-writers (trigger points of *RealMem-Expected*).
2. The gap between *RealMem-Expected* and *RealMem-Achieved* is negligible. This suggests that the computations lost due to other features of GPP like contradiction handling, and unbiased non-productive last-writers handling (use of most-frequent profile bias threshold as shown in line 12 of Algorithm 1) are limited.
3. The code-coverage in all three configurations – *PerfMem*, *RealMem-Expected*, and *RealMem-Achieved* is 100% as only particular functions are considered. The code coverage of GPP in the case of the final evaluations depicted in Figure 4.11 and Figure 4.12 is around 60% as only hot regions are optimized by GPP.

This study also reflects the importance of a robust memory dependence modeling strategy and inclusion of code duplication. Both these strategies can be effectively used to ensure that the non-productive and productive backslices are as segregated as possible. Lastly, code coverage also has an important role to play in further reducing the dynamic code footprint of the narrow ISA via GPP.

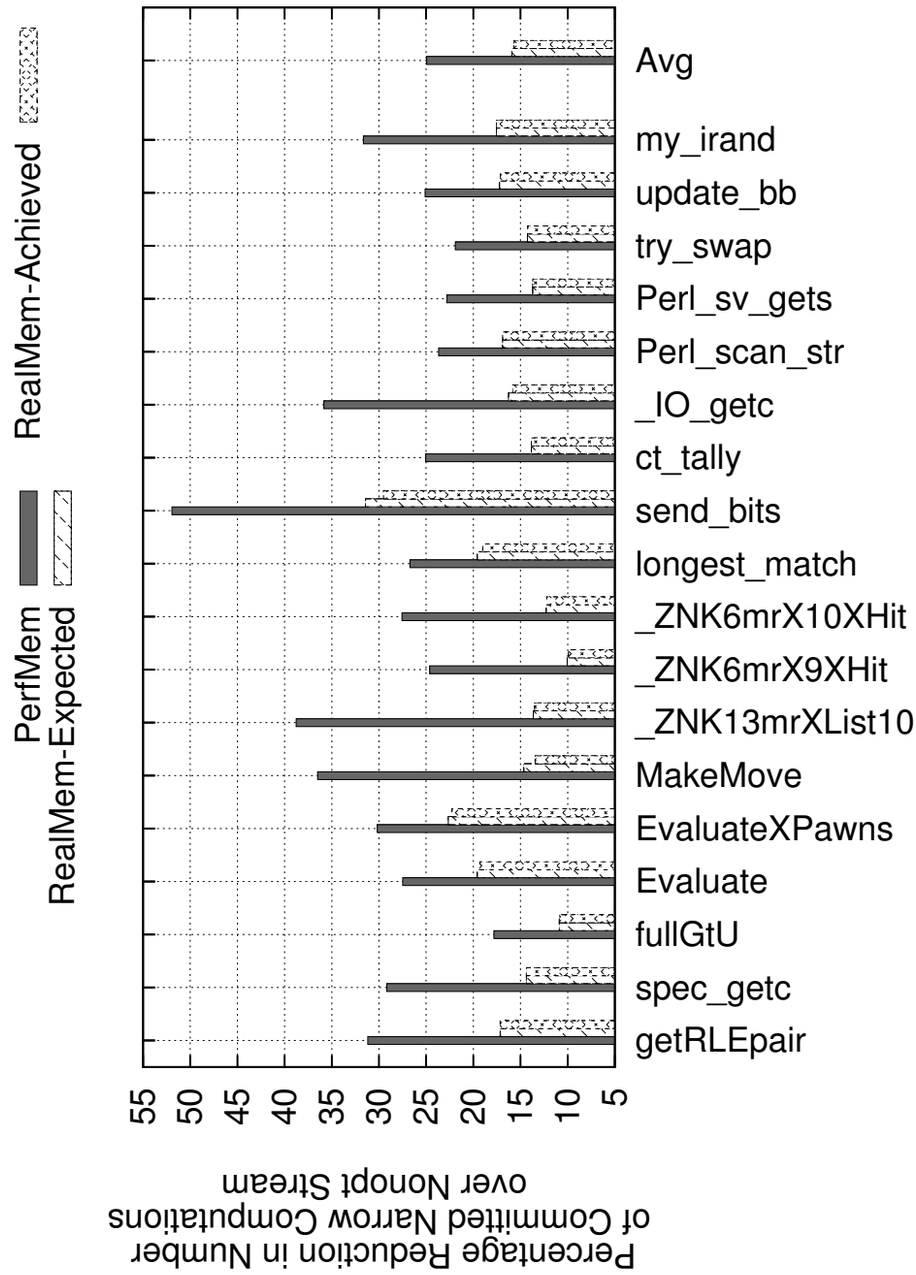

Figure 4.14: Pinning down the bottlenecks in GPP - A subset of selected functions have been evaluated with three different configurations

4.5 Conclusions

In this chapter, we have proposed the first productiveness-based compilation technique to reduce the dynamic code footprint of the narrow ISA. The technique incorporates a profile-based, speculative, aggressive code pruning strategy.

Given a code region, Global Productiveness Pruning distinguishes between useful (productive) and useless (non-productive) narrow ISA computations based on profile data. In order to prune the latter group of instructions, asserts are properly placed to redirect the execution to a safer version of the code when the assumed conditions do not hold at run-time. Overall gains by GPP are up to 6.6% reduction in the committed stream and 4.5% reduction in the number of cycles for a 1-issue, in-order narrow processor, when an average of 60% of the code has been optimized via GPP.

It has been observed that one of the main potential stumbling roadblocks is a conservative memory dependence analysis assumed by GPP. Other robust and aggressive schemes for memory dependence modeling must be studied to unleash the full potential of GPP. As GPP is already speculative in nature, we believe that further speculation on even memory dependences may also be easily accommodated. Conservatively modeled memory dependences cause large fan-out in a bottom-up data-flow analysis and hence, a reason for further conservative actions for code pruning techniques. In this regard, not only an aggressive memory dependence modeling but code duplication may also be useful. Lastly, strategies to enhance coverage of the optimization may also fetch significant benefits.

5

Local Productiveness Pruning

In this chapter, we formalize and describe the second optimization technique under the broader technique of *non-productiveness based pruning*. The optimization is called Local Productiveness Pruning and aims to reduce the dynamic code footprint of the narrow ISA. It applies the heuristic of productiveness to the smallest possible region, i.e. single narrow computation, in isolation. As evaluated in Section 2.4.3.2, assuming a perfect, advance knowledge of the data values generated by each narrow computation, a pruning strategy at such a fine grain granularity has shown a potential reduction of about 48% in the number of dynamic narrow computations.

This chapter begins by establishing a formal definition of Local Productiveness Pruning. Next, it describes the overall flow of the optimizer, followed by the design and implementation of the optimization. Local Productiveness Pruning is a profile-based, speculative code pruning strategy and shares some of its aspects with the previously discussed optimization technique (GPP). Both the similarities and differences with GPP are highlighted in appropriate sections throughout the chapter. Local Productiveness Pruning is then evaluated in both static as well as dynamic optimizer model. Finally, the observed roadblocks preventing Local Productiveness Pruning to achieve its full potential are highlighted.

5.1 Definition

Local Productiveness Pruning (henceforth, LPP) is a speculative, profile-guided code optimization technique that prunes out individual non-productive narrow ISA computations based on their productiveness bias, *when viewed in isolation*. More precisely, LPP can be formally defined as follows :

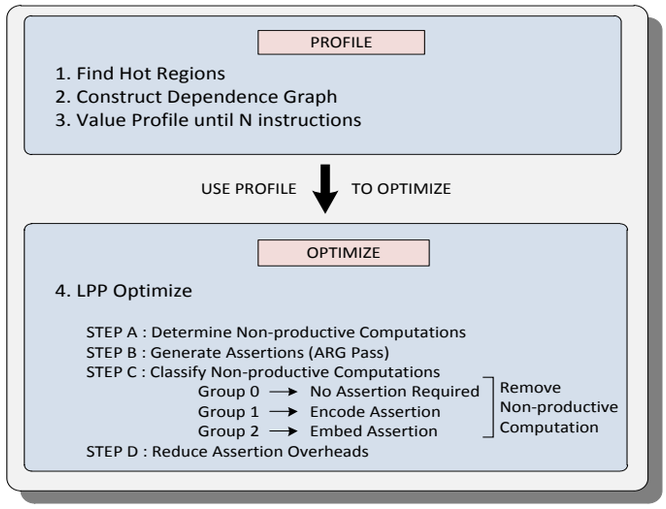

Figure 5.1: Local Productiveness Pruning : An overview with the component passes

Definition 5.1 (**Local Productiveness Pruning**). *Local Productiveness Pruning* on a region is a speculative optimization technique that marks for *inclusion* an individual productive narrow computation. Conversely, LPP marks for *exclusion* the non-productive narrow computations.

Recall that, unless otherwise stated, the notion of productiveness relates to static productiveness (Definition 2.14). Further, static non-productiveness is a profile-based inference drawn from the dynamic non-productiveness (Definition 2.11) of the individual computation. Finally, in the context of LPP, an optimization region consists of a single narrow computation. Hence, all computations are last-writers by definition. All the foregoing inferences are the direct outcome of the previously mentioned background definitions in Section 2.4.1 and more advanced definitions in Section 2.4.2.

5.2 Description

5.2.1 Overview

The overall workflow of the LPP optimization is depicted in Figure 5.1. As LPP is a profile-guided optimization, there are two main components of the optimization process – profile-generation phase, and profile-use and optimization phase.

The profile-generation phase for LPP collects both productiveness profiles and values profiles for the hot regions of a program. Unlike GPP, the optimization region (Definition 2.3) for

LPP is a single narrow computation. As productiveness is a profile-based inference, we need to devise ways to detect the unassumed cases (i.e. those dynamic instances when the pruned computation turns out to be productive) to ensure correctness of the optimization. LPP optimization algorithm achieves this via a *software-based* approach : it *reverse-engineers* each non-productive computation and embeds sufficient checks (auxiliary assertion-like instructions) inside the region to enable it to detect the unassumed cases. This is achieved by a backward-traversal of connected computations in a region at hand to infer the dynamic assertion-based checks via an adapted version of the Assertion Rules Generator (ARG has been previously introduced in Section 4.2.4).

Comparison with GPP. Compared to GPP, LPP adopts a similar overall approach towards code pruning – first, learning productiveness based on profiles, followed by pruning computations, and finally generating assertions to make the code self-sufficient. The essence of the LPP optimization, however, lies in viewing each *individual narrow computation in isolation*, much unlike GPP which views a *chain of computations as a potential single-unit* for pruning. The advantage of working on individual narrow computations in isolation is that it allows more fine-grained maneuvering as compared to slice-based pruning adopted in GPP. Further differences between the approach of LPP vs. GPP are described over the course of this chapter.

5.2.2 Initial Steps

The initial steps to obtain profiles before optimizing the narrow code stream are similar to those required for GPP (Section 4.2.2). Nonetheless, in this section we briefly outline the main steps again for sake of clarity.

As the first step of the profile-generation process, *hot regions* are detected. These are those regions of code that contribute the maximum number of user instructions in the committed trace of a program. Next, a control and data dependence graph[18] of the region, denoted by PDG(R), is created with all register and memory dependences. Memory dependences are handled conservatively.

The profiling required for LPP is similar in concept to that required for GPP (a detailed overview has already been provided in Section 4.2.2). In summary, the two profiles required are :

- *Productiveness Profiles* : For each static narrow computation, the productiveness profile indicates how many times the computation has been *dynamic productive* and the total execution count of the instruction in the profile-phase. Recall that, a dynamic instance of a computation is (dynamic) productive if it drives a change in the associated output storage location (Definition 2.12).

In case of LPP, as the optimization region is a single narrow computation, the values of the storage location before and after the computation are observed to infer its productiveness. This is unlike GPP, where the value of the storage location in the entry and exit of the region are monitored. Hence, productiveness profiling for LPP requires relatively less book-keeping as compared to GPP. LPP requires productiveness profiles for almost all static narrow computations, as against GPP which requires productiveness profiles for only the non-productive last-writer computations.

- *Value Profiles* : LPP also requires value profiles [6, 7] of the component edges of the PDG(R) of the region being optimized. A detailed overview of value profiling for GPP has already been provided in Section 4.2.2. In summary, value profiling for LPP requires the *most-frequent value* profiles of the edges. In our model, we keep :

  1. the ten most-frequently occurring values of each edge, and
  2. the total execution count of each edge.

  The component edges in PDG(R) reflect the data-flow relationships between the narrow computations while accounting for control-flow, if any. In other words, a data-flow edge between two computations reflects a dynamic flow of value from the producer to the consumer. Hence, the number of component edges in the PDG(R) of the region to be optimized is a function of how complex the internal control-flow of the region is. As a logical extension, the cost of value profiling for LPP is expected to be lower than that for GPP, as the optimization region for GPP can potentially contain more complex control-flow.

At the end of the profile phase (of say N instructions[1]), the generated profiles are ready to be used to carry out the profile-guided optimization, and 'LPP Optimization' is performed.

5.2.3 LPP Optimization

This section describes the overall workflow of the LPP optimization (Figure 5.1) in more detail.

5.2.3.1 Step A : Determine Non-productive Computations

For each narrow computation, the first step is to measure its profile-based bias towards being non-productive. This is achieved by calculating the Non-productiveness Ratio (NPR) defined as follows :

[1] N is 200m x86 user instructions in our evaluations

Definition 5.2 (**Non-productiveness Ratio (NPR)**). *Non-productiveness Ratio (NPR)* of a computation c is calculated as ratio of the number of times a dynamic instance of a narrow computation is non-productive to the total number of times the computation is profiled. Mathematically, it is evaluated as :

$$NPR_c = \left[\frac{Number\ of\ Times\ Computation\ c\ is\ Dynamic\ Nonproductive}{Total\ Number\ of\ Times\ Computation\ c\ is\ Profiled} \right] * 100 \tag{5.1}$$

Further, a configurable threshold called the *NPR_Threshold* (typically in the range of 90-100%) is used to filter those computations that will *most-probably* be non-productive. Such computations whose NPR is higher than *NPR_Threshold* are marked as non-productive computations. Once the non-productive computations are inferred based on the profile, the algorithm prunes them and embeds assertions for ensuring correctness, using the Assertion Rules Generator Pass (henceforth, ARG pass).

5.2.3.2 Step B : Generate Assertions (ARG Pass)

The fundamental strategy of the ARG pass for LPP is adapted from that explained in Section 4.2.4. For sake of clarity, we first revisit it to show how LPP derives the rules required for non-productiveness of a narrow computation. The ARG pass eventually applies this basic logic recursively on all non-productive computations.

Recall the scenario where the single computation in the region (Figure 5.2) : *add regD = regS,0x2* is *non-productive* (and hence can be removed speculatively from the region). In order to ensure that this pruning is safe for all future executions, all dynamic instances must satisfy the following conditions:

- Firstly, value of storage location regD *before the computation* is V (as shown by assertion A1 in Figure 5.2). This assertion is placed before the computation.
- Secondly, value of storage location regS before the non-productive computation executes must be (V - 0x2) (as represented by assertion A2 in Figure 5.2).

Hence, in this example, the analysis potentially removes one computation and places two assertions. Placing multiple assertions per single pruned computation is clearly not a cost-effective approach. However, there are two facets of this issue :

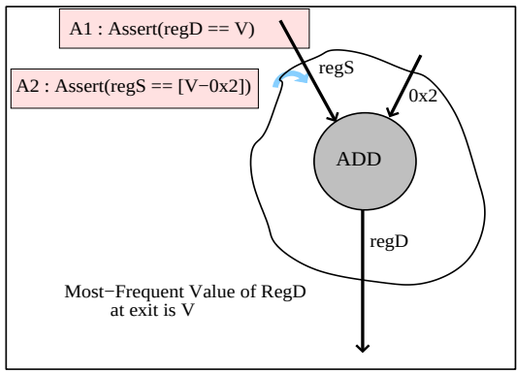

Figure 5.2: Deriving rules for a simple region

1. As shown later, pruning *some* computations correctly does not need multiple (or even single) assertions. This is exploited in the Step C of the LPP optimization pass by classifying computations into groups.

2. The ARG pass *merges* the assertions based on basic producer-consumer relationships, and hence reduces redundancy of assertions. This is exploited in the Step D of the LPP optimization pass.

3. The ARG pass may also *compress* the assertions by exploiting a 3-bit encoding scheme for register values. This too is exploited in the Step D of the LPP optimization pass.

5.2.3.3 Step C : Classify Non-productive Computations

In this section, we introduce the classification of the narrow computations – Group0, Group1, Group2, Group3 in the increasing order of the required complexity to exploit their non-productiveness. This section also reflects upon the data property which is responsible for the high number of non-productive computations across programs.

Group0 : Statically Determinable Non-productive Computations. These are those computations whose outcome can be determined statically by analyzing the computation by itself and without any *propagation* of data values or other statically available information across instructions. For example, the outcome of the instruction – *mov* $inaddr_3$ = *0x2aab*, can be inferred statically. Specific examples of Group0 type of computations are given in Table 5.1.

The relevance of identifying Group0 computations in the current context is that they do not need any assertions at run-time to ensure their non-productiveness. For example, the mov $inrip_i = Imm_i$ operations are generated to update the internal register *inaddr* with the target

Table 5.1: Instruction templates for Group0 instructions

Id	Group0 Instruction Templates ; No Asserts are Required for pruning them
1.	mov $inrip_i$ = Imm_i **Taken Addr Generation Ops** for Branch Operations. Mov Taken IP addr for a **Branch**; inrip is an additional internal register used for branching.
2.	mov $inaddr_i = Imm_i$ Selected **Addr Generation Ops** for Memory Operations. For example, the absolute address of the mem ops that access global variables is often known statically. Recall that $inaddr_i$ is a chunk of the internal register for memory accesses.
3.	xor rax_i = rax_i, $[Imm_i = 0x0]$ maskc rax_i = rax_i, $[Imm_i = 0xff]$ **Logical Operations** where the mask makes the operation an *identity* operation.

address of conditional and unconditional branches. In case of indirect jumps the target address is first loaded from the memory. Note that, a simple static analysis of the control-flow of the program is sufficient to determine the non-productiveness of such operations.

Hence, it must be emphasized that Group0 computations *can* also be eliminated by the *narrow translator* by incorporating an additional minimal data-flow analysis in itself. For sake of simplicity, however, the inference of Group0 non-productive computations remains profile-guided.

Table 5.2: Instruction templates for Group1 instructions. The required assertions for these computations can be encoded by special operation-and-assert opcodes. 'cf' indicates carry flag and 'of' indicates the overflow flag

Id	Instruction Templates (Group1)	Required Assertion (Requires Modified Opcode Add/Sub-n-Assert)
1.	subc rsp_i = rsp_i, $[Imm_i = 0x0]$ Decrement Index / Pointer	subc(!of) rsp_{i-1} = rsp_{i-1}, Imm_{i-1}
2.	addc rax_i = rax_i, $[Imm_i = 0x0]$ Increment Index / Pointer	addc(!cf) rax_{i-1} = rax_{i-1}, Imm_{i-1}

Group1 : Flag-Dependent Identity Operations. These are those addition / subtraction operations that have the following two characteristics :

(i) same source and the destination register storage locations, and

(ii) a small-immediate to increment and decrement the value held in the register storage location (refer to Table 5.2).

For example, addc rax_1 = rax_1, 0x0 is the narrow add operation that updates the $chunk_1$ of the 64-bit *rax* register and is generated as part of the translation of the 64-bit add operation : add rax = rax, 176. As the source and the destination registers are the same, in order to assert the dynamic non-productivity of this operation, it is sufficient to assert that the carry bit generated by add rax_0 = rax_0, 0x176 is zero. Table 5.2 shows two generic templates of Group1 computations.

The relevance of identifying Group1 computations in the current context is that they can be effectively pruned by using auxiliary opcodes : additional opcodes which perform two functions in conjunction – Add/Sub and Assert no ConditionCode.

This group also underlines a fundamental difference of strategy between LPP and the related previous work of Value Range Speculation (VRS) [9] which can uncover narrow bitwidth computations. Recall that VRS is a code optimization technique whereby code regions are optimized for value ranges. Compared to VRS, LPP allows safe pruning of even those computations which VRS deems useful (and hence, cannot be removed by VRS); because the value range may be wide, but the change in value may still be narrow. For instance, an add rax_0 = rax_0, 0x1 operation, in an infinite loop, will generate a carry only once in 2^{16} executions and can be inferred as non-productive. It cannot be removed by VRS, however.

Group2 : Complex LPP Operations. Group2 set of computations are essentially all non-memory (and non-Group0, non-Group1) computations which when non-productive, require at least one added assertion for their safe removal.

Some templates of Group2 computations are given in Table 5.3. For example, an addc rax_1 = rdx_1, rcx_1 operation, if non-productive, will require assertions on both input and output storage locations. Hence, to remove this narrow computation, the following assertions are required : assert (rdx_1 == Val1), assert (rcx_1 == Val2), and assert (rax_1 == Val3), where Val1, Val2, and Val3 are the most-frequent profile values satisfying the equation that Val3 = Val1+Val2.

Table 5.3: Instruction templates for Group2 instructions. Pruning each Group2 computation requires at least one explicit assertion operation

Id	Instruction Templates (Group2)	Required Assertion(s)
Single Assertion (Low Overhead)		
1.	**x86 merging rules** for a 32-bit operation forces the higher chunks to zero.	
	(a) mov rax_2 = zero	(a) assert (rax_2 == zero)
	(b) mov rax_3 = zero	(b) assert (rax_3 == zero)
2.	**Initialize with zero** operations.	
	(a) xor rax_i = zero, zero	(a) assert (rax_i == zero)
	(b) xor rax_i = rbx_i rbx_i	(b) assert (rax_i == zero)
Multiple Assertions (High Overhead)		
3.	**Addition** ops with either (a) rd $\neq$ ra, or (b) ra $\neq$ rb	
	addc rax_i = rdx_i, rcx_i	asserts on all input and output storage locations with most-frequent values
	addc rax_i = rdi_i, Imm	
4.	**Logical** ops with either (a) rd $\neq$ ra, or (b) ra $\neq$ rb	
	and/andc rdx_i = rdx_i, rcx_i	-Same as above-
	andc rax_i = rdi_i, Imm	

Group3 : Memory Operations. This last group consists of memory operations (templates provided in Table 5.4). These operations are the most costly in terms of assertion overheads as compared to the previous groups of computations (Group0, Group1, and Group2). This is because removing a LPP non-productive memory operations requires not only a register based assertion but also a load-assert (effectively a load from memory and assert on value) operation. Due to this complexity, these operations are *excluded* from the current LPP analysis. Exploiting memory-based LPP operations may require extra support from the memory hierarchy, e.g., in the form of additional bit-encoding support, to reduce their assertion overhead costs.

We have investigated on some possible ways of extending LPP to memory operations via Memory Productiveness Pruning (MPP [3]). The techniques, as they are, incur overheads in the form of additional non-negligible hardware support and more research is required to exploit non-productiveness of memory operations. More research on the same is deferred for future work. An overview of MPP is provided in Chapter 7.

Table 5.4: Instruction templates for Group3 (LPP memory operations) instructions. mfValue indicates the most-frequent profile-based value encoded as the immediate

Id	Instruction Templates	Required Assertions
1.	ld rcx_i = Mem[inaddr]	load-assert Mem[inaddr] == mfValue; assert (rcx_i == mfValue)
2.	st Mem[inaddr] = rax_i	load-assert Mem[inaddr] == mfValue; assert (rax_i == mfValue)

Discussion. Figure 5.3 provides quantitative insights into the classification of the non-productive narrow operations into the above-mentioned groups for the profile-phase of the applications (first 200m committed user instructions) [1] assuming perfect profile, i.e. assuming that we know whether the computation is productive or not before it executes. Note that, on an average, Group0, Group1, and Group2 account for more than 80% of all the non-productive narrow computations. These are also the groups of our focus in this chapter. Finally, Group3 accounts for about 20% of the non-productive computations.

5.2.3.4 Step D : Reduce Assertion Overheads

Our experiments reveal that around 40% of the LPP non-productive computations belong to Group2 (Figure 5.3), which require additional (and at times, *multiple*) assertion instructions to affirm their non-productiveness at run-time. The dynamic weight of Group2 computations suggests that a naive approach of introducing assertion checks individually for each computation will unequivocally lead to large assertion overhead costs. To mitigate this problem, the ARG pass additionally exploits the following two data properties :

- *Simple producer-consumer relationships* between computations to remove redundancy of assertions between chains of computations.
- *Encodability* of asserted values to combine multiple assertions into a single instruction.

To understand how producer-consumer relationships can be exploited to merge assertions, the distinction between the notion of an optimization region vs. atomic region must be made. Optimization Region is defined as the code region which is the candidate for optimization. Clearly, the optimization region for LPP is a single narrow computation. To reduce the large

[1] this experiment uses the *ref* input data-set

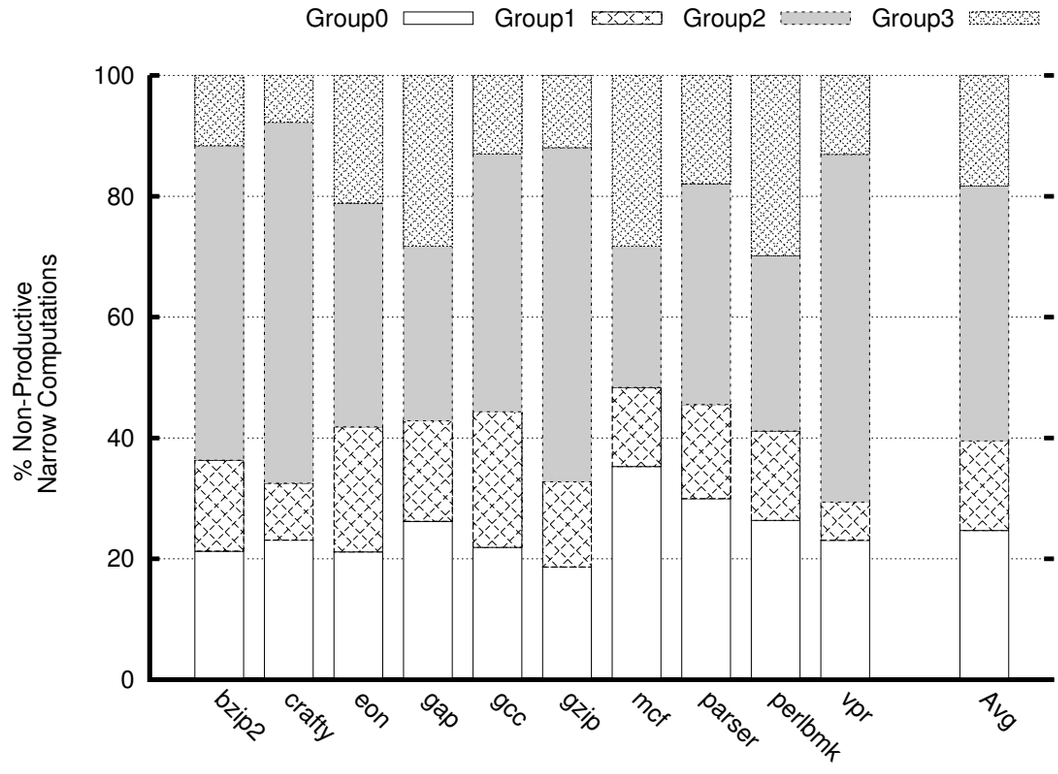

Figure 5.3: Categorizing the non-productive LPP computations into Group0, Group1, Group2 and Group3 to understand where the gains are coming from

assertion overhead costs, however, the strategy of LPP is to inflate the size of the atomic regions (from the default of a single narrow computation to say, a basic block or superblock).

In this thesis, atomic regions can be formalized as follows :

Definition 5.3 (**Atomic Region**). An *atomic region* is a section of code which is guaranteed to be executed and committed as a single-unit. Such a region has only a single flow of control. Only the start and the end of the atomic region represent the points of precise state. Atomic regions are executed speculatively and as such, they may commit or squash.

Hence, an atomic region can be used to *contain* multiple speculations : by traversing producer-consumer relationships in an atomic region, assertions can be merged.

Merging Value-Assertion Requirements. Figure 5.4 shows the two theoretical degenerate possibilities of a producer-consumer relationship graph witnessed in a basic block. The clas-

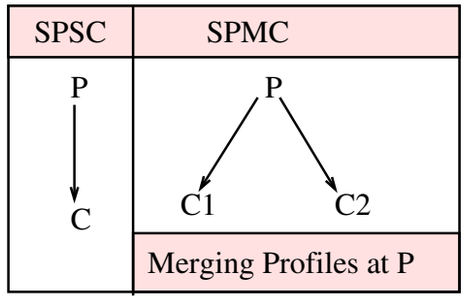

Figure 5.4: Possible types of flows – SPSC and SPMC

sification is valid even for a superblock as its an atomic region with only a single control-flow. Hence, although the upcoming discussion cites basic block explicitly, it holds true for superblock as well. As shown in Figure 5.4, the two types of possible data-flows are –

(i) Single Producer Single Consumer (SPSC)

(ii) Single Producer Multiple Consumers (SPMC)

Note that, Multiple Producers Single Consumer (MPSC) scenario does not arise in a basic block as there is a single control-flow.

In context of the ARG bottom-up traversal, SPMC presents the point of *merging* of value profiles. Such a *merge* of profiles is hassle-free : at the producer P (see SPMC), we need to propagate only a single assertion upwards, as the most-frequent values (or asserted values) of edges P-C_1 and P-C_2 must be equal.

Note that in case of SPSC and SPMC, if P itself is non-productive, then we effectively merge the assertion requirements of the consumers (on the RAW edges P-C, P-C_1 and P-C_2 respectively into P), and place assertions only to ascertain the non-productiveness of P. If on the other hand, P is productive, and if the non-productive consumers C, C_1 or C_2 have assertion requirements (to be fulfilled by their producer P), an assertion is left embedded after the computation P. In summary, the ARG pass re-engineers each computation recursively while merging the requirements bottom-up, until either the beginning of the region is encountered (we process computations in post-order) or the boundary with a productive computation is reached.

Assertion Compression. Many previous proposals have exploited the disposition of data values to have all their higher bits as 0's or 1's, and hence, using sign-extension or zero compression [4, 8, 36, 47] to uncover traditional narrow computations and enhance pipeline-gating, or reduce dynamic activity. In our model, the above-mentioned propensity of data values is exploited to *compress assertions*.

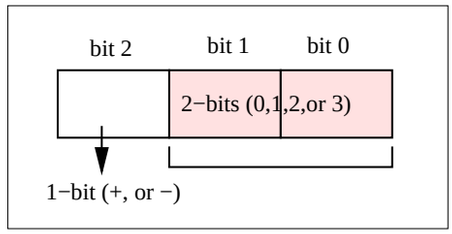

Figure 5.5: Size-sign encoding (3-bit encoding)

We exploit a 3-bit encoding known as the *Size-Sign encoding* whereby the two lower bits indicate that highest chunk position (0^{th}, 1^{st}, 2^{nd} or 3^{rd}) beyond which all higher significant chunks are either all zeros(+) or ones(-) as shown in Figure 5.5. For example, if rax_3 = 0x000, rax_2 = 0x0000, rax_1 = 0xfde1, rax_0 = 0x2312, the 3-bit encoding associated with the 64-bit logical register rax is 1+; while if the chunks contained rax_3 = 0x0000, rax_2 = 0xffff, rax_1 = 0x0000, rax_0 = 0x2e2e, the 3-bit encoding would be 2+. Note that, the encoding values are associated with 64-bit data values for the purpose of compression.

We exploit a new narrow ISA opcode, *assert-enc*, having semantics as follows :

assert-enc [reg_id] [3_bit_encoding]

where *[reg_id]* is a immediate field which indicates which of the 64-bit registers is being asserted for. Further, *[3_bit_encoding]* is an immediate field that stores the 3-bit encoding that the register must adhere to. The 3-bit encoding is incorporated only for register-based storage locations and not in the memory hierarchy. Each logical 64-bit register is enhanced with a 3-bit encoding. Updates to the encoding are carried out on the event of writeback of narrow data chunks to the register file by additional hardware logic incorporated in the register file.

Compression Scheme (SR-CS). We use a ***S**ingle-**R**egister* based assertion ***C**ompression **S**cheme* (henceforth *SR-CS*), whereby contiguous narrow assertion computations in the static stream are merged into a single *assert-enc* assertion. The following conditions must be satisfied for *SR-CS* compression :

1. All narrow asserts must affirm different consecutive chunks of the same 64-bit register.

2. There does not exist any intermittent non-assert based computation between the asserts. Note that, this is a conservative way to ensure in between the assertion computations being combined, there is no update to the chunks of the 64-bit register being asserted for.

3. The asserted values of the chunks must allow size-sign encodability as outlined previously in Figure 5.5.

Table 5.5: Illustrating compression schemes with examples. $regA_x$ denotes the x^{th} chunk of regA being asserted for

Id	Input Assertions	Output Compressed Assert
SR-CS Single-Register Based Compression Scheme		

1.	assert ($regA_1$ == zero);
	assert ($regA_2$ == zero);
	assert ($regA_3$ == zero);
		assert-enc ($regA$ == [0+])

2.	assert ($regA_1$ == Val);
	assert ($regA_2$ == zero);
	assert ($regA_3$ == zero);
		assert ($regA_1$ == Val); // Non-compressed Assert
		assert-enc ($regA$ == [1+]);

Dual-Register Based Compression Scheme		
3.	assert-enc ($regA$ == 1+) assert-enc ($regB$ == 1-)	assert-enc ($regA$,$regB$, 1+,1-) // assert regA is 1+ and regB is 1-

Those assert computations which are not size-sign encodable cannot be compressed (template 2 in the Table 5.5). Hence, those asserts that are not encodable remain embedded in the static stream as they are.

Finally, just like *SR-CS* generates single wide-register based asserts, other schemes which generate dual (template 3 in the Table 5.5) and quad-register compressed asserts are also possible. However, we observed a negligible reduction (a further reduction of 0.7% only) in total number of dynamic assertions when upgrading from single-register to dual-register compression scheme. Hence, we believe that *SR-CS* represents a good design point.

Conclusions. Algorithm 2 summarizes the workflow of the LPP optimization. The computations of the optimization region are processed in post-order (line 1). For each narrow computation, the Non-productiveness Ratio (NPR) is determined using profiles. If the NPR is higher than a threshold, the node is considered non-productive (line 3). The optimizer then proceeds to prune the node (narrow computation). First, it obtains the most-frequent value of the destination register of the node. The assertion generated using this value is designated as assertionA1. Next, this assertion is merged (merging assertions has been previously illustrated in Figure 4.5) and sanity checks are performed to ascertain that there are no contradictions. Finally, the optimizer proceeds according to the groupID to prune the computation (lines 8 through 23)

Assertion compression is performed at the final stage after all the computations of the region have been processed (line 27 of Algorithm 2).

ALGORITHM 2: **LPP_Optimize_Region**

```
   // Post Order traversal of the optimization region
1  for each node in post order travesal do
2      Get Non Productive Ratio (NPR) for the node;
3      if node is non productive then
4          Get MostFreq Output Value from ValueProfile for the node;
5          assertionA1 ← assert(destination register == mostfreq output value);
6          mergedAssertionA1 ← Sanity Check for node with assertionA1;
7          groupID ← classify node into groups;
8          switch groupID do
9              case Group3
                   // LPP Group3 are not included in the optimization
10                 break;
11             end
12             case Group2
13                 Process group 2 node for mergedAssertionA1;
14                 break;
15             end
16             case Group1
17                 Process group 1  node for mergedAssertionA1;
18                 break;
19             end
20             case Group0
                   // LPP Group0 do not need any assertions
21                 mark node NonProductive;
22                 break;
23             end
24         end
25     end
26 end
   // Reduce Assertion Overheads by Compression
27 PostProcess Region;
```

5.2.4 Cost Analysis

The cost of optimizing an atomic region by LPP is expected to be proportional to *(n + e)*, where '*n*' and '*e*' denote the total number of nodes and the total number of edges respectively in the PDG(R) of the atomic region to be optimized. This is because each node in the PDG(R) of the atomic region must be visited at least (and at most) once. Further for each node, all the successor data dependence edges and the predecessor data dependence edges must be analyzed. As a data-flow dependence edge is shared between a producer and a consumer, some edges may be visited at most twice.

5.3 Example : Walk-through

Now we illustrate the LPP optimization by using an example (refer to Figure 5.6). The example is based on a small piece of code from a hot basic block of the benchmark vpr (function – try_swap). The original code sequence consists of four dependent add operations, of which only the second add operation is a 32-bit add operation (rest are 64-bit add operations). Hence, according to x86 semantics, higher two chunks of *rax* generated by the narrow ISA computations corresponding to the second add operation are mov operations that restore the destination register location to zeros (nodes 7 and 8 in Figure 5.6 (A)).

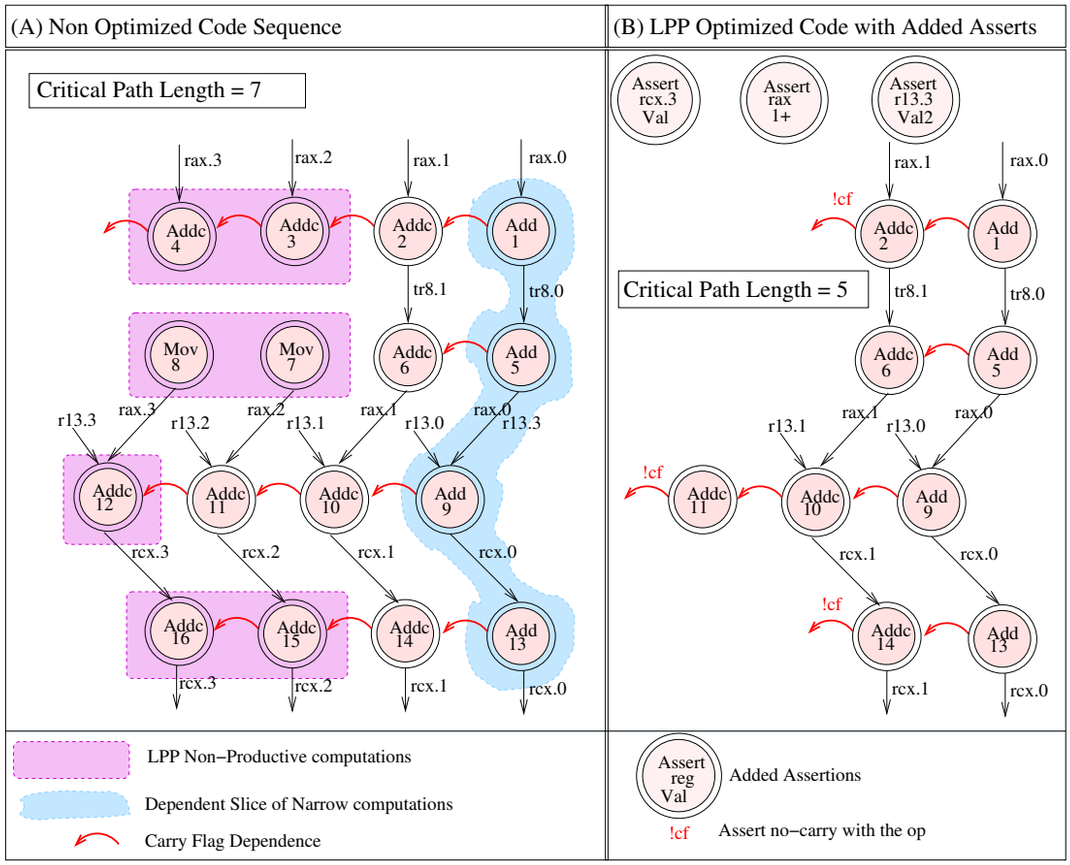

Figure 5.6: LPP optimization on a sample code sequence from vpr - Comparing Non-optimized and LPP Optimized Code sequence

Let's walk through the example depicted in Figure 5.6. Only RAW dependences between computations are depicted. First, the non-productive computations are marked using profiles (step A in Section 5.2.3.1); these computations are marked in Figure 5.6. Next, the ARG

pass begins processing the narrow computations in post-order (reverse program order). Hence, node 16 is processed : as it is non-productive (Group1), pruning this computation requires an assertion on the carry-flag edge (henceforth, denoted as Assert(!cf)) as shown in Table 5.2.

The next non-productive computation in post-order, node 15, merges the Assert(!cf) with its own requirements of non-productiveness, which then generates a similar assert as it is also of category Group1. As node 14 represents the boundary of the productive region, node 14's opcode is modified to addc(!cf) i.e. add operands with the carry flag of the predecessor (node 13) and assert that the computation does not generate any carry flag. Node 14 and 13 are skipped as they are productive.

Next, node 12 is a non-productive node belonging to Group2. Hence, pruning it requires the following asserts – (1) assertion that rcx.3 before computation is Val, (2) assertion on the carry-flag edge, (3) assertion of most-frequent value of r13.3, and (4) assertion of most-frequent value of rax.3. Note that assertions 2, 3 and 4 must always be in sync. Also note that assertion (4) on the value of rax.3 will be further merged with the predecessor nodes. The rest of the nodes are processed in a similar way.

The data-flow graph represented in Figure 5.6 has some producer-consumer chains (RAW dependences), and hence at the end of the ARG pass, three additional single-latency asserts are added (we assume for the sake of this example that the higher chunks of *rax* being asserted for are size-sign encodable) and 7 narrow computations can be removed.

5.4 Evaluation

In this section, the performance evaluation of the LPP optimization is presented. First, we briefly revisit the experimental framework (already described in Chapter 3) in the context of LPP to aid the understanding of the upcoming evaluations. The performance of LPP is compared against the baseline narrow processor (described in Section 3.2), which is an in-order 16-bit integer datapath processor combined with a realistic narrow translator (outlined in Section 2.2.2).

Some aspects of the design space of the LPP optimization have been evaluated. Section 5.4.2 and Section 5.4.3 provide insights on the benefits of the optimization in a static and dynamic optimizer model respectively. In both the optimization models, the overall gains of LPP have been quantified in terms of reduction in narrow computations, reduction in the number of cycles consumed to accomplish the same amount of work, assertion failure statistics and eventually the code coverage of the optimized regions. Further, Section 5.4.4 investigates on how the overall optimization is affected by the changing the notion of an atomic region from a basic block to a superblock.

Lastly, in Section 5.4.7, we comment on LPP's achieved performance against the disposable potential projected in a perfect environment.

5.4.1 Experimental Framework

Local Productiveness Pruning (LPP) has been evaluated as both static and dynamic optimization. In the case of dynamic optimization, it is assumed that the baseline ecosystem comprises of a narrow processor with support for dynamic optimizations. The evaluation of LPP in both static and dynamic optimization models, *does not account for the overheads of profiling and optimization* of the narrow code stream.

Table 3.1 shows the simulation configurations for evaluating LPP on a narrow processor. We model an in-order processor of an issue width of up to four instructions per cycle and compare the performance of a 'narrow processor with the LPP optimized code' against that of a 'narrow processor without such an optimization'.

The execution model profiles for first 200m (after skipping the program initialization phase) and then triggers the LPP optimization. Table 3.3 shows the percentage of the committed stream which is optimized (Expected Coverage) by LPP. The profile-phase of the application uses the *ref* or *training* input data-set in dynamic and static optimizer models respectively.

The optimized regions are then used in the cycle-accurate phase for 200m x86 user instructions. At the beginning of the execution of an optimized region, the system state is checkpointed. In case of assertion failure event in the optimized region, hardware support restores correct program state by using the checkpointed system state. The execution then resumes with re-translated, safe, correct code (non-optimized stream of narrow computations) of the region (i.e., basic block in this evaluation).

5.4.2 LPP as a Dynamic Optimization

The basic workflow to evaluate LPP in a dynamic optimizer model has been illustrated in Figure 3.2. The advantage of a dynamic optimizer model is that the program is optimized on the fly, hence, the profile-based learning is more precise (as its on the same input as the current run). However, such a model entails higher costs than a static optimizer model because the run-time of the application must bear the time and space overheads of profiling and optimization.

Dynamic Stream Classification. Table 3.3 states that the average percentage of the committed stream (x86 instructions) that is expected to come from the optimized regions is around 65%. However, the assertion failures may impact the ratio of optimized stream to the non-optimized one (recall that in event of assertion failure, the speculatively committed narrow

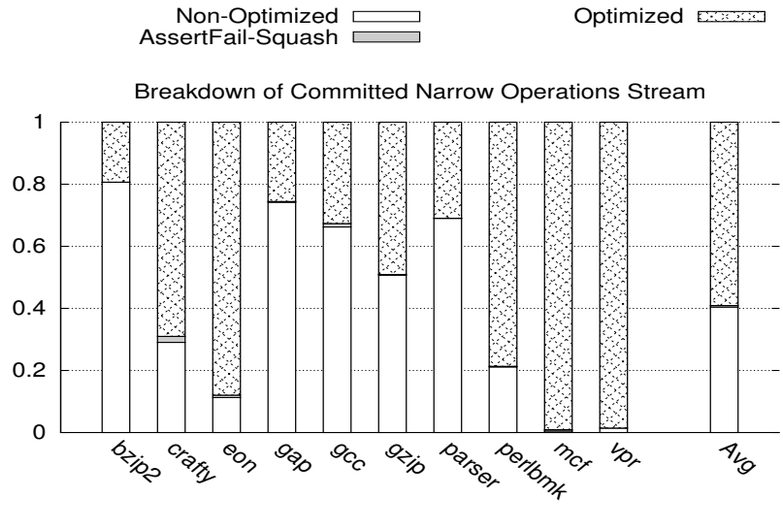

Figure 5.7: LPP in a dynamic optimizer model - Breakdown of the committed stream

operations are squashed, and non-optimized code is executed). Cycle-accurate run of the programs reveals that (see figure 5.7) an average of 59% of narrow operations are committed from the optimized regions. Work lost due to assertion failures is negligible because of many reasons – (i) assertion failure rates are low (Figure 5.8), (ii) basic blocks are small regions to be considered for an atomic commit, and finally (iii), sometimes assertions are placed at the beginning of the basic block itself, hence the work lost due to failure is minimal.

Assertion Failure Statistics. There are two type of assertions that LPP embeds in the code stream – (a) type *AssertOp* : additional computations of type assert, and (b) type *AssertNoCCFlag* : enhanced addsub opcodes (e.g. addc(!cf)) which trigger an event of failure at execution if the assertion on the condition code flag is false. Figure 5.8 shows the ratio of such failure events to the total number of dynamic basic blocks committed (although low, all the programs have non-zero failure rates). As can be seen, the average failure rate is 2.3%. gcc has high assertion failure rate (14%), most of which are of *AssertNoCCFlag* type. Further, the failures in gcc are mainly due to a single static instance of an increment stack-pointer instruction, which due to profile-phase change, increments a higher value of stack where the *change* overflows beyond the expected lowest chunk.

LPP with Compression Schemes. Figure 5.9 (a) shows the reduction in number of narrow operations achieved with LPP. We compare the number of committed narrow operations in the two scenarios – LPP with no compression for assertions (*LPP-NoCompression*), and LPP with

Figure 5.8: LPP in a dynamic optimizer model - Assertion failure rates

Single-Register based Compression Scheme (*LPP-SR-CS*). Both the configurations account for the cost of assertion failures. In other words, assertion computations are also accounted for in the committed stream and further, on the event of a failure, all computations *committed* (and hence, accounted for) since the beginning of the basic block are squashed, and the execution begins afresh from the beginning of the basic block. As the assertion failure events remain the same in both the configurations of *LPP-NoCompression* and *LPP-SR-CS* (in number and type), an average difference of 2.3% in the number of committed narrow operations is mainly due to the efficacy of the *SR-CS*. Figure 5.9 (b) shows the total number of committed assertions only as ratio of the total number of committed narrow operations in the non-optimized stream. *SR-CS* is able to reduce the number of assertions by about 33%. Overall, LPP with *SR-CS* reduces the committed stream by about 20%.

Effect on Number of Cycles. Figure 5.10 shows the number of cycles taken by each benchmark to complete the cycle-accurate phase (the next 200m x86 user instructions after the profile-phase). An average performance improvement of 18% is observed across benchmarks. One of the key factors influencing the gains obtained by LPP is the coverage of the optimized code. Some benchmarks suffer with low code coverage e.g., bzip2, gcc, and gap. Although low code-coverage remains to be a caveat of our infrastructure – we profile for hot functions (barring recursive ones) and then choose the basic blocks, nothing prevents the optimization of all the code. vpr achieves a code coverage of about 99%, and reduces the number of cycles by a significant amount of 40%. mcf fetches low gains with LPP as it has high percentage of

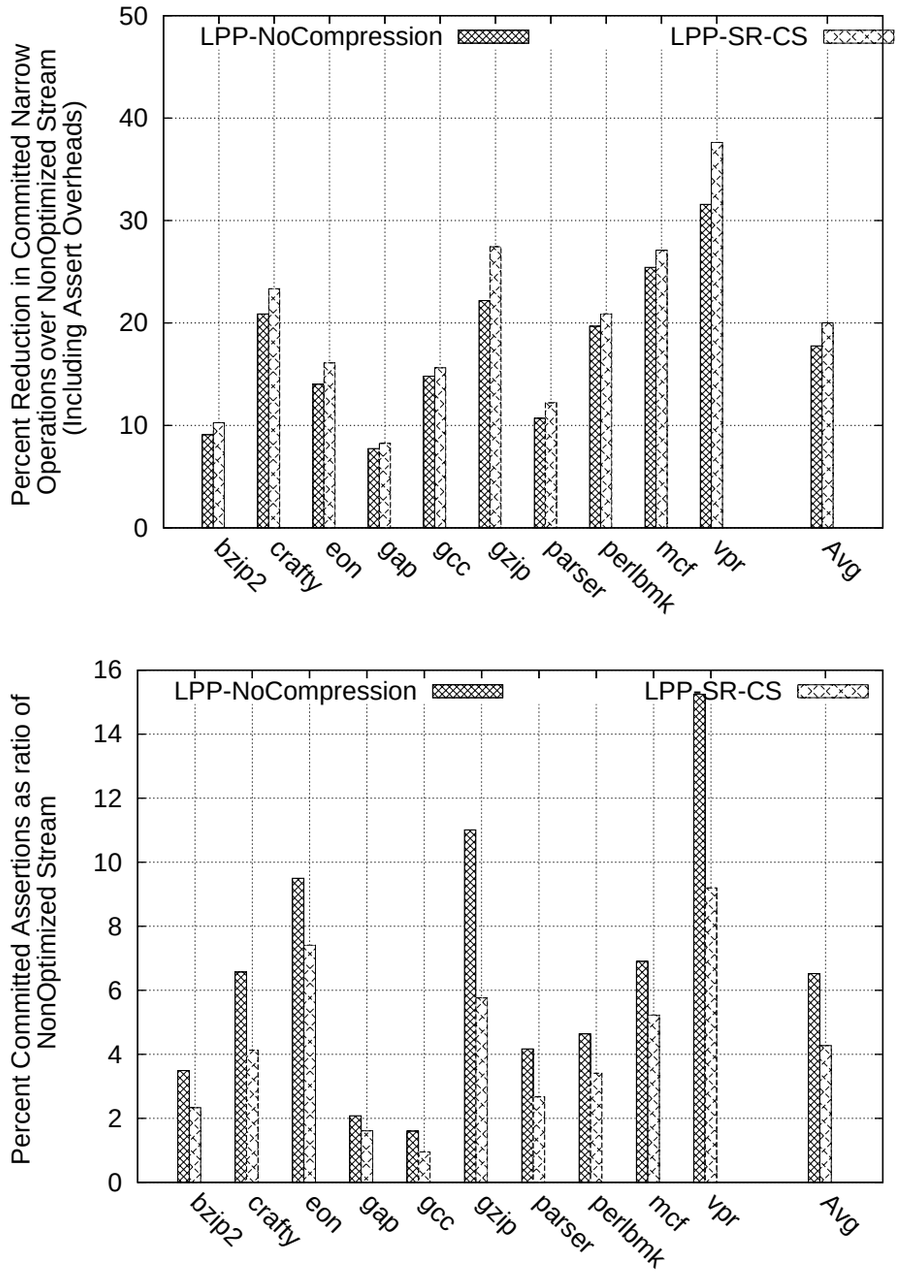

Figure 5.9: LPP in a dynamic optimizer model - (a) Reduction in committed stream (b) Overheads as ratio of non-optimized stream

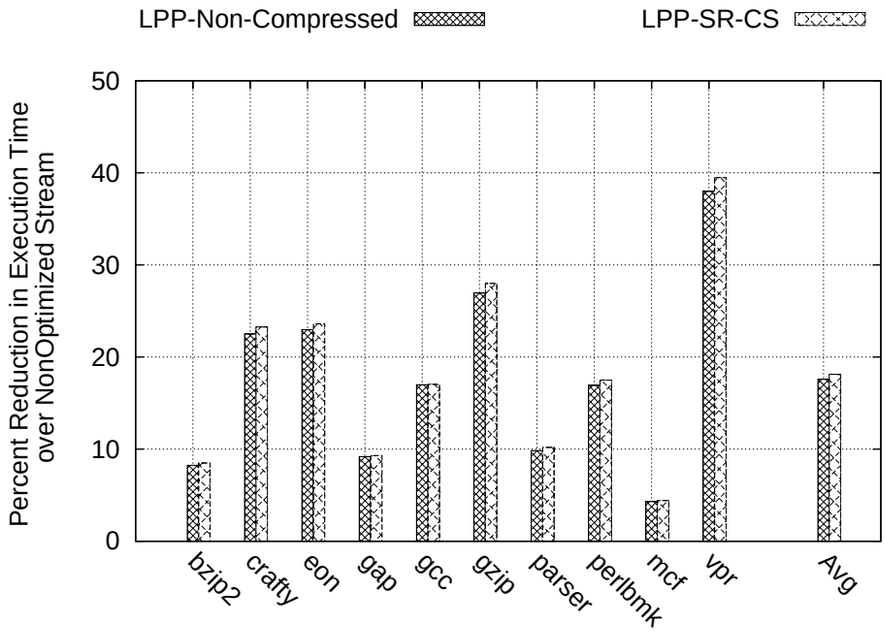

Figure 5.10: LPP in a dynamic optimizer model - Effect on number of cycles

memory instructions (Group3), which LPP does not optimize.

Conclusion. A dynamic optimizer, with more accurate profile data, obtains a performance improvement of 18% with LPP with a single-register compression scheme (LPP-SR-CS), about 20% reduction in number of computations, and an assertion failure rate of 2.3%. An average of 60% of the dynamic narrow stream is successfully optimized. All these gains surely involve an additional overhead (of profiling and optimization at run-time) that are not measured in this thesis.

5.4.3 LPP as a Static Optimization

The basic workflow to evaluate a static optimization is illustrated in Figure 3.2. As against the previous evaluation, profiling is performed using the *training* data-set of each benchmark. As the data-set used for learning is different from the actual input data-set, the effect of this somewhat imprecise[1] learning can be seen in different aspects including the achieved code coverage of the LPP optimization, reduction in number of computations and number of cycles,

[1] in the loose sense of the word

and also assertion failure rates. The upcoming paragraphs provide quantitative details on these aspects of LPP as a static optimization.

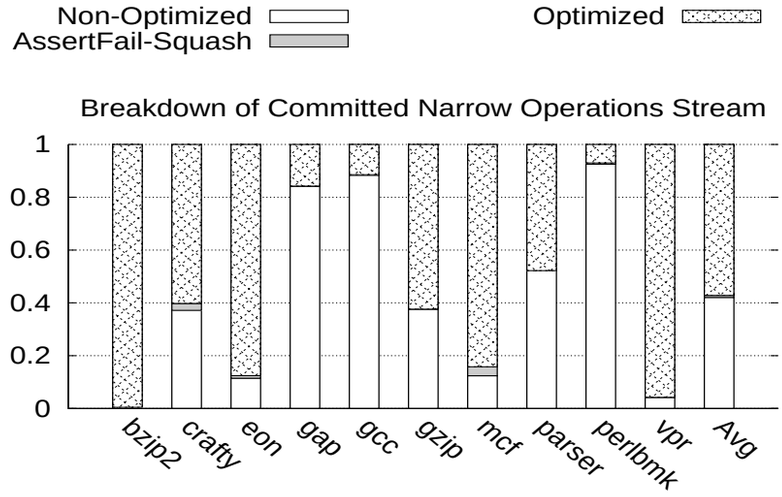

Figure 5.11: LPP in a static optimizer model - Breakdown of the committed stream (achieved coverage)

Dynamic Stream Classification. As stated previously, the average percentage of the committed stream (x86 instructions) that is expected to come from the optimized regions is around 60%. However, the assertion failures may impact the ratio of optimized stream to the non-optimized one (recall that in event of assertion failure, the speculatively committed narrow operations are squashed, and non-optimized non-scheduled code is executed). Cycle-accurate run of the programs reveals that (see Figure 5.11) an average of 57.18% of narrow operations are committed from the optimized regions. Work lost due to assertion failures is negligible again (for the same reasons as previously mentioned in the dynamic optimization evaluations)

Assertion Failure Statistics. Figure 5.12 shows the assertion failure rate of each benchmark in a static optimizer model (although low, all the programs have non-zero failure rates). As can be seen, the average failure rate is 2.52%. perlbmk has high assertion failure rate (14%), most of which are of *AssertNoCCFlag* type. Again, the failures in perlbmk are mainly due countable number of static instances of increment stack-pointer / decrement stack-pointer instructions, which owing to a different input set, increment / decrement a different value of stack where the *change* overflows beyond the expected lowest chunk.

Figure 5.12: LPP in a static optimizer model - Assertion failure rate

LPP with Compression Schemes. Figure 5.13 (a) shows the reduction in number of narrow operations achieved with LPP in a static optimizer model. We compare the number of committed narrow operations in the two scenarios – LPP with no compression for assertions (*LPP-NoCompression*), and LPP with Single-Register based Compression Scheme (*LPP-SR-CS*). The assertion computations are accounted for in both the configurations. Further, on the event of an assertion failure, all computations *committed* (and hence, accounted for) since the beginning of the basic block are squashed, correct state is restored and the execution begins afresh from the beginning of the basic block. As the assertion failure events remain the same in the both the configurations of *LPP-NoCompression* and *LPP-SR-CS* (in number and type), an average difference of 2.7% in the number of committed narrow operations is mainly due to the efficacy of the *SR-CS*. Figure 5.13 (b) shows the total number of committed assertions as ratio of the total number of committed narrow operations in the non-optimized stream. *SR-CS* is able to reduce the number of assertions by about 39%. Overall, LPP with *SR-CS* reduces the committed stream by 20%.

Effect on Number of Cycles. Figure 5.14 shows the number of cycles taken by each benchmark to complete the cycle-accurate phase (the first 200m x86 user instructions using the *ref* input data-set). LPP with *SR-CS* achieves a reduction of 15.54%. One of the key factors influencing the gains obtained by LPP is the coverage of the optimized code. Some benchmarks suffer with low code coverage e.g., gap, gcc, and perlbmk. Although low code-coverage remains to be a caveat of our infrastructure – we profile for hot functions (barring recursive ones)

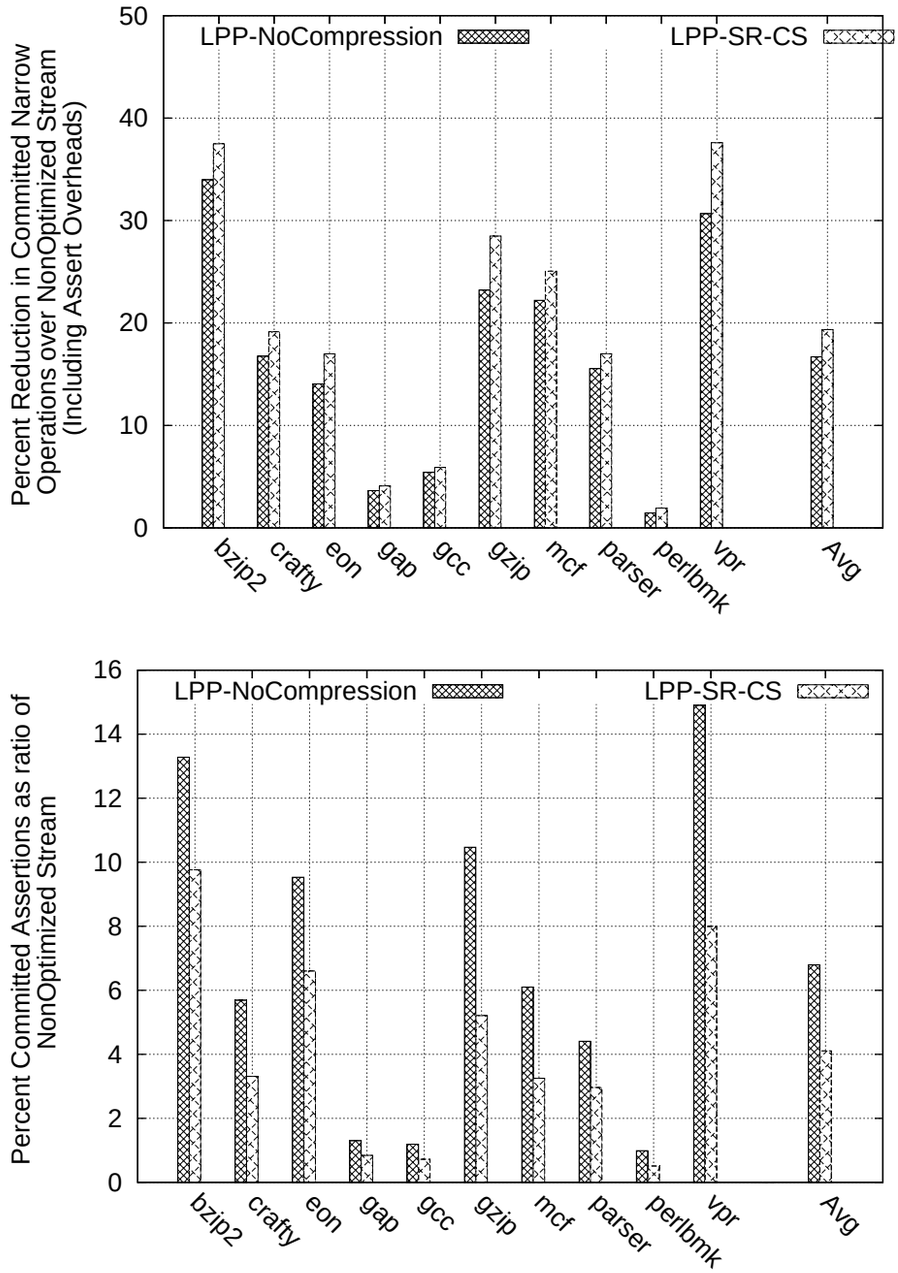

Figure 5.13: LPP in a static optimizer model - (a) Reduction in committed stream (b) Overheads as ratio of non-optimized stream

and then choose the basic blocks, nothing prevents the optimization of all the code. vpr and bzip2, the model benchmarks with an average code coverage of > 95%, achieve significant reduction in the total number of cycles – an average of 32%. Lastly, mcf (in spite of high coverage) fetches low gains with LPP because it has high percentage of memory instructions (Group3), which LPP does not optimize.

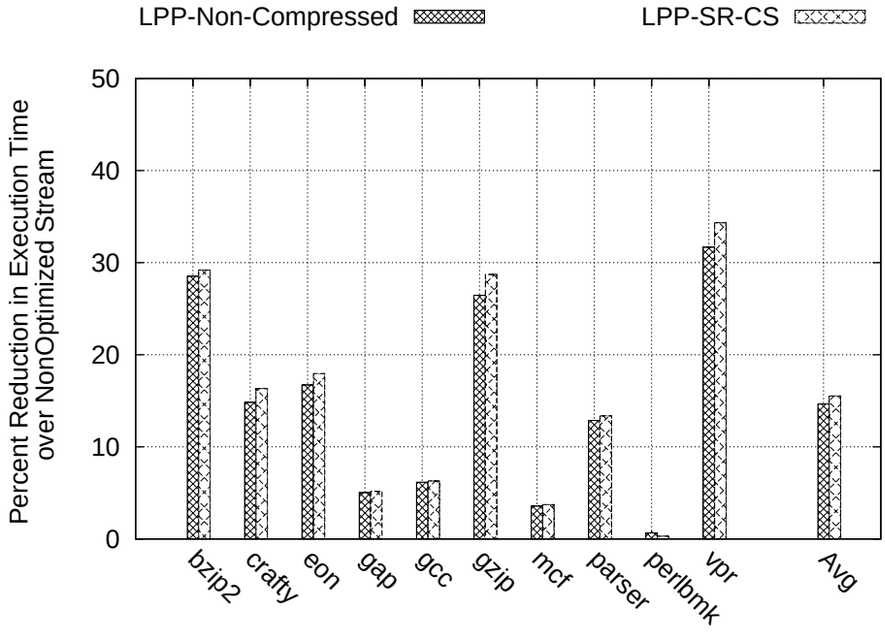

Figure 5.14: LPP in a static optimizer model - Effect on number of cycles

Conclusion. LPP optimization with single-register based compression scheme (LPP-SR-CS) in a static optimizer model achieves a performance improvement of 15.54% over non-optimized stream of narrow computations. Further, about 20% reduction in the number of committed narrow computations is achieved. On an average, higher assertion failure rate (than dynamic optimizer model) of 2.52% is observed. Finally a lower coverage (than dynamic optimizer model) of about 60% is seen. In perspective, the evaluations indicate that LPP in as a *static optimization* offers a good design point as the loss of performance when compared to a dynamic optimizer model is acceptable. The advantage of the static optimizer model is that the applications do not need to bear the run-time overheads of profiling and optimization.

5.4.4 Atomicity : Basic Block vs. SuperBlock

Another design decision that may impact the overall performance of LPP is the *size of atomic regions*. As introduced in Section 5.2.3.4, the assertions are merged inside the region via following producer-consumer relationships [1]. Conceptually, (in the context of LPP) the larger the atomic region, the more the opportunities to merge assertions and hence, to reduce the assertion overheads.

The evaluations presented in previous sections have been performed on basic blocks as atomic regions. In order to explore the design space with regard to the size of atomic region, we now evaluate superblocks. More details on the definition and configurations of the superblocks have been mentioned in Section 3.1.1.1. As superblock generation is profile-based, we evaluate the sensitivity of LPP with respect to size of atomic region in a *dynamic optimizer model*.

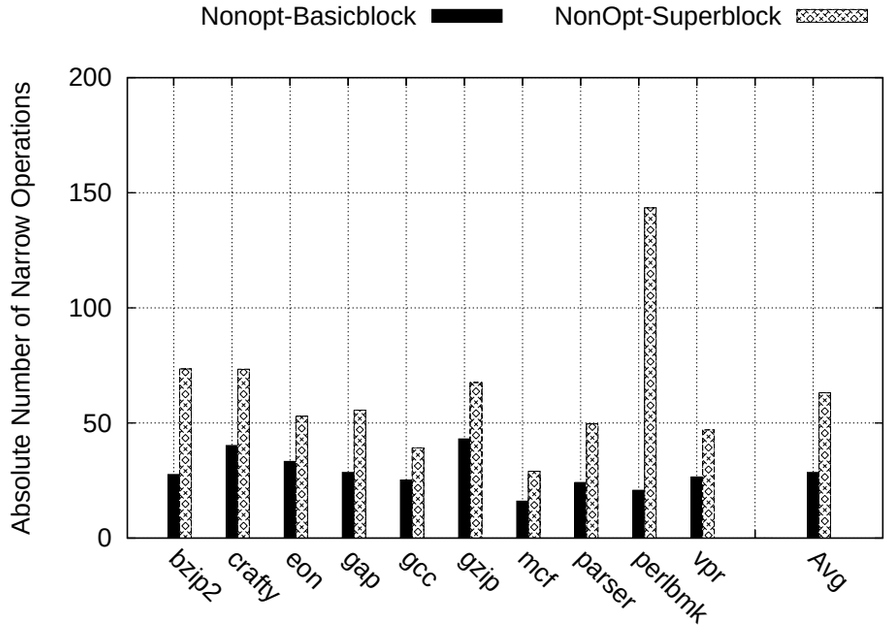

Figure 5.15: Size of atomic regions

Size of Atomic Regions. Foremost, we quantify the increase in the size of regions by considering superblocks. Figure 5.15 shows the average dynamic size of both the non-optimized basic blocks and the non-optimized superblocks for each application. The size is measured in terms of the absolute number of narrow operations and represent the weighted average across

[1] Hence, it is necessary for correctness that the atomicity of the region be ensured

all the dynamically committed regions in the profile-phase of the applications. On an average, the size of a superblock is about 2.2x of that of the size of a basic block.

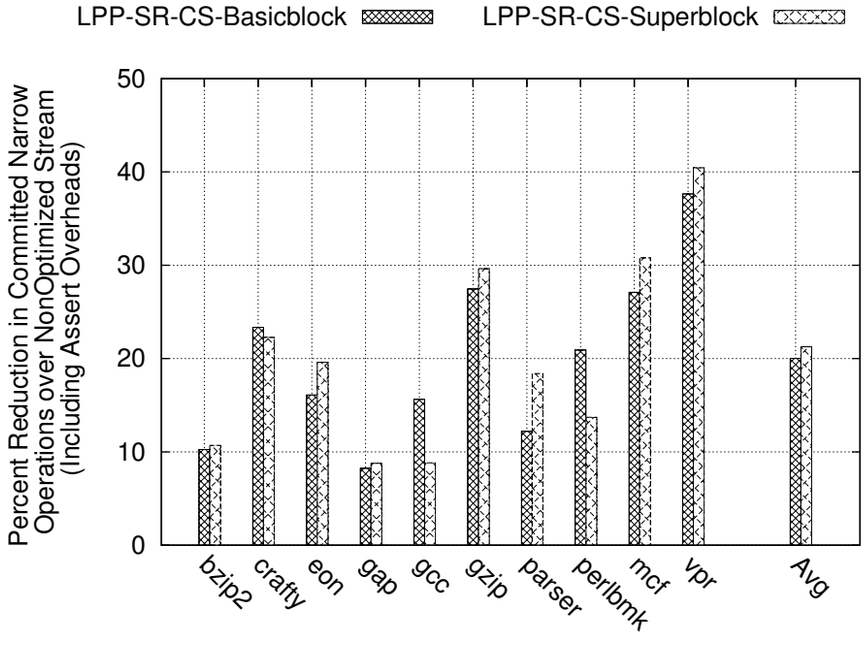

Figure 5.16: Impact of the size of atomic regions - Reduction in computations comparing basic block vs. superblock

Overall Reduction in Computations. Figure 5.16 shows the overall reduction in the number of narrow computations committed in the cycle-accurate phase of the applications (with *ref* input data-sets as the evaluations are in a dynamic optimizer model). Seven (bzip2, eon, gap, gzip, parser, mcf, and vpr) of ten benchmarks show improvements when the atomic region size is increased to a superblock. However, gcc and perlbmk show marked degradation even though the static number of computations in an optimized superblock has been observed to be less than the static number of computation in an optimized basic block (evaluations avoided for sake of brevity). This degradation is due two reasons :

(i) The superblocks are treated as single-entry, single-exit regions, and on an event of early exit, the speculative region is squashed and non-optimized version of the region is executed. Hence, this leads to additional failures apart from the assertion failures.

(ii) Further, it is also possible to have more assertion failures in case of speculative superblocks, when compared to speculative basic blocks.

Both these factors lead to a overall decrease in the achieved coverage of LPP (ratio of committed computations from the optimized regions over the total committed computations) in case of gcc and perlbmk (8% and 23% respectively). On an average, LPP on superblocks furnishes 1.25% more reduction in the committed narrow operations.

Conclusions. Evaluations suggest that on an average, LPP when applied on superblocks as atomic regions furnishes similar gains as when applied on basic blocks. Superblocks deteriorate the performance in two of ten benchmarks. In general, a basic block ensures sufficient balance between the opportunities to merge assertions, and the implicated costs due to assertion failures (we squash the complete region on assertion failure). Given the simplicity of basic blocks, we believe LPP on basic blocks presents a good design point.

5.4.5 Effects on Instruction Scheduling

Instruction scheduling is an important code optimization technique to obtain high performance and better resource utilization on a parallel (pipelined or superscalar) processor. Since narrow stream has more tasks of finer granularity, it furnishes more opportunities for scheduling techniques to achieve better resource utilization and performance improvement.

We evaluate the code scheduling opportunities with the help of a simple list-scheduling algorithm – *Earliest-start Time Slack* (abbreviated as ETS) based scheduling. The ETS heuristic based scheduling focuses on two targets – stall time, and critical path. The ETS instruction scheduling is based on the classical greedy list-scheduling algorithm [21].

ETS scheduling is performed at the basic block level with the target of reducing the overall execution time. Each basic block is represented using its data dependence graph DDG(R). Next, static priorities to each node in the DDG(R) are assigned. These static priorities are essentially a set of two scores : *Earliest-start Time* and *Slack*. The candidate list [21] is a sorted list of eligible instructions for scheduling : Earliest-start time as the first sort key, followed by slack.

Slack of a node is defined as the difference between the latest-start time and the earliest-start time of the node. This is a greedy algorithm towards instruction scheduling whereby the overall execution time is reduced (by sorting nodes in increasing order of their earliest-start time), while prioritizing the instructions in the critical path of the DAG as the second priority (by sorting those nodes that have the same earliest-start time in increasing order of their slack) [1].

Figure 5.17 shows how ETS scheduling can impact the overall execution time using the same example seen previously in this chapter. It is important to note how LPP not only removes

[1] Branch instruction remains the last instruction always

computations, but also decreases the critical path of narrow computations. The critical path length of the DDG in (B) is 7, whereas that of DDG in (C) is 5.

Renaming of Flags. Considering false data dependences for flags can be overly restraining for instruction scheduling. For example, recall that a 64-bit wide add operation is broken down into a flag-dependent chain of narrow add/addc operations (Section 2.2.2) where each narrow computation generates the carry flag for the next. To avoid limiting the opportunities for instruction scheduling in wake of such flag dependences, we assume a flag management scheme similar to [62], together with explicit referencing of flags in instructions which can be managed with a 3 source-operands encoding semantics of the proposed narrow ISA.

Discussion. With the help of the ETS scheduling scheme, we only intend to affirm our proposition that the narrow stream of computations offers significant opportunities for code scheduling as it has more tasks of finer granularity. Instruction scheduling also furthers the benefits provided by LPP because LPP may as well prune computations on the critical path; which allows the child narrow computations to be scheduled freely. This is substantiated by the results as shown in Figure 5.18 which clearly show that code scheduling provides more performance improvement ($\sim$ 26%) when realized over LPP optimized code than when applied on the non-optimized narrow stream of computations ($\sim$ 19%).

Evaluation : ETS scheduling with LPP. Figure 5.18 illustrates how ETS scheduling interacts with the stream of narrow computations. Simply applying ETS code scheduling on the non-optimized narrow stream (see Nonopt-Scheduling) achieves an average speedup of about 19% (with as high as up to 40 % for bzip2). This result underlines our hypothesis that narrow stream offers more freedom to schedule operations. Recall that code scheduling is done at a basic block level. Further, LPP with *SR-CS* achieves a reduction of 15.54% by itself, and when combined with code scheduling, cumulative reductions of about 26% in the number of cycles can be achieved.

5.4.6 Comparison with 64-bit In-Order Pipeline

To put things in perspective, Figure 5.19 (a) illustrates the overall performance of the narrow processor as compared to a 64-bit wide processor. The first two bars (16bit-HotCode-NonOpt and 16bit-HotCode-LPP+Sched) show the relevant data for the *hot regions* in exclusion. Next, the latter two bars (16bit-Program-NonOpt and 16bit-Program-LPP+Sched) show the behavior of the complete program (200m user commits) where only a part of the program (hot-code) is optimized. More specifically –

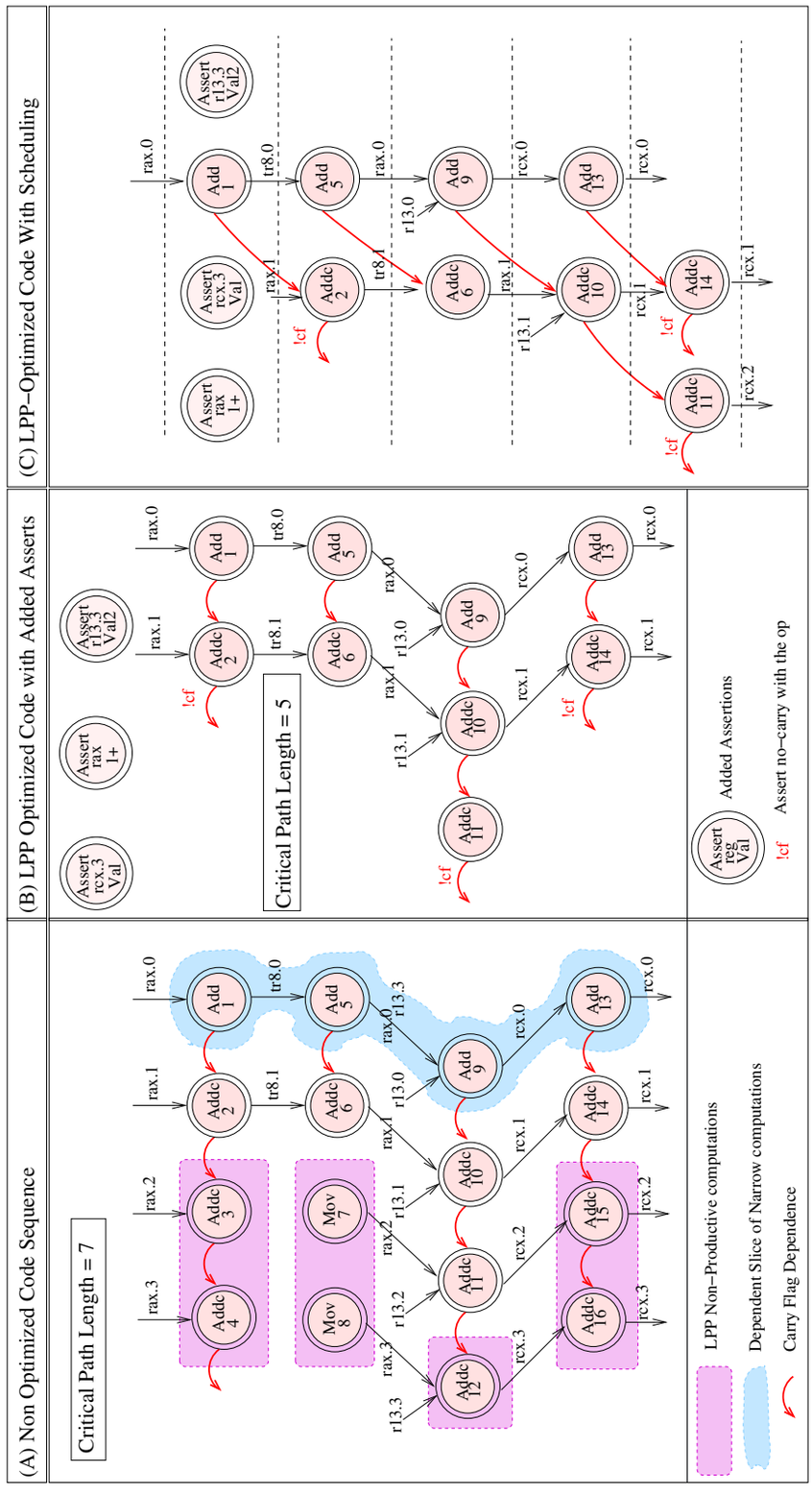

Figure 5.17: Incorporating code scheduling with LPP on a sample code sequence from vpr

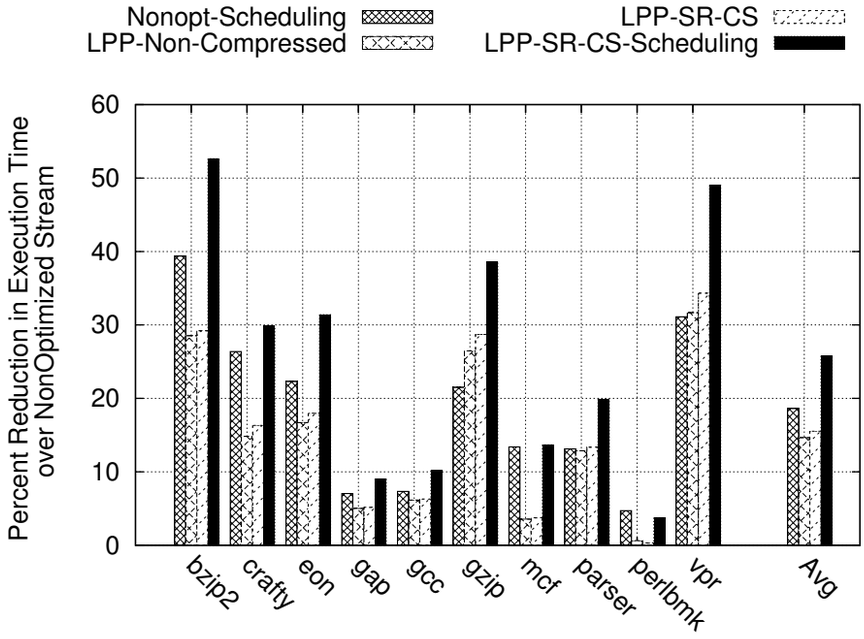

Figure 5.18: Effect on number of cycles with LPP and code scheduling

(i) *16bit-HotCode-NonOpt* shows the ratio of the number of cycles for the execution of the non-optimized narrow stream over 64-bit version for the *hot regions* in exclusion.

(ii) *16bit-HotCode-LPP+Sched* shows the ratio of the number of cycles for the execution of the LPP-SR-CS optimized and ETS scheduled narrow stream of computations over 64-bit version for the *hot regions* in exclusion.

(iii) *16bit-Program-NonOpt* shows the ratio of the number of cycles for the execution of the non-optimized narrow stream over 64-bit version for the the complete program (200m user commits).

(iv) *16bit-Program-LPP+Sched* shows the ratio of the number of cycles for the execution of the LPP-SR-CS optimized and ETS scheduled narrow stream of computations over 64-bit version for the complete program (200m user commits) where only a part of the program (hot-code) is optimized.

The results indicate that using the two optimizations (LPP and ETS scheduling), the performance penalty of the narrow ISA can be potentially reduced from 2.3x to 1.38x, considering *hot regions* in exclusion. Further, the latter two bars (16bit-Program-NonOpt and 16bit-Program-LPP+Sched) show the behavior of the complete program (200m user commits) where only a

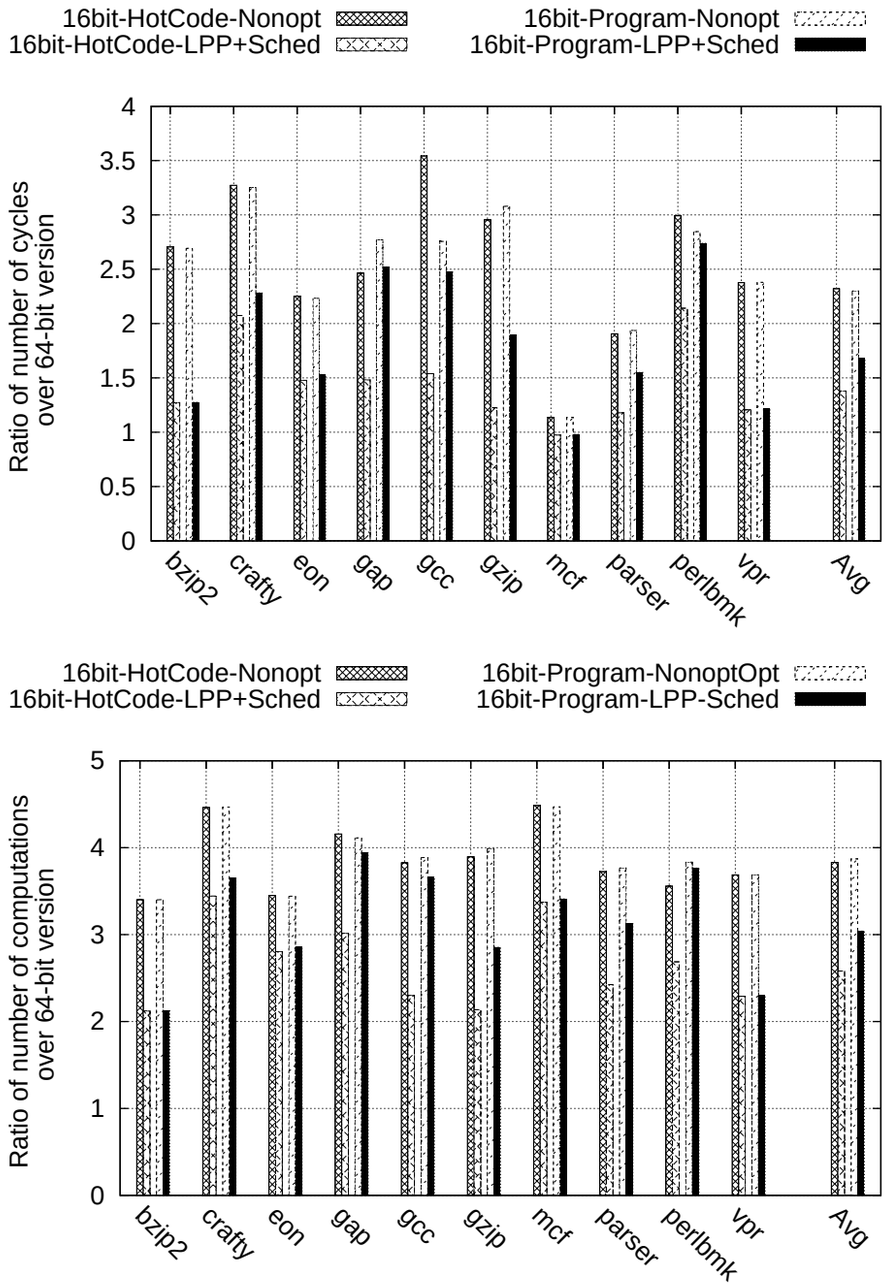

Figure 5.19: LPP as a static optimization on basic blocks : Comparison with 64-bit execution (a) Cycles (b) Committed computations)

part of the program (hot-code) is optimized. As can be seen, LPP and code scheduling can reduce the penalty of the narrow ISA from 2.3x to 1.65x (with a code coverage of about 57%).

Similarly, Figure 5.19 (b) illustrates the dynamic code footprint of the narrow ISA programs as compared to a 64-bit wide programs. Similar to Figure 5.19 (a), the different bars show the behavior for both *hot regions* in exclusion and for the complete program (200m user commits) where only a part of the program (hot-code) is optimized. It can be seen by comparing the 16bit-HotCode-LPP+Sched configuration against the 16bit-HotCode-NonOpt configuration that LPP can potentially reduce the dynamic code size from 3.9x to 2.68x (32 % reduction).

5.4.7 Observed Roadblocks

The disposable potential for LPP assuming perfect, advance knowledge of data values and assuming that no assertion-like instructions are required, has been shown to be around 44% when NPR_Threshold is assumed to be 95% (evaluations have been provided in Section 2.4.3.2). These computations include all type of narrow computations – Group0, Group1, Group2 and also Group3. As LPP is applied on the complete dynamic stream of computations, the coverage of the LPP in this experiment is 100%.

The achieved reduction in narrow computations by LPP in a static optimizer model, with basic block as atomic region, has been shown to be around 20% with an achieved code coverage of around 60%. The following have been observed to be the reasons why LPP in practice achieves less benefits :

(i) LPP does not optimize memory operations. As illustrated in Figure 5.3, Group3 account for 20% of the dynamically non-productive computations.

(ii) LPP has been performed on basic blocks of hot functions only. This is mainly a caveat of our infrastructure and nothing prohibits from extending LPP to further hot basic blocks. An achieved coverage of 60% has been observed. On the other hand, the experiments to evaluate the disposable potential analyze all the code, and hence a 100% code coverage. Further, as we illustrate in the evaluations in Section 5.4.6, isolating the benefits of LPP on hot functions only reveals that LPP reduces the narrow code stream by 31.28% [1].

(iii) The disposable potential study reflects a no-overhead, best-case potential. LPP does incur not only assertion overheads (an average of 4% additional computations), but also assertion failure overheads in the form of region squash and execution of non-optimized code.

[1] useful evaluation to correlate with 100% coverage of LPP

We believe that taking the aforementioned issues into account, the performance of LPP is close to its disposable potential.

5.5 Conclusions

This chapter proposes and evaluates Local Productiveness Pruning (LPP). LPP is a profile-based, aggressive, speculative optimization technique, which prunes narrow computations based on their productiveness index, when viewed in isolation. It also embedded checks in the code stream to detect any unassumed case of dynamic productiveness.

Additional hardware is required to implement LPP apart from the baseline narrow processor with support for speculation recovery. This includes additional assertion-like opcodes and support for 3-bit encoding for each 64-bit register.

Compared to GPP, the rationale for LPP is to study whether productiveness on a finer granularity of individual computation is more powerful. The atomic region, however, cannot be a single computation; atomic regions are assumed to be basic blocks to reduce the assertion overheads of LPP. In this chapter, we explored the design space of the LPP optimization by evaluating static vs. dynamic optimization models. Further, altering the size of atomic regions has also been investigated. Based on the evaluations, we believe LPP in a static optimizer model with basic block as atomic regions presents a good design point.

LPP can reduce the dynamic code footprint of the narrow ISA by 20%, with around 15% reduction in number of cycles, and an overall coverage of about 60%. Assertion overheads are admissible : 4% additional narrow computations using single-register compression scheme. Lastly, assertion failure rates are as low as 2.52% on an average.

Unlike GPP, LPP shows that the heuristic of productiveness can be effective in reducing the dynamic code footprint of the narrow ISA. Given the low gains of GPP, together with other conceptual differences (like size of atomic regions) between GPP and LPP, we do not combine the two techniques together.

6

Minimal Branch Computation

The optimizations proposed until now (Global Productiveness Propagation and Local Productiveness Pruning) are heuristic-based optimizations aimed towards generating the *minimum required computations* (MRC) of a region via data-flow analyses and code pruning. In this chapter, we extend the notion of MRC further and introduce a different approach to reduce the dynamic code footprint of the narrow ISA.

Code reordering is a well-established technique and is used in traditional systems for achieving better resource-utilization, hiding memory latencies, improved cache performance and throughput. In this chapter, we introduce the concept of *reordering narrow backslices* so as to reduce the dynamic code footprint of the narrow ISA applications. The idea is to reorder the backslices containing narrow computations such that the minimal necessary computations to generate the same (correct) output are performed in the *most-frequent case*; the rest of the computations are performed only when necessary.

In this chapter, we propose and evaluate a particular use case of the broader concept of reordering narrow backslices : the use case of reordering narrow backslices around *conditional branch computations*. This is because it has been observed that conditional branches and their backslices have specific properties that can be leveraged easily for this strategy. Conceptually though, the strategy of reordering narrow backslices is broader in its impact than that explored in this chapter and can as well be applied to computations apart from conditional branches.

The code optimization technique that reorders backslices containing narrow ISA computations around conditional branch computations is termed as the *Minimal Branch Computation* (henceforth, MBC) optimization. The MBC optimization technique is very specific to the narrow stream of computations; it truly exploits the fact that narrow ISA decomposes the traditional quanta of work (32-bit / 64-bit computations) into tasks of finer granularity in the form

of narrow ISA computations. Hence, MBC in its current form may not be applicable to wide 64-bit computations.

This chapter is organized as follows. First, it introduces the concept of the MBC optimization and presents the basic know-how for understanding the strategy of the optimization. After discussing the background definitions used in the rest of the chapter, the MBC optimization is formally defined. The proposed optimization is then described in detail with the individual passes composing the optimization technique. To the best of our knowledge, the proposal in this chapter is the first to exploit narrow computations for reducing the cost of resolving a conditional branch. Nevertheless, Section 6.5 highlights some previous research performed for reducing the overheads of branches. Finally, the chapter is concluded with the performance evaluation of the proposed optimization technique.

6.1 Introduction

The underlying motivation for the MBC optimization is to exploit the fundamental distinction between a flag [1] and a data value : flag reflects a *property* of the data value in a storage location. Succinctly speaking, flags have two useful attributes :

(i) A flag is a 1-bit status bit which may be computed *without the knowledge of the precise and complete value* of the corresponding multiple-bit length data value, and

(ii) Computing a flag can be done in *more than one ways*. This is because for computing the value of a flag, the correct and complete data value is not always necessary; sometimes a subset of the data value is sufficient.

For example, consider a 64-bit logical *and* operation : *and tr0 = rax, rbx*, which performs a logical *and* operation of two 64-bit values, rax=0x2aab|7b00|1000 and rbx=0x1111|0000|0001 to check if specific bits of rax are set or not, and updates the ZF accordingly. For dynamic instance of the computation, it is sufficient to perform the logical *and* operation on only $Chunk_2$[2] (0x2aab with 0x1111) to generate the correct value of ZF (instead of performing logical *and* on all four chunks – $Chunk_0$, $Chunk_1$, $Chunk_2$, and $Chunk_3$). Hence, in the narrow ISA programs, there exists an opportunity to compute the flag bits minimally. Exploiting these properties, the key responsibility of MBC is to infer how to generate the flag values with the minimal required computations. More formally,

[1] also known as condition code value or status-bit in some ISAs

[2] $Chunk_2$ is the 16bit data from bit positions 32 to 47

Given a series of computations responsible for generating only the flags which are eventually consumed by a conditional branch to determine the direction of program execution, what is the minimal necessary subset of these computations which can generate the same correct value of the required flags.

Next, we describe the basic know-how required to understand the MBC optimization. Although MBC is independent of the baseline ISA, this thesis assumes x86 / IA64 [26] based semantics with respect to flags / condition codes. A brief overview of the narrow ISA flag semantics has been previously provided in Section 2.2.2. The two aspects of narrow ISA that are relevant in the context of this chapter are further explained next – the EFLAGS register, and the condition codes.

EFLAGS Register. The x86 / IA64 ISAs have a 32-bit EFLAGS Register [25] which consists of a group of status flags, a control flag, and a group of system flags. This MBC optimization (and hence, this chapter) concerns only the status flags. There are six status flags in the x86 / IA64 ISA, namely : Carry Flag (CF), Parity Flag (PF), Adjust Flag (AF), Zero Flag (ZF), Sign Flag (SF), and Overflow Flag (OF).

Of these aforementioned flags, only the CF, ZF, OF and SF are most-commonly used. This is corroborated by our preliminary evaluations performed to refine the design space of the MBC optimization (presented later in Figure 6.2). Hence, the MBC optimization is built around only these four status flags (illustrated in Figure 6.1) :

(i) *Zero Flag* indicates whether the result generated by the computation is zero or not. ZF=1 indicates that the result is zero and ZF=0 indicates that the result is non-zero.

(ii) *Sign Flag* is the most significant bit of the result. This is the sign bit of a signed integer. SF=0 indicates positive value and SF=1 indicates a negative value.

(iii) *Carry Flag* is the bit that flows out of the sign bit (carry out of the sign bit).

(iv) *Overflow Flag* indicates whether a computation produces a result which is out of bounds for the corresponding data-type. The overflow flag is usually computed as the *xor* of the carry into the sign bit and the carry out of the sign bit. In other words, exclusive-ORing the carry flag and the bit carried into the sign bit (last bit of the destination operand) generates the Overflow Flag.

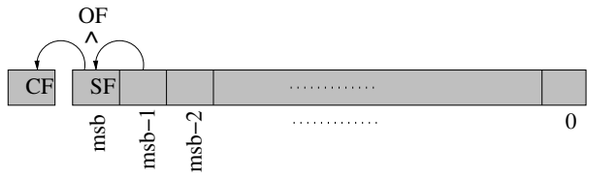

Figure 6.1: Semantics of selected flags - sign flag, carry flag, and overflow flag

Condition Codes. The narrow ISA implements *Jcc* – 'Jump if Condition is Met' operations. Much of the narrow ISA *Jcc* operations remain the same as that for x86 / IA64 [25]. A condition code is associated with each of these instructions to indicate the condition code being tested for. Few examples of condition codes are E/NE (equal or not equal), Z/NZ (zero or non zero), LE/NLE (less than equal or not less than equal), BE/NBE (below or equal, or not below or equal), L/NL (less than or not less than) etc. As both E/NE, and Z/NZ condition codes check the same flag bit (i.e., zero flag ZF), for the rest of the chapter the two are often used interchangeably.

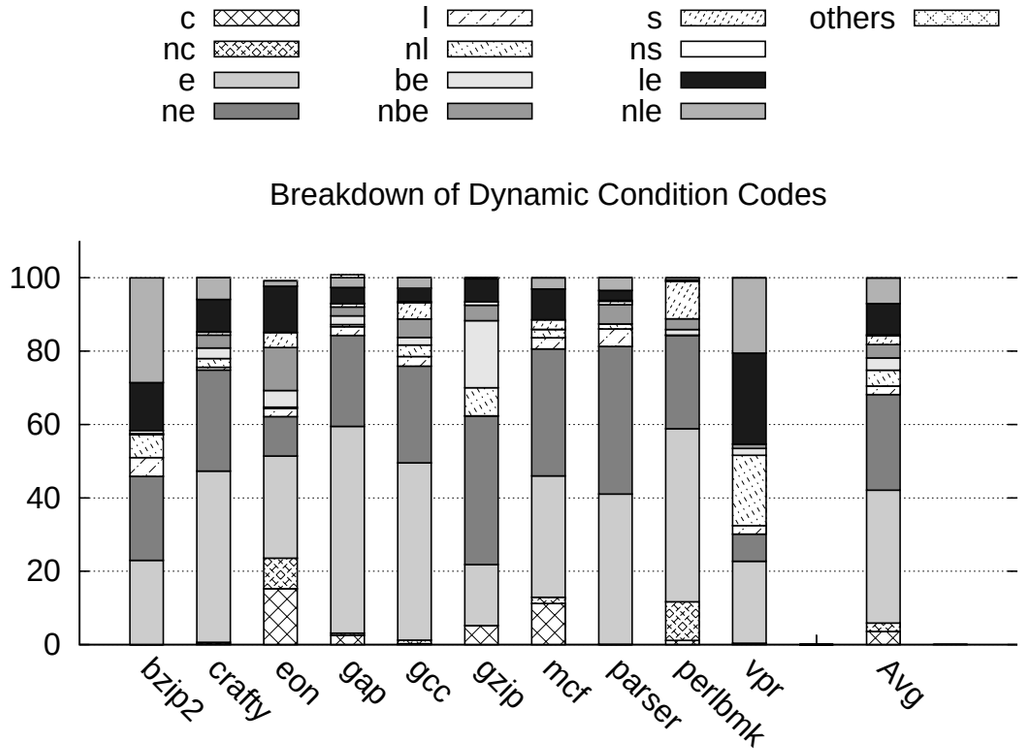

Figure 6.2: Histogram distribution of dynamic condition codes for conditional branches

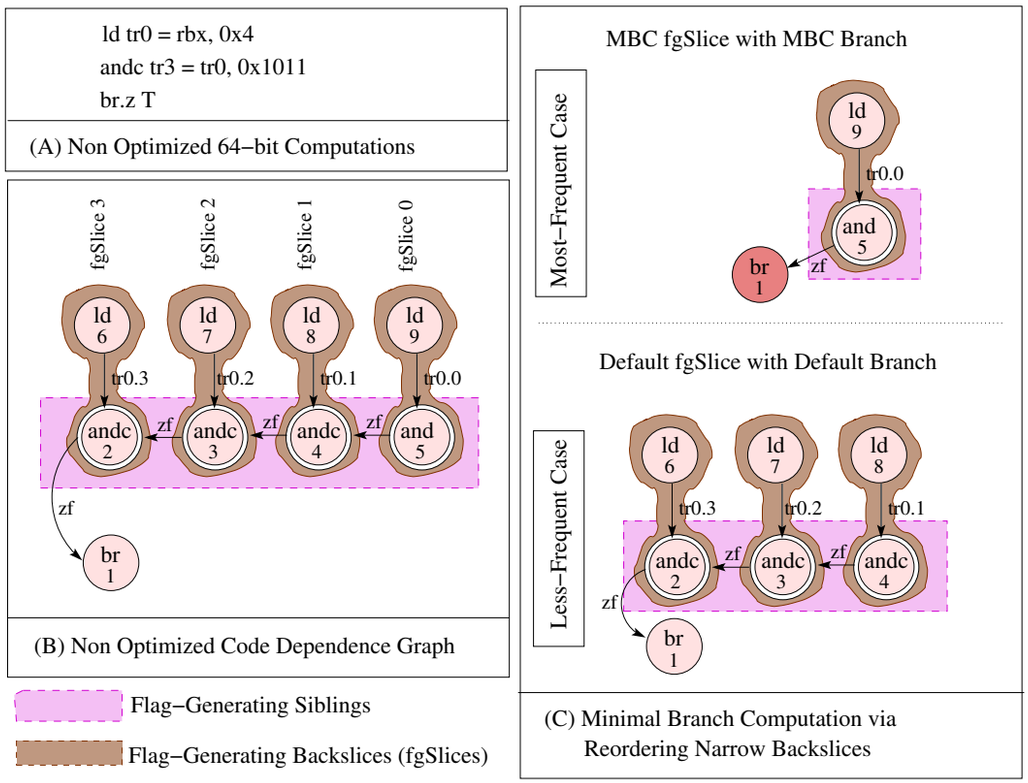

Figure 6.3: Background definitions and minimal branch computations conceptually

Dynamic Condition Codes Distribution. Figure 6.2 shows the histogram distribution of the dynamically seen condition codes for all committed conditional branches for the profile-phase (200m committed user instructions using the *ref* input data-set) of the respective applications. As the graph illustrates, about 62.28% of the conditional branches are associated with e/ne (evaluated based on value of ZF), another 15.46% with le/nle (evaluated based on the values of ZF and OF), and 6.48% with be/nbe (evaluated based on the values of ZF and CF). Together, these six condition codes can provide acceptable coverage for MBC : a total of 84.22%.

Hence, MBC optimization, in its current form, optimizes only the six foregoing condition codes. This is done to merely balance the trade-off between the implementation effort required vs. the harvested gains; nothing prevents from adding the rest of the condition codes to achieve 100% coverage across the complete set of condition codes of the narrow ISA.

6.2 Definition

Informally, MBC is a profile-based technique that reorders narrow computations around conditional branches so that the condition code required for the conditional branch be computed using the minimal necessary computations.

Figure 6.3 diagrammatically shows the overall mechanism of MBC using a sample code from a basic block. Only those instructions that are used to generate the value of the condition code are shown in Figure 6.3 (A). In nutshell, the branch is taken if the value loaded from memory (location rbx+0x4) has three specific bits set to zero. Figure 6.3 (B) shows the data dependence graph of the narrow ISA operations. Finally, Figure 6.3 (C) shows what MBC can potentially achieve. MBC uses a profile-based learning approach to infer which set of computations is sufficient *most of the times* to generate the correct value of the flag (ZF in this case). It reorders code in order to exploit this property to reduce the overall dynamic code footprint of sample piece of code.

Apart from the previously described background definitions and terminologies used in this thesis (Section 2.4.1), next section lays downs some more definitions which are used to formally define and describe the Minimal Branch Computation optimization.

6.2.1 Background Definitions

Refer to Figure 6.3 for a diagrammatic view of the following definitions.

Definition 6.1 (**Flag-generating Siblings**). Those narrow computations which are decomposed from the same RISC-like, wide 64-bit computation are called *Flag-generating Siblings* when they co-operatively compute the flags consumed by an impending conditional branch in the static stream of narrow computations.

Hence, each of the flag-generating siblings has a data dependence due to the flag register with its predecessor flag-generating sibling in program order (except that sibling which generates chunk zero). Consider the code sequence shown diagrammatically in Figure 6.3 : computations 2, 3, 4, and 5 are flag-generating siblings. Following the x86 semantics, the OF and CF flags are cleared for a narrow *and / andc* operation; the SF, ZF, and PF flags are set according to the result. Hence, note that there is a flag dependence (zero flag) that flows in the same order.

Definition 6.2 (**Data-flow Computations**). Given a code region R and its data dependence graph DDG(R), those computations which are the live-outs of R, and their backslices are de-

fined as *Data-flow Computations* of the region R. The live-out analysis is performed for all narrow architectural registers except instruction-pointers (rip.0, rip.1, rip.2, and rip.3) and flags.

In the context of the MBC optimization, the code region R is also the optimization region (Definition 2.3). In this thesis, MBC has been evaluated on a basic block, although other granularities like a superblock can be explored too.

Definition 6.3 (**Conditional Control-flow Computations**)**.** Given a conditional branch, *Conditional Control-flow Computations* is the group of computations consisting of all the flag-generating siblings and their backslices such that the backslices contribute to generating the flag values only (and do not contribute towards the generation of a live-out data value).

Definition 6.4 (**Reorderability of Flag-generating Siblings**)**.** The flag-generating siblings are said to be *Reorderable* if altering the order of these instructions never impacts the outcome (the evaluated flag). The flag-generating siblings 2, 3, 4 and 5 in Figure 6.3 are *Reorderable*.

Table 6.1: Sample code to illustrate non-reorderability of flag-generating siblings

Sample Translation for a Conditional Branch	
Wide Operation (64-bit)	Equivalent set of Narrow Operations
sub tr0 rax = rax.0 imm.0	sub tr0.0 = rax.0 imm.0 subc tr0.1 = rax.1 imm.1 subc tr0.2 = rax.2 imm.2 subc tr0.3 = rax.3 imm.3
	mov inrip.0 = riptaken.0 mov inrip.1 = riptaken.1 mov inrip.2 = riptaken.2 mov inrip.3 = riptaken.3
br rip = cf, riptaken	br rip = cf, inrip;
Register name R_i refers to the ith chunk (16-bit data) of the equivalent 64-bit register register R, R_0 holds the least significant chunk. Immediate name Imm_i refers to the ith chunk of the equivalent 64-bit immediate, Imm_0 specifies the least significant chunk. inrip is an additional 64-bit internal register used for redirection of control-flow due to branches.	

On the other hand, consider the code sequence shown in Table 6.1. The flag-generating siblings are the four subtract operations. These computations are *Non-reorderable* as subc

semantics require that the value of carry flag from the dependence is used as input to generate the current updated value of the flag (CF).

Definition 6.5 (**Flag-generating Backslice of a Sibling (*fgSlice*)**). *Backslice of a Flag-generating Sibling (fgSlice)* is the static backslice (computed by a backward-traversal) of the flag-generating sibling such that :

- it does not contain any data-flow computation, and
- flag dependences between the flag-generating siblings are followed only when they are *non-reorderable*.

Hence, an fgSlice gives those narrow computations which are responsible for *generating the flags only* (directly or indirectly).

6.2.2 Definition of MBC

Definition 6.6 (**Minimal Branch Computation**). *Minimal Branch Computation* is a profile-based code reordering technique which places those flag-generating backslice(s) *first* which is(are) *most-probably sufficient* to generate the outcome of the condition code which is eventually consumed by the impending conditional branch computation.

Further, following must be asserted in the context of MBC :

1. Scope : MBC assumes that all branches – conditional and unconditional are always necessary. It does not 'optimize the control-flow' as done by some previous research [31, 60]; it only *reduces the control-flow backslices* of conditional branches.
2. Effect : To reduce the control-flow backslice, MBC does not change a flag-generating computation by a computation of lesser strength. MBC simply *reorders* the original narrow ISA computations around conditional branches.
3. Program equivalence with respect to flags : MBC optimization may forsake the strict guarantee of program equivalence with respect to flags selectively. Recall that the narrow ISA operations may update multiple flags as their side-effect of execution (similar to x86)[1]. This redundancy can limit the applicability of the MBC optimization. Further, given the redundancy, it has been observed that most of the generated flags are

[1]Assuming similar flag semantics for narrow operations as x86 allows simple semantic specification of the narrow ISA

dead (written before use) or are not used at all. Hence, the MBC optimization *does not guarantee precise state of dead flags*.

Hardware Support. Unlike the previously studied non-productiveness based pruning techniques (GPP and LPP) in this thesis, MBC optimization technique does not require any additional hardware support. However, in logical progression of the thesis, we propose and evaluate MBC on an LPP-optimized narrow computation stream. Hence, the hardware support required for MBC is at least the same as that required for LPP. Recall that this mainly implies hardware-based speculation rollback and recovery methods (discussed previously in Section 3.2).

Comparison with Non-productiveness based Pruning Techniques. In perspective with the non-productiveness based pruning techniques previously studied in the thesis, the MBC optimization technique is profile-guided too. The difference, however, is in the how self-sufficient the generated code is. The code generated after non-productiveness based pruning techniques is speculative and in wake of assumption failure, hardware mechanisms are required to perform a rollback and recovery action. The MBC generated code, on the other hand, is self-sufficient to generate the correct state without the need of a rollback. MBC reordering embeds a sort of compensation code (known as the default backslice) which in the event of mis-speculation, if any, can overwrite the previous partial (and probably incorrect) state and can generate the correct state of the required flags always.

6.3 Description

6.3.1 Overview

MBC is a profile-based code reordering technique. Figure 6.4 depicts the overall flow of the proposed optimization. As the first step of the profile-generation process, *hot regions* are detected. Next, all conditional branches and the corresponding flag-generating siblings are inferred. The flag-generating siblings are then profiled. The profiling strategy of MBC is more elaborate than that required for the non-productiveness based pruning techniques described in previous chapters. At the same time, MBC profiles only the flag-generating siblings, and hence, the overhead of profiling for MBC as compared to the previously studied code pruning techniques is much lower. MBC requires profiles for certain *sufficiency rules* for each dynamic instance of a condition code. The sufficiency rules are highly specific to the associated condition code. Details on profiling required for MBC and the sufficiency rules for each selected condition code are provided in Section 6.3.2.

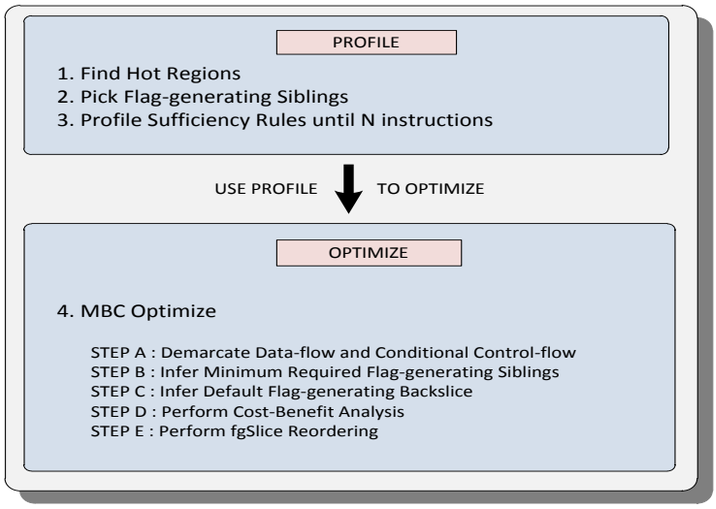

Figure 6.4: Minimal Branch Computation : An overview with the component passes

After the profile phase (of say N instructions), the MBC optimization is triggered using the collected profiles. The individual component passes of the optimization are described in the next section.

6.3.2 Profiling for MBC

The profiling strategy for MBC aims to infer which of the flag-generating sibling(s) is(are) most-probably sufficient to generate the correct value of the required flag. Table 6.2, Table 6.3 and Table 6.4 list the most common opcodes (JZ/JNZ, JLE/JNLE, and JBE/JNBE respectively) for conditional branches and their associated condition codes. Individual profiling strategy for each of the condition codes is described in the following paragraphs.

In summary, the two main types of profiles required for MBC optimization across the specified condition codes are –

(i) *Outcome Profile*, which measures how many times the condition code is evaluated as true vs. false, and

(ii) *Property Profile*, which measures specific properties of the data values involved in the computation of the corresponding flag.

Table 6.2: Conditional branch JZ / JNZ, the associated status flags, and the profiling strategy

Conditional Branch Opcode	Status Flags Required	Most-Common Flag-generating Opcodes
JZ / JNZ	ZF / !ZF	and(c), add(c), sub(c)
Profiling Strategy		
1. Outcome Profile : Taken %, Not-Taken %. 2. Property Profile : Zero-Profile or Nonzero-Profile is gathered, if ZF = 0, Nonzero-Profile is gathered $\equiv$ profile for which chunk is nonzero, if ZF = 1, Zero-Profile is gathered $\equiv$ profile for size-sign encoding of src1 and src2.		

Profiling for JZ / JNZ. The two most commonly used conditional branches are JZ, i.e., Jump if Zero (ZF=1), and JNZ, i.e., Jump if Not Zero (ZF=0). The most-common flag-generating opcodes used for JZ / JNZ conditional branches are *and*, *add*, and *sub* (in decreasing order of their use in the workloads evaluated in this thesis). Table 6.2 shows the profiling strategy for JZ / JNZ. Two main profiles are required to learn that operating on which chunk may be sufficient to infer the correct value of the ZF :

(i) *Outcome Profile* : For each static branch computation, this profile counts the direction, i.e. taken (TRUE) vs. not-taken (FALSE) for all its dynamic executions. This is later used to infer the bias of a particular branch towards being taken / not-taken. The Outcome Profile is also used to keep track of the execution count of the respective conditional branch, also referred to as the *ExecCntBranch* hereafter.

(ii) *Property Profile* : The Property Profiler applies a split strategy. For each static branch computation, if the ZF is evaluated to zero (which implies that at least one of the flag-generating siblings generated a non-zero data value in their respective destination registers), the *Nonzero-Profile* is performed. Nonzero-Profile counts which of the 16-bit chunks generated by the flag-generating siblings is non-zero, and hence, sufficient for the inference of ZF being zero.

If the ZF is evaluated to one (which implies that the all flag-generating siblings generated a zero data value in the destination register), the *Zero-Profile* is performed. Zero-Profile counts the bias of the involved 64-bit operand values of the flag-generating operation (src1 and src2 operands) towards a particular size-sign encoding. The size-sign encoding is the same as shown previously in Figure 5.5 (Chapter 5).

Following is the rationale for the profiling scheme chosen for JZ / JNZ (summarized in Table 6.2). If the ZF is evaluated to zero (i.e., the data value result is not numerically zero), it is sufficient to operate on the non-zero chunk only. For example, in case of logical *and* operation on two 64-bit values : src1=0x2aab|7b00|1000 and src2=0x1111|0000|0001, it is sufficient to perform logical *and* operation only on the $Chunk_2$ (0x2aab with 0x1111) to generate the correct value of ZF (ZF = 0 in this example). Note how, even though the profiling is performed on the narrow computation stream, the profiler creates a sense of the semantic 64-bit values for input and output data values by combining the profiles of all the flag-generating siblings.

Similarly, the rationale for Zero-Profile is intuitive. If the ZF is evaluated to one, (i.e., the data value of the result generated by the flag-generating siblings is numerically zero), it may be sufficient to assert on the size-sign encoding of the 64-bit values of src1 and src2 (by combining all the input data chunks of the flag-generating siblings), and operate on the minimum necessary chunks only. The notion of extending the 16-bit register file in a narrow bitwidth architecture with a 3-bit size-sign encoding per 64-bit data value have been previously discussed in Figure 5.5 (Chapter 5). For example, in case of logical *and* operation on two 64-bit values : src1=0x1000 (encoding 0+) and src2=0x0001 (encoding 0+), it is sufficient to : (i) assert dynamically that size-sign encoding of src1 and src2 is 0+ each, and (ii) perform logical *and* operation on $Chunk_0$[1] of both src1 and src2 only.

Profiling for JLE / JNLE. The second most-commonly used conditional branches are JLE i.e., Jump if less or equal (ZF=1 or SF≠OF), and JNLE i.e., Jump if not less or equal (ZF=0 and SF=OF). The most-common flag-generating opcodes used for JLE / JNLE conditional branches are subtraction based opcodes (sub and subc). Table 6.3 shows the profiling strategy for JLE / JNLE. Two main profiles are required to learn that operating until which chunk may be sufficient to infer the correct value of the LE condition code :

(i) *Outcome Profile* : Same as that explained for the case of profiling for JZ / JNZ previously.

(ii) *Property Profile* : Similar to the JZ / JNZ conditional branch, the Property Profiler for JLE / JNLE conditional branch applies a split strategy. For each static JLE / JNLE branch computation, if the LE (less than or equal) is evaluated to one, the *LE-Profile* is gathered. Further, LE-Profile is essentially a placeholder name for two set of profiles, and a value of one for the LE condition code implies that either of these two conditions must be true :

   (i) *src1 equal to src2*, in which case the profiler saves the size-sign encoding of the operands src1 and src2. This is known as the *Equals-Profile*.

[1] $Chunk_0$ is the least significant 16-bit chunk of bits 0 to 15

Table 6.3: Conditional branch JLE / JNLE, the associated status flags, and the profiling strategy

Conditional Branch Opcode	Status Flags Required	Most-Common Flag-generating Opcodes
JLE	ZF | (SF≠OF)	sub(c)
JNLE	!ZF & (SF=OF)	sub(c)
Profiling Strategy		
1. Outcome Profile : Taken %, Not-Taken %. 2. Property Profile : LE-Profile or Greater-Profile is gathered, if *LE is True, LE-Profile* is gathered : ⋇ if ZF = 1, Equals-Profile is gathered ≡ profile for size-sign encoding of src1 and src2, ⋇ else Less-Profile is gathered ≡ profile for which chunk of src1 is < respective chunk of src2, if *LE is False, Greater-Profile* is gathered ≡ profile for which chunk of src1 is > respective chunk of src2.		

(ii) *src1 < src2*, in which case the profiler updates the *Less-Profile*. Even though the profiling is performed on the narrow computation stream, the profiler creates a sense of the semantic 64-bit values for input and output data values. By scanning incrementally from the most significant chunk to the least significant chunks, the profiler saves that chunkID of src1 and src2 using which it is sufficient to draw the same logically correct inference (src1 < src2). For example, to infer whether the 64-bit value of src1=0x2aab|6b00|1000 is less than the 64-bit value of src2=0x2aab|7b00|1000, it is sufficient to compare only $Chunk_3$ (0x0000 and 0x0000), $Chunk_2$ (0x2aab and 0x2aab) and $Chunk_1$ (0x6b00 and 0x7b00). Hence, the profiler will save chunkID of one for this dynamic instance.

A zero value of LE implies that src1 must have been greater than src2. Hence, in this case, the profiler updates the *Greater-Profile*. By scanning incrementally from the most significant chunk to the lower significant chunks, the profiler saves that chunkID of src1 and src2 using which it is sufficient to draw the same logically correct inference (src1 > src2). For example, to infer that 64-bit value of src1=0x2ccc|7b00|1000 is greater than the 64-bit

value of src2=0x2aab|6b00|1000, it is sufficient to compare only $Chunk_3$ (0x0000 and 0x0000) and $Chunk_2$ (0x2ccc and 0x2aab). Hence, the profiler will increment chunkID of two for this dynamic instance.

Table 6.4: Conditional branch JBE / JNBE, the associated status flags, and the profiling strategy

Conditional Branch Opcode	Status Flags Required	Most-Common Flag-generating Opcodes
JBE	CF | ZF	sub(c)
JNBE	!CF & !ZF	sub(c)
Profiling Strategy		
1. Outcome Profile : Taken %, Not-Taken %. 2. Property Profile : BE-Profile or Greater-Profile is gathered, if *BE is True, BE-Profile* is gathered ⁎ if ZF = 1, Equals-Profile is gathered ≡ profile for size-sign encoding of src1 and src2, ⁎ else Below-Profile is gathered ≡ profile for which chunk of src1 is < respective chunk of src2, if *BE is False, Greater-Profile* is gathered ≡ profile for which chunk of src1 is > respective chunk of src2.		

Profiling for JBE / JNBE. The next most-commonly used conditional branches are JBE i.e., Jump if below or equal (CF=1 or ZF=1), and JNBE i.e., Jump if not below or equal (CF=0 and ZF=0). For the JBE / JNBE conditional branches, the most-common flag-generating opcodes observed via profiling are subtraction based opcodes (sub and subc).

JBE / JNBE is analogous to the previously discussed JLE / JNLE. The only difference is that JLE / JNLE account for signed arithmetic. Similar to the x86 / IA64 semantics, the terms *less* and *greater* are used for comparisons of signed integers and the terms *above* and *below* are used for unsigned integers. Table 6.4 shows the profiling strategy for JBE / JNBE (which remains very similar to JLE / JNLE shown in Table 6.3).

Finally, after the profile phase, the MBC optimization is triggered which eventually applies a cost-benefit analysis based on the profiles to reorder the flag-generating siblings and their backslices.

6.3.3 MBC Optimization

6.3.3.1 Step A : Demarcate Data-flow and Conditional Control-flow

The first step for MBC optimization is to demarcate the data-flow computations (Definition 6.2) and the conditional control-flow computations (Definition 6.3) in the region to be optimized.

It is important to separate the two so that the MBC optimizer can freely reorder the backslices in the conditional control-flow of the optimization region, without forsaking correctness. The data-flow computations are required outside the region and must always be generated. Note that the set of conditional control-flow computations of an optimization region may be empty even in presence of a conditional branch.

6.3.3.2 Step B : Infer Minimum Required Flag-generating Siblings

Given a conditional branch and its flag-generating siblings, in this pass the algorithm infers the *minimum sufficient* flag-generating siblings (using the profiles). The inference of the minimum sufficient flag-generating siblings is different for each condition code, much like how the profiling strategy varies with each condition code.

For sake of brevity, this section only discusses how to infer the minimum required flag-generating siblings for the JZ conditional branches (Jump if zero). Algorithm 3 shows the pseudo code for the same. The logical similarity between the profiling strategy (Table 6.2) and the means of inferring the minimum sufficient flag-generating siblings for the JZ conditional branch (Algorithm 3) is apparent.

If the bias of the particular JZ conditional branch is towards not-taken (lines 1 and 2 of Algorithm 3), this implies that the ZF had been set to zero most of the times. This, in turn, means that the Nonzero-Profiles of this conditional branch can be used to learn which of the flag-generating siblings had been generating a non-zero value most of the times.

On the other hand, if the bias of the JZ conditional branch is towards taken, this implies that the ZF had been set to 1 most of the times.

ALGORITHM 3: listFGS ← **Infer_Minimum_FlagGenerating_Siblings_Z** (brNode)

```
1  NotTakenbias ← outcomeProfile.get(brNode).notTaken >
   outcomeProfile.get(brNode).taken;
2  if NotTakenbias ≥ 1 then
       // Use NonZero Profile if ZF is 0 most of the times
3      nzprof ← get NonZeroProfile for brNode;
       // Which one is sufficient most of the times?
4      minimum_required_chunk ← chunk ID of max(nzprof.Chunk0, nzprof.Chunk1,
       nzprof.Chunk2, nzprof.Chunk3);
5      listFGS.add(minimum_required_chunk);
6      Update ExecCntSufficient to frequency of minimum_required_chunk;
7  else
       // Use Zero Profile if ZF is 1 most of the times
8      zeroProf ← ZeroProfile for brNode;
       // Do at least the max of src1 encoding and src2 encoding
9      minimum_required_chunk ← max(zeroProf.max_ra_encoding,
       zeroProf.max_src2_encoding);
10     for chunk from Chunk0 to Chunk_minimum_required_chunk do
11         listFGS.add(chunk);
12     end
13     Update ExecCntSufficient to frequency of minimum_required_chunk;
14 end
15 return listFGS;
```

As the flag-generating siblings generated a value of Zero most of the times, the profiler infers the most-frequent size-sign encodings of the input values of operands src1 and src2 (line 7 Algorithm 3). The size of the *listFGS* may vary from 1 to a maximum of 4. Also note that the algorithm updates the *ExecCntSufficient*, which denotes the execution count of the number of times the listFGS is sufficient to correctly compute the flag. *ExecCntSufficient* is inferred from the profiles and is later used to perform a cost-benefit analysis.

The minimum required flag-generating siblings, hence inferred (*listFGS* in Algorithm 3) form the basis of the *MBC Flag-generating Backslice*, which is formally defined as follows :

Definition 6.7 (**MBC Flag-generating Backslice**)**.** The *MBC Flag-generating Backslice* for a conditional branch consists of the minimum required flag-generating siblings and their backslices within the conditional control-flow of the optimization region. The inference of the minimum required flag-generating siblings is profile-based. Hereafter, it is also referred to as '*mbc-controlflow*'.

6.3.3.3 Step C : Infer Default Flag-generating Backslice

The next step is to infer the *Default Flag-generating Backslice*. In the event that the MBC flag-generating backslice is not sufficient, the control-flow needs to default to a non-speculative static narrow ISA code stream which is *always sufficient* for computing the correct value of flag. This piece of code is referred to as the Default flag-generating backslice and is defined as follows :

Definition 6.8 (**Default Flag-generating Backslice**)**.** The *Default Flag-generating Backslice* for a conditional branch consists of those computations in the conditional control-flow, which are always sufficient to generate the correct value of the flag for the respective conditional branch. Hereafter, it is also referred to as '*default-backslice*'.

Table 6.5: Illustrating reorderability for MBC using the most-common cases

Id	Wide Opcode (64-bit)	Required Condition Code	Category	Comments
1.	sub, add, and, or	E	Reorderable	Reorderable because the final value of the zero flag is the logical *or* of the zero flag of the individual chunks
2.	sub, add	C	Non-reorderable	Non-reorderable because CF flows from the least significant chunk to the most significant one
3.	sub, add	LE / BE	Reorderable	If the condition being evaluated is LE / BE, an implicit order is already in place
4.	shr, shl	E	Non-reorderable	Occurs infrequently. Non-reorderable because a chunk may have non-zero bits shifted into it from its neighboring chunk

The notion of reorderability of flag-generating siblings is exploited to infer the Default flag-generating backslices. Table 6.5 showcases the most-common cases to further illustrate the concept of reorderability. If the required condition code is E/NE and is generated by any opcodes like sub, add, logical and / or, the default flag-generating backslice simply needs to operate on the left over chunks. This is because the final value of the zero flag is the logical *or* of the zero flag of the individual chunks. Similarly, if the required condition code is LE/BE/NLE/NBE, the default flag-generating backslice can simply take over from the chunk

left over by the MBC backslice. For example, if both $Chunk_3$ and $Chunk_2$ are indecisive to generate the correct value of the condition code LE, it is sufficient to check the rest of the chunks $Chunk_1$ and $Chunk_0$.

Non-reorderability in the current context implies that the state of the flags generated by the MBC flag-generating backslices cannot be considered as a correct partial state, and all the computations must be redone to generate the correct value of the flag, should the MBC flag-generating backslice not suffice. For example, as a specific example of case 2 in Table 6.5, if $Chunk_3$ on subtraction does not generate a carry, all four chunks must be subtracted in order (from least significant chunk to the most significant chunk) to generate the correct value of the CF.

Hence, reorderability allows the flag-generating siblings to reuse each other's generated flag values irrespective of their order of execution. As shown in Algorithm 4, when the flag-generating siblings are *reorderable* (Definition 6.4), the default flag-generating backslice is computed as the set difference of Set(*lppopt-controlflow*) and Set(*mbc-controlflow*) (line 3 of Algorithm 4). A set difference of two sets B and A is denoted as B / A where B / A = { x∈ B | x ∉ A } (same as [B - (B ∩ A)]). On the other hand, if the flag-generating siblings are non-reorderable, all the computations in the conditional control-flow (*lppopt-controlflow* in this case as the control-flow is already reduced by LPP) of the branch need to be done again.

ALGORITHM 4: $default\text{-}backslice$ ← **Infer_Default_FG_Backslice** (brNode)

```
1 reorderable ← is_reorderable(brNode, fgSiblings);
2 if reorderable then
      // Set difference denoted by B / A
3     default-backslice ← Set(lppopt-controlflow) / Set(mbc-controlflow);
4 else
      // Use Zero Profile if ZF is 1 most of the times
5     default-backslice ← Set(lppopt-controlflow);
6 end
7 return default-backslice;
```

6.3.3.4 Step D : Perform Cost-Benefit Analysis

This section outlines a simple cost-benefit analysis used to evaluate the efficacy of MBC-based narrow backslice reordering. Costs and benefits are estimated in terms of the dynamic number of narrow ISA computations using collected information from the profiles because the goal of the MBC optimization is to reduce the dynamic code footprint of the narrow ISA. Specific execution frequencies of the different optimization scenarios are available via the collected profiles.

Cost of conditional Branch before MBC optimization can be calculated as :

$$Cost_{before} = \left[\sum_{0 \leq i \leq msc} |fgSlice_i| + |Branch| \right] * ExecCntBranch \tag{6.1}$$

where, $fgSlice_i$ denotes the i^{th} flag-generating sibling and its backslice. The number of fgSlices is correlated with the value of msc (most significant chunk). The value of msc may range from 0 to 3 because it depends on the operation size of the original flag-generating 64-bit computation. For example, if the x86 operation size of the wide flag-generating operation is 32-bit, the value of msc will remain 1 (signifying the $Chunk_1$). | X | denotes the cardinality of the set X. For example, the cardinality of the set containing the conditional branch is 1. $ExecCntBranch$ is inferred from the Outcome Profile of the static conditional branch and denotes the dynamic execution count of the conditional branch.

Next, the cost of conditional branch after MBC optimization can be calculated as:

$$\begin{aligned} Cost_{after} = & \left[\sum_{i} |MBCfgSlice_i| + |MBCBranch| \right] * ExecCntBranch \\ & + \left[|DefaultfgSlice| + |DefaultBranch| \right] \\ & * (ExecCntBranch - ExecCntSufficient) \end{aligned} \tag{6.2}$$

where, $MBCfgSlice_i$ denotes the i^{th} MBC flag-generating sibling and its backslice. Note that, more than one MBC flag-generating sibling may be required to compute the flags minimally. $ExecCntSufficient$ is inferred from the profiles and denotes the dynamic execution count of the number of times the set of MBC flag-generating backslices were sufficient to compute the flags correctly. | $DefaultfgSlice$ | denotes the number of narrow computations in the Default flag-generating backslice, which is required to be executed whenever the MBC flag-generating siblings are not sufficient.

Hence, the reordering of backslices via MBC is beneficial if the cost of the branch after optimization is less than the cost of the branch before such an optimization. Clearly, the cost-benefit analysis does not necessarily guarantee that the reordering will be beneficial because the actual behavior of a conditional branch may differ from what witnessed in the profile phase.

6.3.3.5 Step E : Perform fgSlice Reordering

Finally, if the cost-benefit analysis indicates that the reordering may be beneficial, fgSlice reordering is performed. A conceptual layout of the reordering has been provided in Figure 6.5. Finally, the pseudo code provided in Algorithm 5 summarises the work flow of the MBC optimization.

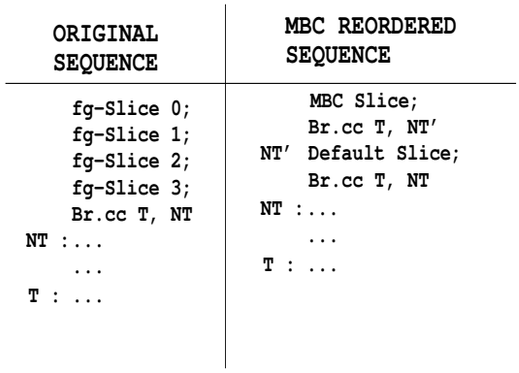

Figure 6.5: MBC reordered code

ALGORITHM 5: **MBC_Optimize_Region**

```
1  brNode ← Infer Conditional Branch from Region;
2  if brNode then
3      lppopt-controlflow ← Get_LPPOptimized_ControlFlow_Backslices(brNode);
       // Infer Min Required flag-generating sibling(s) from Profiles
4      switch conditionCode do
5          case JZ / JNZ
6              mbc-siblings ← Infer_Minimum_FlagGenerating_Siblings_ZNZ(brNode);
7              mbc-controlflow ← Get_Backslices(mbc_siblings);
8              break;
9          end
10         case JLE / JNLE
11             mbc-siblings ← Infer_Minimum_FlagGenerating_Siblings_LENLE(brNode);
12             mbc-controlflow ← Get_Backslices(mbc_siblings);
13             break;
14         end
15         case JL / JNL
16             mbc-siblings ← Infer_Minimum_FlagGenerating_Siblings_LNL(brNode);
17             mbc-controlflow ← Get_Backslices(mbc_siblings);
18             break;
19         end
20     end
21     if sizeof(lppopt-controlflow) > sizeof(mbc-controlflow) then
22         default-backslice ← Infer_Default_FG_Backslice(brNode);
23         costEffective ← Perform_Cost_Benefit_Analysis(brNode, mbc-controlflow,
           default-backslice);
24         if costEffective then
25             Reorder_fgSlices(brNode, mbc-controlflow);
26         end
27     end
28 end
```

Issues with Reordering Backslices. In the context of MBC, there is a fundamental limitation in reordering narrow backslices around conditional branches : a live-in variable of the Default fgSlice cannot be a live-out of the MBC fgSlice. This is because a live-in of the Default fgSlice is meant to consume the same value of the variable that is present before the MBC fgSlice executes. The current implementation of MBC handles them by renaming such references with no additional overheads (assuming free register names are available at all times). The dynamic weight of these cases remains extremely low (as the conditional control-flow slices are small in general).

6.3.4 Cost Analysis

Optimizing a region by MBC involves applying the sequential steps as shown in Figure 6.4 previously. The cost of optimizing code regions by MBC is the sum total of the cost of applying each of these individual steps. The cost of categorising data-flow computations from conditional control-flow is proportional to the static number of computations in the region to be optimized. Next, the cost of Step B and Step C each is proportional to the static number of computations comprising the conditional control-flow of the programs. This is because the inference of both the MBC flag-generating backslice and the default flag-generating backslice, in the worst case, is bounded by the static number of computations in the conditional control-flow of the region to be optimized.

6.4 Example : Walk-through

Now we illustrate the MBC optimization by using an example (refer to Figure 6.6). The example is based on a small piece of code from a hot basic block of the benchmark vpr (function try_swap).

For sake of clarity, Figure 6.6(A) shows only the non data-flow computations of the basic block. Each node shows the respective opcode together with the post-order numeric underneath the opcode. The add(c), ld, and sub(c) operations are the conditional control-flow computations. The graph also shows the four mov to the *inrip* register operations[1] – nodes 2, 3, 4, and 5, which update the *inrip* register with the taken address of the conditional branch.

Before MBC is applied, the optimizer prunes the code with LPP. First, profile-based LPP non-productive computations are inferred. Next the optimizer prunes them and places the required asserts appropriately. Nodes 2, 3, 4, and 5 are Group0 computations and do not require any assertions to prune them. Nodes 14, 15, 16, and 17 are Group2 computations (notice that

[1]By definition, these four mov operations are not conditional control-flow computations

although destination is *inaddr*.x, the source register is *r13*.x). Pruning node 16 requires a value based assertion, whereas the assertion requirements of node 14, 15 can be combined into a single assertion using size-sign encoding. Hence, two asserts are embedded in lieu of nodes 14, 15 and 16.

Nodes 6, 7, 8, and 9 are the flag-generating siblings. Also shown in Figure 6.6(B) are the flag-generating backslices of each of these siblings. The next step is to infer which of the flag-generating siblings are minimally required *most of the times* (Step 7 of Algorithm 5). As the conditional branch was not-taken most of the times, the associated Nonzero-Profile of the conditional branch (node 1) is consulted (Step 3 of Algorithm 3). In this example, profiles indicated that the least significant flag-generating sibling (node 9) is sufficient 92% of the total number of profiled executions.

Next, the optimizer needs to infer the Default backslice for MBC. Notice, the carry flag dependence (cf) between the flag-generating siblings (node 6, 7, 8, and 9). The presence of such dependences makes these flag-generating siblings non-reorderable. However, a simple data-flow analysis reveals that the carry flag generated by node 9 is never consumed (dead flag-writer). Hence, removing the carry-flag dependences between nodes 6, 7, 8 and 9 is safe and hence, they can be made reorderable.

Hence, the default backslice is calculated as the set difference of the LPP-optimized conditional control-flow (nodes 6 through 13, node 17 and the two asserts) and the flag-generating backslice of the profile-based minimum required sibling : $zero^{th}$ sibling (nodes 9 and 13). Hence the default backslice consists of nodes 6, 7, 8, 10, 11, 12 (shown in Figure 6.6(C)).

Before finally carrying out the reordering, the optimizer computes whether the reordering is expected to be beneficial, based on the profiles. Hence, it computes cost of the conditional branch (before reordering) according to Equation 6.1 by substituting $|fgSlice_i|$ by 2, |Branch| by 1, and ExecCntBranch by 100 :

$$Cost_{before} = \left[\sum_{0 \leq i \leq 3} 2 + 1\right] * 100 = 900 \tag{6.3}$$

Similarly, the cost of the conditional branch (after reordering) is calculated using Equation 6.2 by substituting $|MBCfgSlice\_i|$ by 2, m by 0, |MBCBranch| by 1, and ExecCntSufficient by 92 :

$$\begin{aligned} Cost_{after} &= \left[\sum_{i=0} 2 + 1\right] * 100 + \left[6 + 1\right] * (100 - 92) \\ &= 356 \end{aligned} \tag{6.4}$$

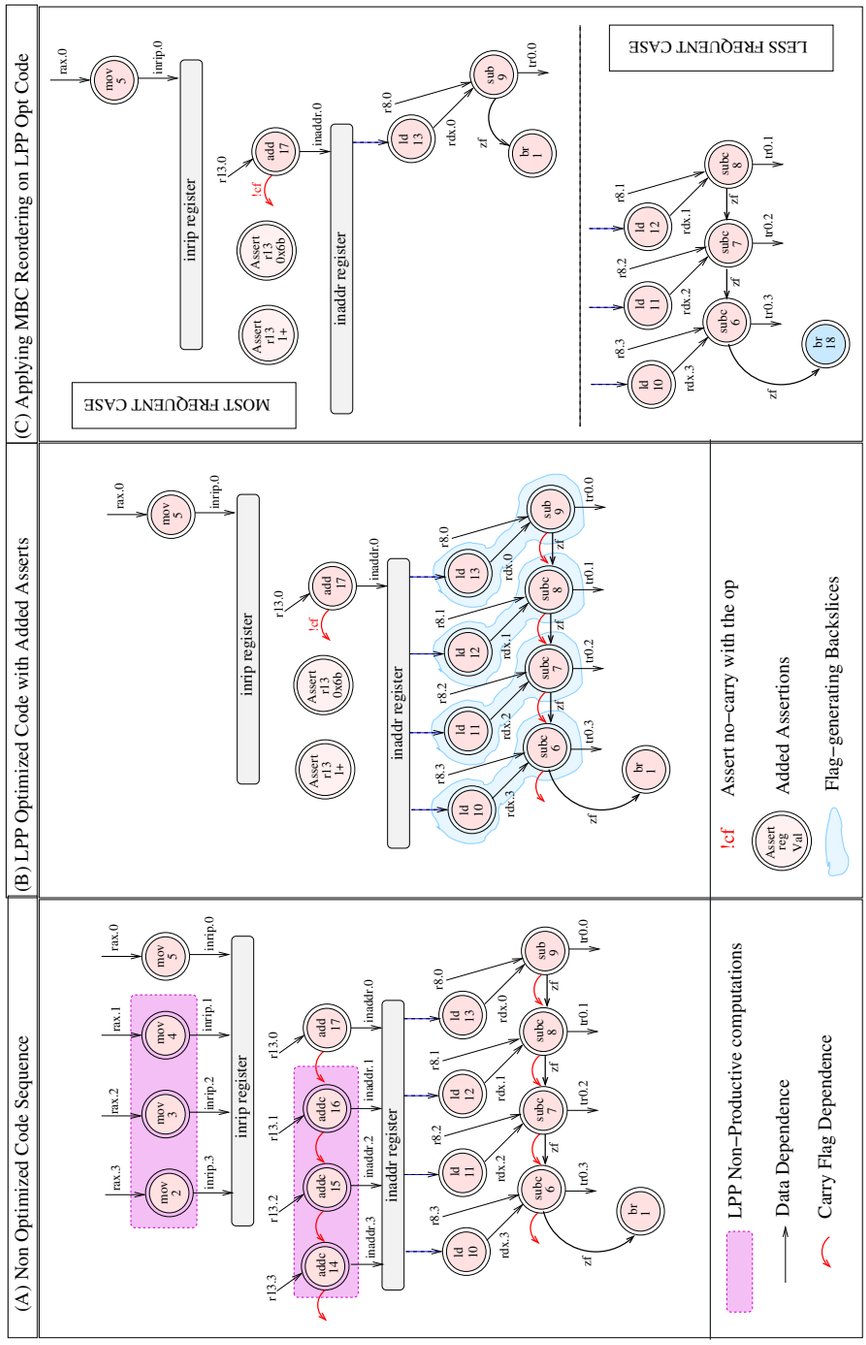

Figure 6.6: MBC at work

Hence, as profiles suggest that the reordering can be potentially beneficial, MBC reordering is carried out (Step 26 of Algorithm 5). The final reordered code sequence is shown in Figure 6.6(C).

6.5 Related Work

Conditional branches have been viewed as expensive operations. This is because it may take multiple computations to determine the outcome and may cause pipeline flushes when mis-predicted. The cost of a mis-prediction increases with deeper pipelines and the degree of superscalar execution. Reducing the number of conditional branches can be done in various ways (code reordering, speculation, loop transformations etc.) and there exists extensive research in this regard. Recall that the MBC optimization does not 'optimize the control-flow' as done by some previous research [31, 60]; it only *reduces the control-flow backslices* of conditional branches. Hence, in this section, we only compare two of the most relevant previous works in the context of the MBC optimization.

One of the previous works [61] proposes to reduce the cost of conditional branches by reordering them, such that lesser number of conditional branches are executed and hence, increasing performance. The proposed technique is also profile-based and may also result in insertion of additional branches. The results of applying this transformation are an average reduction of 8% fewer instructions executed, reduction of 13% in branches executed, and lastly, about 4% decrease in execution time. The reorderable sequences typically consist of branches comparing the same variable or expression to constants.

Another work proposes to perform conditional branch elimination by condition merging [31]. This is also a profile-based technique which replaces multiple conditional branches with a single branch on a conventional scalar processor. For instance, the test if (p1 != 0 && p2 != 0), which is testing for NULL pointers, can be replaced with if (p1 & p2 != 0). Overall, 15.8% of branches were eliminated, leading to an average reduction of 5.74% in the number of instructions executed.

It can be clearly seen that the MBC optimization in a hardware/software ecosystem around the narrow bitwidth architecture is different from the previous approaches. In fact, MBC can be combined with the foregoing techniques to fetch gains of multiple code pruning techniques.

6.6 Evaluation

In this section, the performance evaluation of the MBC optimization is presented. First, we briefly revisit the experimental framework (already described in Chapter 3) in the context of

MBC in order to aid the understanding of the upcoming evaluations.

The optimization region for consideration is a basic block. The performance of MBC has been evaluated using two different configurations –

(i) *MBC-On-NonOpt*, where the MBC is applied on the non-optimized stream of narrow computations, and

(ii) *MBC-On-LPPopt*, where the compiler first prunes the narrow code stream using Local Productiveness Pruning (LPP) and then applies the MBC optimization.

6.6.1 Experimental Framework

The MBC optimization has been evaluated in a *dynamic optimizer model*. However, it must be noted that the evaluation of MBC *does not account for the overheads of profiling and optimization* of the narrow code stream. The overheads of profiling for MBC are expected to be much lower than previously proposed optimizations in this thesis. This is because as compared to the code pruning techniques, only a small number of dynamic instances are profiled (only dynamic instances of flag-generating siblings for conditional branches).

Table 3.1 (in Chapter 3) shows the simulation configurations of the narrow processor used for evaluating MBC on a narrow processor. We model an in-order processor of an issue width of four instructions per cycle and compare the performance of a 'MBC optimized code on a narrow processor' against that of a 'Narrow computations stream without the MBC optimization on a narrow processor'.

The execution model first profiles for profile-phase of 200m user instructions (after skipping the program initialization phase) using the *ref* input data-set and then the MBC optimization is triggered. For the *MBC-On-LPPopt* configuration, the optimizer prunes the narrow stream of computations via the LPP optimization and then triggers the MBC optimization. For the *MBC-On-Nonopt* configuration, the optimizer prunes the non-optimized narrow stream of computations via the MBC optimization only. Table 3.3 shows the percentage of the committed stream represented by the optimized regions in the profile-phase of the applications; this remains the same as observed for the LPP optimization as both the optimizations are applied on *hot regions only*.

The optimized regions are then used in the cycle-accurate model for the cycle-accurate phase (the next 200m user instructions) using the same input data-set as the foregoing profile-phase for each application. At the beginning of the execution of an optimized region, the system state is checkpointed (recall that the regions are speculative as LPP is applied on them). In case of assertion failure event in the optimized region, hardware support restores correct

program state by using the checkpointed system state. The execution then resumes with re-translated, safe, correct code (non-optimized stream) of the region (i.e., function in this evaluation). For the *MBC-On-NonOpt* configuration, however, no checkpointing / rollback recovery mechanisms are required because MBC as a standalone optimization generates completely self-sufficient code and both there is no requirement for initiating a rollback recovery mechanism.

Handling the Branch Predictor. A primary side-effect of the MBC optimization is that the number of both static and dynamic conditional branches increases : An MBC optimized code region may execute twice the number of conditional branches in the worst case[1]. Although the impact on the dynamic code footprint of the application due to additional conditional branches may be negligible, the additional branches may have an impact on behavior of the Branch Predictor. In some cases, the MBC optimized narrow stream of computations experiences a small dip in the accuracy of the branch predictor. The effects, if any, on the performance are duly accounted for in the upcoming evaluations.

6.6.2 Quantifying Conditional Control-flow

Figure 6.7 quantifies the percentage of narrow computations which constitute the conditional control-flow backslices (Definition 6.3). It shows the split of the dynamic narrow computations in the non-optimized hot regions of the applications between –

(i) *Branches* : all the conditional and unconditional branch operations illustrated by *Dyn-Cond-Br* and *Dyn-Uncond-Br* respectively,

(ii) *Conditional control-flow backslices* : all flag-generating siblings and their backslices which contribute to the generation of flag value only, illustrated by the *Cond-Control-flow*, and

(iii) *Rest of the computations* : rest of the program consisting of the data-flow, illustrated by the *Rest*.

For this experiment and further evaluations of MBC, the data-flow (Definition 6.2) and the conditional control-flow backslices of a region are inferred on a basic block as the optimization region. As shown in Figure 6.7, on an average, about 20% of the narrow computations committed from the hot regions (in the profile-phase) are flag-generating siblings and their backslices. These are those computations which are responsible for the generation of flag values only (directly or indirectly). Further, these are the only computations that MBC can potentially reorder

[1] If the MBC-Opt branch fails, the Default fgSlices and finally the Default branch will be executed

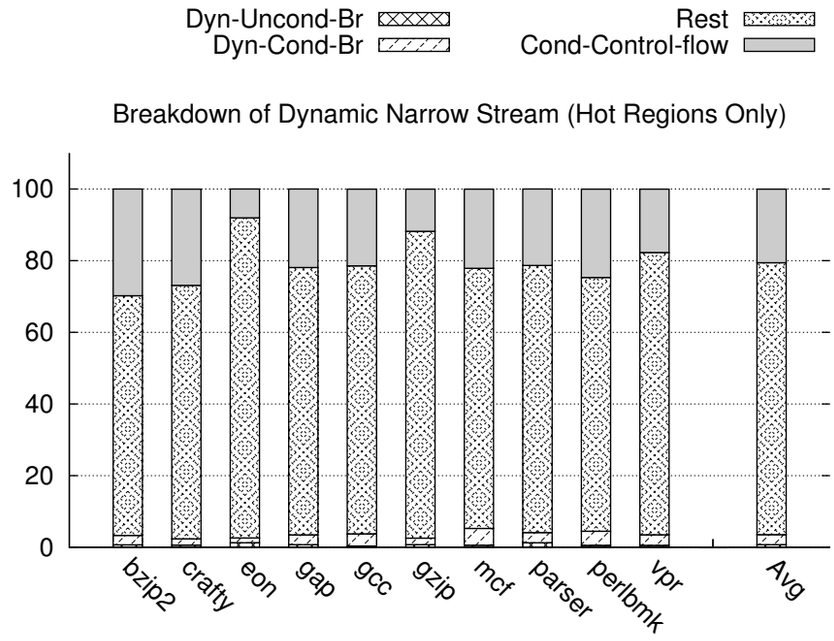

Figure 6.7: Conditional control-flow vs. rest of the program

to reduce their dynamic code footprint. It must be noted that the distribution as shown in Figure 6.7 is of the narrow computations in the non-optimized hot regions only. LPP affects the overall scope and benefits of MBC (as suggested by the upcoming evaluation results).

On an average, the conditional control per branch is 7.31 narrow computations in non-optimized workloads (an average of 20.56% conditional control-flow computations across an average of 2.81% of dynamic conditional branches as seen in Figure 6.7).

Marginal increase in the control-flow slices is observed if superblocks are used as the optimization region. An increase is explicable because a live-out of a basic block may be so because it is consumed by the conditional control-flow computation of the subsequent basic block. When a superblock is created however, such computations on the most-frequent paths can be detected as conditional control-flow of the respective branch in a superblock. For this reason, we choose to evaluate the MBC optimization on basic block as the optimization region.

Further, another dimension that impacts the overall scope of the MBC optimization is the number of flag-generating siblings per conditional branch. Recall that if the operation size of the wide flag-generating instruction specified at the x86 ISA level is 16-bit, only one flag-generating sibling is generated by the narrow translation scheme; an operation size of 32-bit

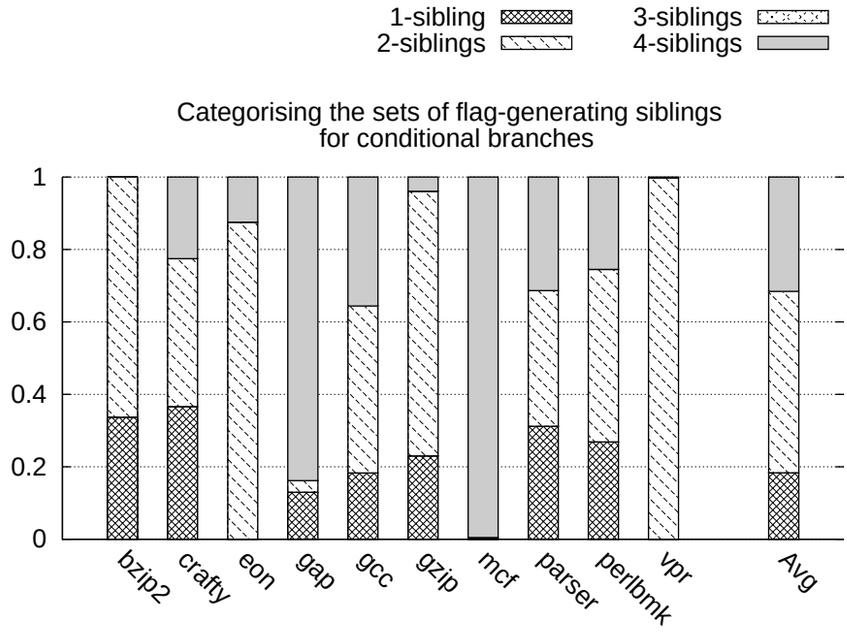

Figure 6.8: Breakdown of dynamic conditional branches in terms of the size of the set of flag-generating siblings

implies a set of two flag-generating siblings[1]. A set of three flag-generating siblings is not seen because x86 does not allow an operation size of 48-bit. Lastly, an operation size of 64-bit implies that the narrow translator generates four flag-generating siblings.

Figure 6.8 shows the distribution of these dynamic sets. Around 50% of the conditional branches have only two flag-generating siblings; MBC can potentially reduce the cost of the conditional branch (by reordering them) only by around a half. Hence, the distribution shown in Figure 6.8 suggests that bzip2, eon and vpr are expected to offer less gains by the MBC optimization. Around 18% of the conditional branches have only one flag-generating sibling and hence, out of scope for any benefits by MBC. Around 31% of the conditional branches have four flag-generating siblings, providing MBC more opportunity to reorder the siblings and their backslices. It must be underlined here, that the presence of four flag-generating siblings is not a necessary condition for high gains by MBC, as we will see later for the benchmark mcf.

[1]There may be mov operations to push zero into the higher 32-bits of the destination, but these do not offer much potential for MBC because such mov operations do not have any backslices.

6.6.3 MBC Evaluation

As stated previously, this section compares the performance of a 'narrow processor with the MBC optimized code' against that of a 'narrow processor[1] without such an optimization'. The performance metrics for MBC include – the default rate of MBC optimized conditional branches, the number of cycles taken for execution, and the number of committed computations. MBC has been applied on individual basic blocks as optimization regions.

MBC is evaluated in a dynamic optimizer model. The cycle-accurate phase of each application lasts for 200m x86 user instructions. It must be noted, however, that the upcoming evaluations are reported only for the hot regions in isolation.

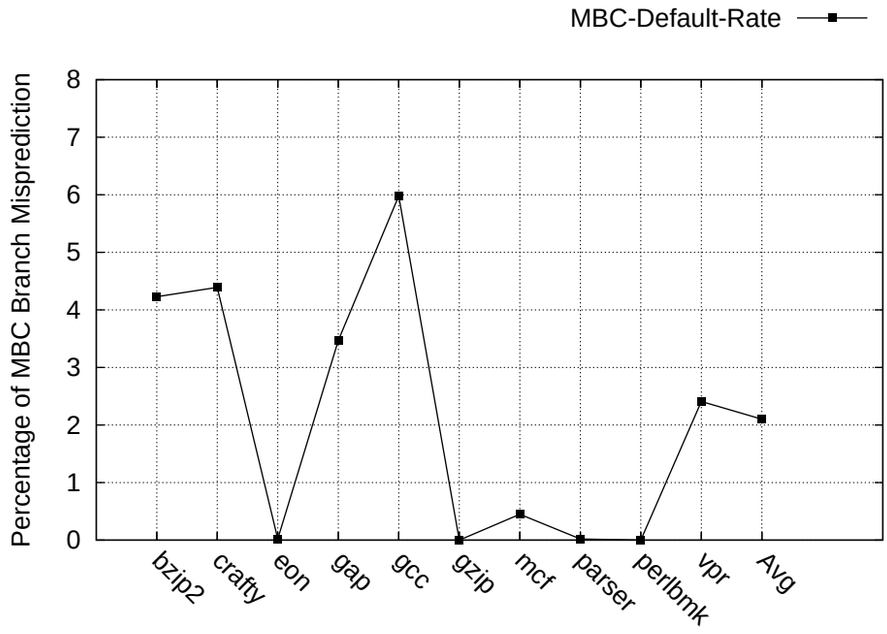

Figure 6.9: Percentage of times MBC's reordering strategy fails

Default Rate of MBC. The Default Rate of MBC is defined as the rate at which the MBC flag-generating backslice fail to be sufficient. Mathematically, it is evaluated as :

$$DefaultRate = \left[\frac{Number\ of\ Times\ MBC\ Default\ Slice\ Required}{Total\ number\ of\ MBC\ optimized\ Conditional\ Branches}\right] * 100 \tag{6.5}$$

[1] two different configurations are evaluated – with and without LPP

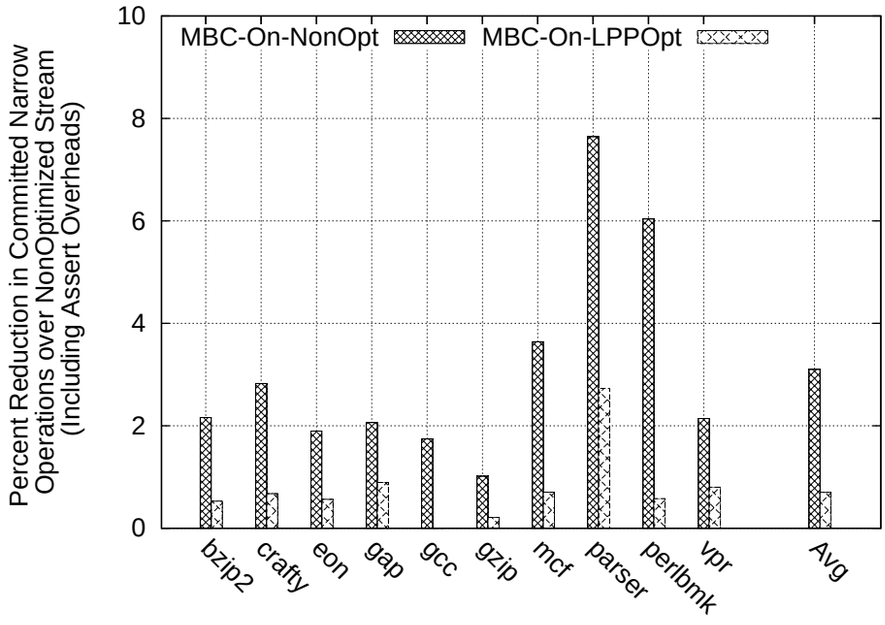

Figure 6.10: Hot Regions in isolation – Reduction in narrow computations achieved by MBC

Figure 6.9 shows the default rate of each application. The average default rate is 2.1%. gcc has a high rate of defaulting to the Default flag-generating backslices for flag computation (around 6%).

Reduction in Narrow Computations. Figure 6.10 shows the percentage reduction in the dynamically committed narrow computations in two different configurations. *MBC-On-NonOpt* shows the percentage reduction in the committed stream by applying MBC on the non-optimized narrow computation stream. *MBC-On-LPPOpt* shows the percentage reduction in the committed stream by applying MBC on an already LPP optimized narrow computation stream.

The benchmarks eon and gzip have low gains because of a rather small set of conditional control-flow computations (average 10% as shown in Figure 6.7). The three benchmarks bzip2, crafty and perlbmk exhibit a relatively higher percentage of conditional control-flow (an average of more than 25%). However, of these, bzip2 and crafty suffer low gains by MBC due to relatively higher default rate of around 4%. Similar reason holds for gap and gcc.

On an average, about 3.12% of the dynamic stream is reduced when MBC is applied on non-optimized regions. When MBC is applied on an LPP optimized stream, only 0.74% of the dynamic stream is further reduced[1]. This indicates that most of the conditional control-flow

[1] if LPP reduces x% on an average, LPP+MBC reduces (x+0.74)% computations

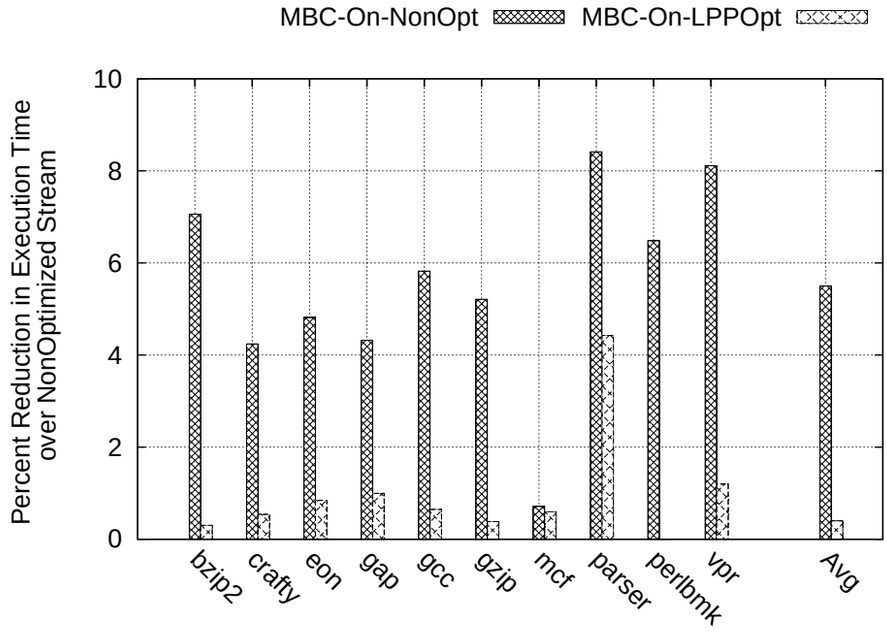

Figure 6.11: Hot Regions in isolation – Reduction in total number of cycles

slices are already sufficiently pruned by LPP.

Reduction in Cycles. Figure 6.11 shows the percentage reduction in the total number of cycles spent by each program in the hot regions only. Both the configurations – *MBC-On-NonOpt* and *MBC-On-LPPOpt* have been shown.

On an average, compared to the non-optimized narrow computation stream, MBC achieves a 5.52% reduction in the total number of cycles. When MBC is applied on an LPP optimized stream, only a further reduction of 0.42% in the total number of cycles[1] is seen. This, once again, is due to the fact that most of the conditional control-flow slices are already sufficiently pruned by LPP.

Dynamic Stream Classification. Figure 6.12 shows the histogram distribution of the committed stream of the LPP + MBC Optimized code stream. The code coverage of MBC is very similar to that observed by LPP, when the latter was evaluated in a dynamic optimizer model. The only anomaly is in the case of gcc, where the use MBC causes a drop in the achieved code coverage. This is due to the high default rate that has been observed in gcc.

[1] if LPP reduces x% on an average, LPP+MBC reduces (x+0.42)%

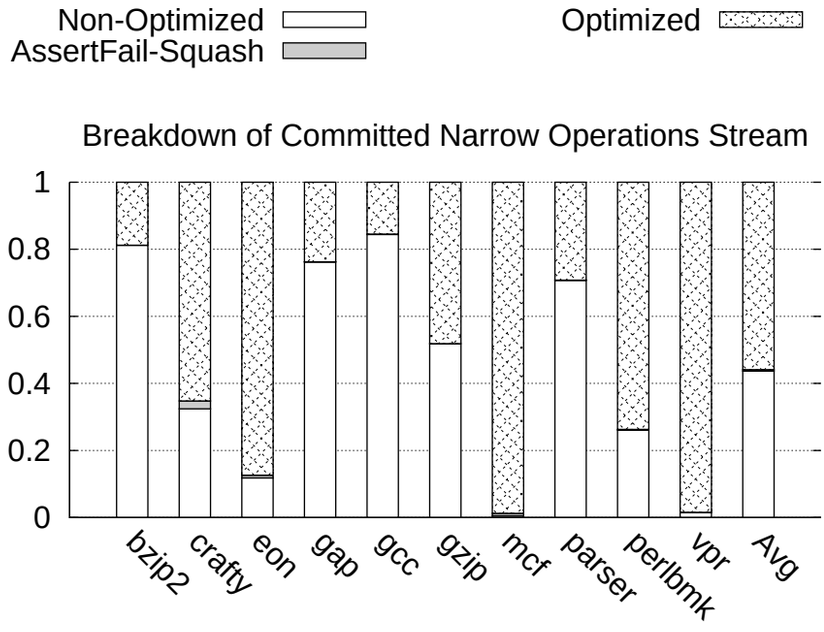

Figure 6.12: Dynamic stream classification of the MBC+LPP optimized narrow computation stream

On an average, 56% of the complete program is executed from the optimized regions (LPP+MBC). The amount of the committed stream which shows the narrow operations lost due to assertion failures, including the assertions themselves (in *assert-fail*) remains similar to what has been observed for LPP (Figure 5.7). This is expected as the number of assertion failures are not impacted much by MBC, although MBC does introduce a small number of additional assertions.

Impact on the Static Code Size (After Optimizations). It has already been shown in Figure 2.3 (Chapter 2) that on an average, a non-optimized narrow ISA program needs about 3.9 times more static instructions than its equivalent 64-bit program. With respect to the optimized code, however, recall that when applying the code pruning techniques (GPP and LPP), the compiler needs to generate not only the optimized but also the non-optimized versions of the narrow ISA programs. The MBC optimization differs from both GPP and LPP in that it does not require the non-optimized code. The generated (reordered) code by MBC is self-sufficient and the execution model does not require a rollback to safe, non-optimized code.

Figure 6.13 shows the average static code size explosion due to the two optimization techniques when applied incrementally – LPP and MBC. Note that on an average, the LPP op-

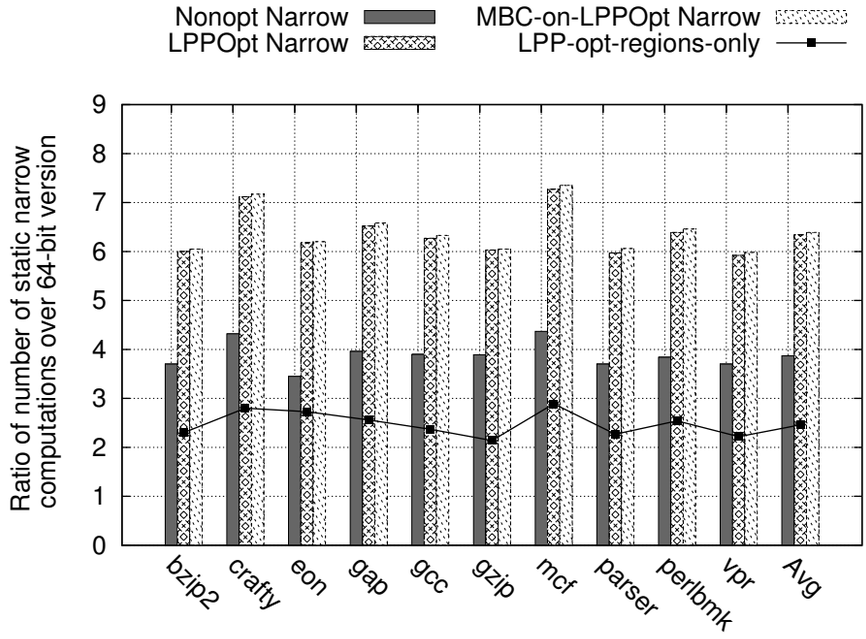

Figure 6.13: Impact on the static code size - Before and after optimizations

timized narrow ISA programs have about 6.3x more static instructions than their equivalent 64-bit versions (refer to *LPPOpt Narrow* bar plot in Figure 6.13). Further, applying MBC and LPP together, the optimized narrow ISA programs are observed to have about 6.4x more static instructions than their equivalent 64-bit versions (refer to *MBC-on-LPPOpt Narrow* bar plot in Figure 6.13).

It must be underlined here that LPP reduces the static code size of the optimized regions from 3.9x to 2.4x (line plotted as *LPP-opt-regions-only* in Figure 6.13). Secondly, in case of LPP, the execution rarely goes to the non-optimized code regions (recall that the assertion failure rates of LPP are very low). The latter can be useful in code layout optimizations of the speculatively optimized narrow ISA programs.

6.6.4 Observed Roadblocks

Figure 6.14 shows the dynamic stop conditions encountered by MBC, weighted by the associated control-flow of each conditional branch. The histogram distribution shows a split amongst the following categories :

(i) *Optimized* shows the percentage of the conditional control-flow which is successfully optimized.

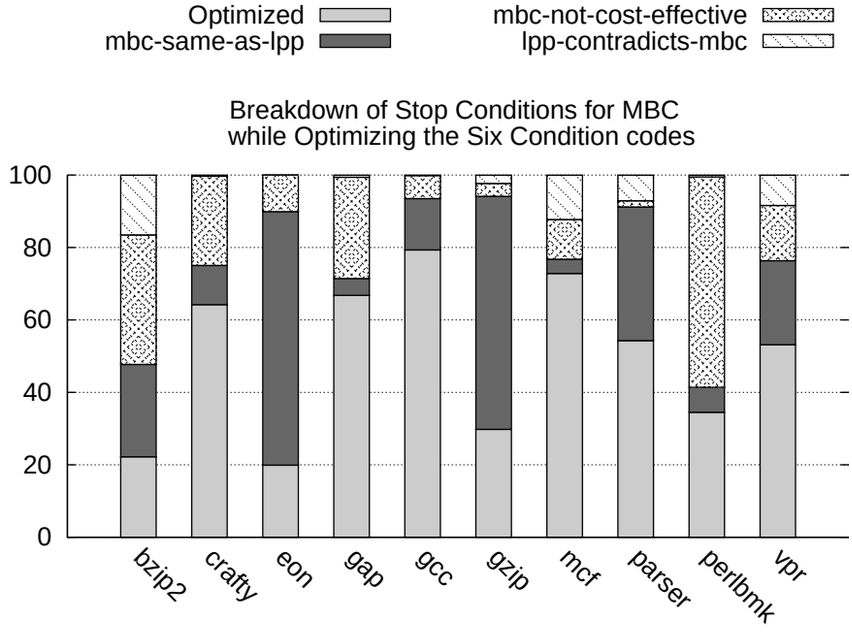

Figure 6.14: Dynamic stop conditions

(ii) *mbc-same-as-lpp* shows the percentage of conditional control-flow where MBC has no further scope of reducing the flag-generating backslices by reordering. The redundant computations targeted by MBC is already pruned by LPP.

(iii) *mbc-not-cost-effective* shows the percentage of conditional control-flow where applying MBC may potentially be unfavorable and hence, MBC is not performed.

(iv) *lpp-contradicts-mbc* shows the percentage of conditional control-flow where the MBC flag-generating backslice is already pruned by LPP. Hence, what remains behind after code pruning by LPP is a sub-optimal way of generating the flags.

Hence, we can conclude that MBC performs close to its expected potential. However the way MBC defines the conditional control-flow of an optimizable region is expected to impact the overall performance of MBC and is one of the areas of improvement. On an average, the conditional control per branch is 7.31 narrow computations in non-optimized workloads (an average of 20.56% conditional control-flow computations across an average of 2.81% of dynamic conditional branches as seen in Figure 6.7).

6.7 Conclusions

This chapter proposes and evaluates the Minimal Branch Computation optimization. This optimization presents itself as a use-case of the broader concept of *reordering narrow backslices* for reducing the dynamic code footprint of the narrow ISA. Minimal Branch Computation is a profile-based code reordering technique, which rearranges narrow computations around conditional branches based on certain opcode specific sufficiency rules.

Evaluating MBC as a dynamic code optimization reveals that it can reduce an average of 3.12% of the non-optimized narrow ISA code stream. With respect to the number of cycles, it achieves an average reduction of 5.52% over non-optimized narrow stream of computations. However, applying MBC on an already LPP optimized code stream fetches minimal reductions. This is because LPP is already effective in pruning even the conditional control-flow backslices of the optimizable conditional branches.

7

Conclusions and Future Work

7.1 Summary

Motivated by the propensity of narrow computations, this thesis evaluates a hardware-software collaborative approach to exploit them. The approach involves a redesign of the hardware to a narrow bitwidth architecture which is essentially a 16-bit datapath architecture combined with a 64-bit address interface. Such a redesign attempts to attack the problem of computational inefficiency which is inherent in our traditional computing systems with wider datapaths. The narrow bitwidth architecture strongly aims to keep the hardware very simple (narrow, in-order) and low power, while garnering the advantages of a 64-bit system (larger address space, support for current software and compilers, etc.). Software, in the form of a compiler accomplishes the key task of translating and optimizing the 64-bit applications on to the 16-bit hardware.

The Narrow Instruction Set Architecture (Narrow ISA) provides the software interface of the narrow bitwidth architecture to the outside world. In this thesis, we have developed and evaluated a realistic code translator which cracks the 64-bit RISC-like computations to the narrow ISA computations. Narrow ISA presents opportunities : there are more tasks of finer granularity. It also presents challenges : even using a realistic narrow translator induces a significant performance penalty. Compared to 64-bit programs, the dynamic code size of narrow ISA programs increases to 3.9x and the number of cycles taken increases to 2.2x.

This is a significant performance penalty and it is important to reduce it to make the narrow bitwidth architectures viable. This thesis entrusts the central responsibility of alleviating the negative performance penalty of the narrow ISA on the compiler. The thesis explores code optimization techniques woven around the perspective of *Minimum Required Computations*. Given a program, the notion of *minimum required computations* (MRC) aims to infer the minimum

set of computations which are required to generate the *same (correct) output* as the original program.

This thesis uses two main profile-guided heuristics to approximate the notion of MRC :

(i) *Non-productiveness based code pruning* to prune narrow computations, and

(ii) *Reordering narrow backslices* to reorder narrow computations.

Both the set of heuristics are profile-based and are designed with a strong focus to minimize the dynamic code footprint of the narrow ISA. The code pruning techniques are speculative and generate code that is self-sufficient to detect mis-speculations. This is achieved by embedding assertion computations at appropriate locations. As the optimized code contains speculation, hardware support is exploited to contain the speculation until it is safe to commit the atomic speculatively optimized region. On the other hand, although the technique of reordering narrow backslices also uses profile-based learning approach, the generated code can always calculate the correct state without the need for any hardware support for rollback or recovery.

Global Productiveness Pruning (Chapter 4) and Local Productiveness Pruning (Chapter 5) are the two code optimization techniques that investigate the design space of non-productiveness based pruning strategies to reduce the dynamic footprint of the narrow ISA. The main difference between the two techniques is the size of the optimization region. GPP aims to optimize large regions, and hence choses to work on the whole function as a region. LPP on the other hand investigates the behavior of the definition of productiveness on the smallest possible optimization region of a single narrow computation.

Global Productiveness Propagation (GPP). This optimization, given a code region, distinguishes between useful (productive) and useless (non-productive) 16-bit computations based on profile data. In order to prune the useless instructions, asserts are properly placed to redirect the execution to a safer version of the code when the assumed conditions do not hold at runtime. Overall gains by GPP are up to 6.6% reduction of the committed narrow computation stream and 4.5% reduction in the number of cycles for a 1-issue, in-order narrow processor, when an average of 60% of the code has been optimized via GPP.

Overall, the number of instructions reduced by GPP is not very high and there exists a significant gap in the expected gains in a perfect setup vs. the achieved gains. One of the key roadblock identified in this regard is the memory dependence modeling. Other schemes for memory dependence modeling may augment the efficacy of GPP and can be easily accommodated as GPP is already speculative in nature. Further, our evaluations suggest that future work, if any, must focus on investigating upon some conservative heuristics followed by GPP,

viz., marking all control-flow as productive, contradiction-handling (marking complete backslice as productive), no code duplication, and no code scheduling. Lastly, strategies to enhance coverage of the optimization will prove to be important too.

Local Productiveness Pruning (LPP). This is a speculative, profile-guided code optimization technique that prunes out individual non-productive computations based on their productiveness bias when viewed in isolation. Evaluations suggest that applying the definition of Productiveness on a finer granularity is more effective. LPP reduces the dynamic stream of narrow computations by $\sim$20%, and achieves around 15.54% reduction in cycles over non-optimized narrow stream. The average code-coverage achieved is $\sim$ 57% and the assertion-failure rates of this speculative technique also remain low ($\sim$ 2.5%).

Overall, the number of instructions reduced by LPP falls in close proximity to the measured disposable potential. This is after taking into account that LPP excludes memory operations completely and optimizes an average of 57% of the dynamically committed narrow computation stream. Given the fundamental difference of the size of the atomic region in GPP and LPP, and the low overall gains achieved by GPP, we do not combine the two code pruning strategies in the thesis.

Apart from reducing the narrow computation stream effectively, there is another dimension in which LPP acts *as an enabler* for further code optimizations. A brief study of a well-understood greedy scheduling algorithm (ETS scheduling) reveals that simply applying ETS code scheduling on the non-optimized narrow stream achieves an average speedup of about 19%. Further, compared to the non-optimized narrow stream of computations, LPP with *SR-CS* achieves a reduction of 15.54% in the number of cycles by itself. When LPP with *SR-CS* is combined with code scheduling, a cumulative reduction of about 26% in the number of cycles can be achieved.

Minimal Branch Computation (MBC). This is a profile-based code reordering technique which places those flag-generating backslice(s) *first* which are *most-probably sufficient* to generate the outcome of the condition code eventually consumed by the impending conditional branch computation. Optimizing a set of six condition codes (E/NE, L/NL, LE/NLE), MBC reduces an average of 3.12% of the non-optimized code stream, and reduces an average of 5.52% of the total cycles required by the optimized regions. Applying MBC on an already LPP optimized code stream fetches minimal reductions, as LPP is already effective in pruning even the conditional control-flow backslices of the optimizable conditional branches.

Hence, MBC explores the concept of *reordering narrow backslices* such that the minimal necessary computations are performed in the best case; the rest of the computations are performed only when necessary. Although MBC remains a specific use-case, we believe that this

approach of a lazy computation model can be extended to data-flow computations as well, whereby data chunks are generated on a need basis : a particular chunk is generated only if required.

To conclude, in this thesis we have proposed three code optimization techniques specific to the narrow ISA. Combining LPP together ETS code scheduling, a cumulative reduction of around 31% in the number of computations (from 3.9x to 2.68x) and an overall reduction of 37% in the number of cycles (from 2.2x to 1.38x) can be achieved.

7.2 Future Work

The proposed hardware/software collaborative approach towards the narrow computations opens up opportunities for further research in this paradigm, which is corroborated by the code optimization techniques proposed in this thesis. With each optimization, we have duly identified the key roadblocks limiting the efficacy of the optimizations in a realistic scenario.

We believe that further aggressive compiler optimizations can promote the narrow bitwidth architecture as an interesting design point for future low-power, low-cost execution cores. In the following sections, we have identified key areas to further focus on strategies to achieve competency with a wide (64-bit) in-order architecture in future.

7.2.1 Optimizing the Memory Interface

The proposed optimizations, especially Local Productiveness Pruning (Chapter 5), have been effective in reducing the dynamic code size of the narrow stream of computations. However, memory operations have been excluded as of yet. This is because applying a straightforward extension of LPP to memory operations could have led to non-trivial hardware changes, thereby defying the focus of the thesis.

Memory operations, however, remain an important class of operations in the context of this thesis. Figure 7.1 shows the histogram distribution of the different class of opcodes in two configurations – non-optimized narrow computation stream and LPP optimized stream of *hot regions only* in the cycle-accurate phase (second 200m x86 user instructions using the *ref* input data-set) of each application. As can be seen, LPP is effective for almost all type of operation classes. Also, to reduce the dynamic code footprint of the applications further, the next operation class that can be seen as the most prominent (hence, the most unoptimized) is the class of memory operations.

Not only that, attacking memory operations is also important for the following reasons :

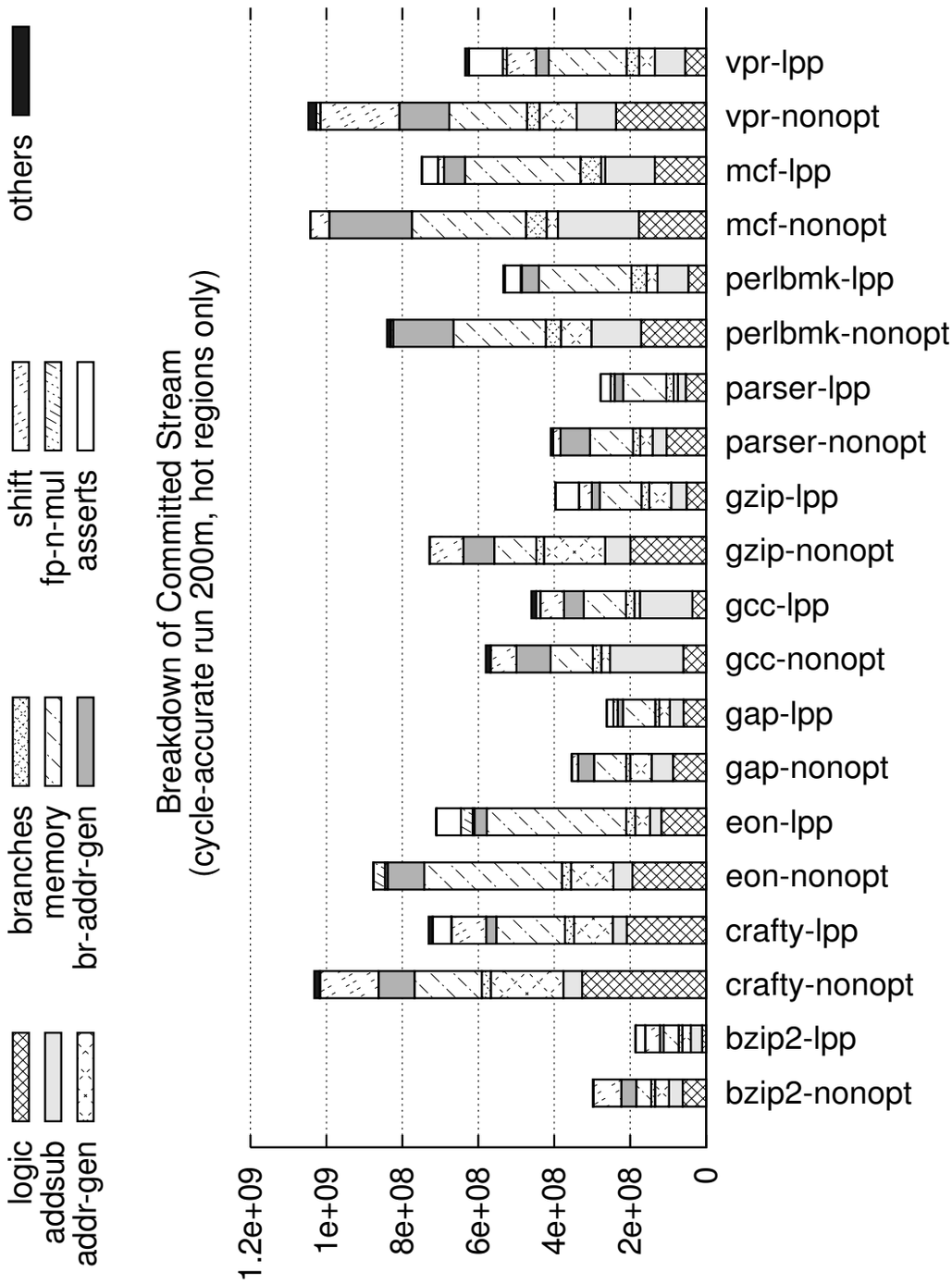

Figure 7.1: Breakdown of committed stream : Before and after LPP

1. Memory operations form the interface to the outside world in the context of a Von-neumann computation model. Hence, useless activity when detected at this interface can be used to further remove dependent operations (like addsub or logic class of operations), by explicit propagation of this inference.

2. Memory operations still remain to be one of the most costly operations (branches being the others). Memory pipelines are more expensive than execution units. They are critical not only for performance but also to reduce activity in the memory hierarchy – caching, buffering, and contention for memory bandwidth.

3. We have observed that pruning optimizations on ALU operations can be applied easily (Chapter 5), leaving memory operations as one of the least efficiently translated operations (from 64-bit to 16-bit).

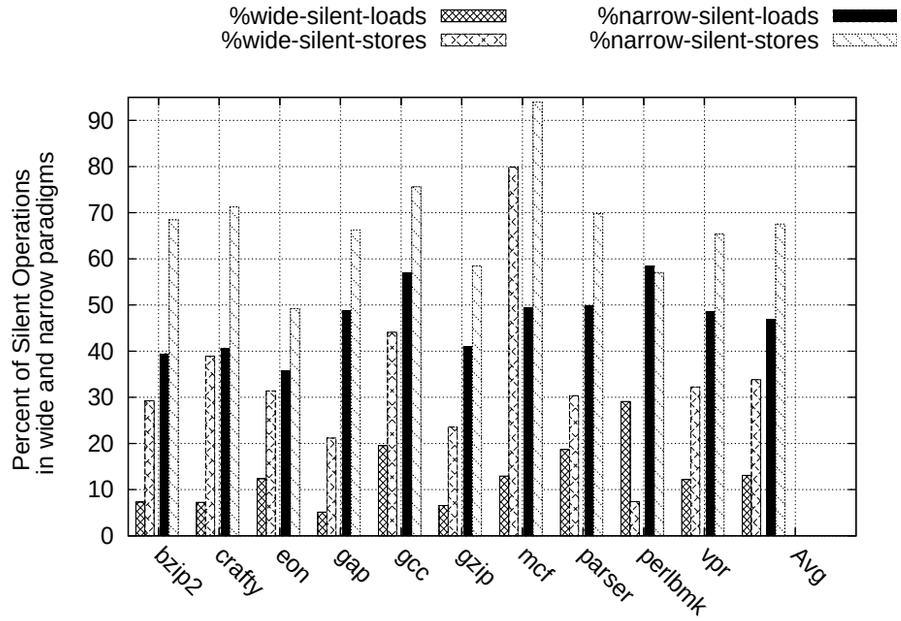

Figure 7.2: Silent memory operations in the narrow and wide paradigms

Further, Figure 7.2 illustrates the importance of following the definition of productiveness for *narrow* memory operations as compared to wide memory operations (measured for profile-phase[1] of each program). As can be seen, an average of 34% of all 64-bit wide stores are non-productive, and 13% of all wide 64-bit loads are non-productive. On the other hand, the

[1] first 200m x86 user instructions using the *ref* input data-set

narrow paradigm uncovers much more non-productive activity – about 67% of all 16-bit stores are non-productive and yet another 47% of all 16-bit loads are non-productive. Thus, the non-productive memory loads and stores together represent a significant 53% of the total narrow memory computations. On a parallel note, the notion of non-productive stores is the same as silent stores [2, 34].

Memory Productiveness Pruning. The foregoing evaluations also underline our hypothesis that narrow paradigm uncovers more opportunities to remove useless work. We have briefly investigated on the opportunity of pruning memory operations in this thesis. A naive extension of the Local Productiveness Pruning to memory operations has been proposed and evaluated as Memory Productiveness Pruning (MPP [3]). MPP exploits dual forms of speculation – apart from productiveness of memory operations, it also speculates on memory dependences and their predictability. Memory Dependence Speculation [13, 39, 40, 41] has been studied by previous researchers. MPP exploits this behavior of the predictability of memory dependences to remove memory-based asserts using a small additional hardware.

Evaluation of this optimization as a dynamic optimization reveals that it reduces the committed narrow memory computation stream by about 22% , while the optimal reduction possible is 28%. As compared to the non-optimized narrow stream, MPP achieves average reductions of 2.3% reduction in the number of cycles and 4.75% reduction in the overall number of narrow computations over the non-optimized stream.

Our evaluations suggest that MPP is effective in pruning the memory operations. But the hardware support required by MPP in its current form remains non-trivial and does not befit the overall goal of this thesis. More research is required in order to include memory operations via an affordable scheme.

7.2.2 Memory Dependences and Data-flow analysis

Memory Dependences in this thesis have been modeled conservatively. This has also been identified as a key roadblock in the Global Productiveness Propagation optimization. We believe more research and development in the regard of modeling memory dependences may prove useful. On an orthogonal note, even speculative memory dependence modeling techniques (like our attempt with MPP [3]) can also be easily accommodated as the code pruning optimizations are speculative anyway.

7.2.3 Coverage and Regions

The optimization region for GPP is chosen to be a complete function, whereas LPP works on individual basic blocks as atomic sections of code. The chosen basic blocks for LPP are

essentially the component basic blocks of the hot functions used for the GPP optimization. The expected code coverage of these hot functions still remains around 60%. It remains a caveat of our infrastructure that recursive functions could not be analyzed. Having said that, improving the code coverage remains one of the simplest ways of augmenting the efficacy of the optimizations.

Further, with respect to the notion of optimization region / atomic regions for the non-productiveness based pruning techniques, other granularities like loops may also be explored. More opportunities with respect to code pruning and reordering may arise in case of loops. Specifically, we have observed that there exists further opportunity to perform *Loop-invariant narrow code motion*. Loop-invariant narrow code motion can hoist narrow computations outside the body of a loop without affecting the semantics of the program. As a 64-bit computation has been broken down into smaller tasks of finer granularity, loop-invariant code motion has been observed to be potentially more efficacious on the narrow stream of computations than its wide counterpart.

Bibliography

[1] Pritpal S. Ahuja, Douglas W. Clark, and Anne Rogers. The performance impact of incomplete bypassing in processor pipelines. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, MICRO 28, pages 36–45, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press. ISBN 0-8186-7349-4.

[2] Gordon B. Bell, Kevin M. Lepak, and Mikko H. Lipasti. Characterization of silent stores. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, PACT '00, pages 133–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0622-4.

[3] Indu Bhagat, Enric Gibert, Jesús Sánchez, and Antonio González. Eliminating non-productive memory operations in narrow-bitwidth architectures. In *Proceedings of the 9th Workshop on Optimizations for DSP and Embedded Systems (ODES-9)*, ODES 9, 2011.

[4] David Brooks and Margaret Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*, pages 13 –22. IEEE, jan 1999.

[5] Mihai Budiu, Majd Sakr, Kip Walker, and Seth Copen Goldstein. Bitvalue inference: Detecting and exploiting narrow bitwidth computations. In *Euro-Par Conference on Parallel Processing, Proceedings from the 6th International*, pages 969–979, 2000.

[6] B. Calder, P. Feller, and A. Eustace. Value profiling. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 259–269. IEEE Computer Society, 1997.

[7] Brad Calder, Peter Feller, and Alan Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*, 1, 1999.

[8] R. Canal, A. González, and J.E. Smith. Very low power pipelines using significance compression. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 181–190. ACM, 2000.

[9] R. Canal, A. González, and J.E. Smith. Software-controlled operand-gating. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 125. IEEE Computer Society, 2004.

[10] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 227–238. IEEE Computer Society, 2006.

[11] Ben-Chung Cheng and Wen-Mei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 57–69, 2000. ISBN 1-58113-199-2.

[12] F.C. Cheng, S.H. Unger, and M. Theobald. Self-timed carry-lookahead adders. *Computers, IEEE Transactions on*, 49(7):659–672, 2000.

[13] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th annual international symposium on Computer architecture*, ISCA '98, pages 142–153, 1998. ISBN 0-8186-8491-7.

[14] Marcelo Cintra, José F. Martínez, and Josep Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 13–24, 2000. ISBN 1-58113-232-8.

[15] The Standard Performance Evaluation Corporation. Spec cpu2000 benchmarks. 2000.

[16] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. *Code Generation and Optimization, IEEE/ACM International Symposium on*, 0:15, 2003.

[17] Oguz Ergin, Deniz Balkan, Kanad Ghose, and Dmitry Ponomarev. Register packing: Exploiting narrow-width operands for reducing register file pressure. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 304–315, 2004. ISBN 0-7695-2126-6.

[18] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9:319–349, July 1987. ISSN 0164-0925.

[19] Krisztián Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy caches: simple techniques for reducing leakage power. In *Proceedings of the 29th annual international symposium on Computer architecture*, ISCA '02, pages 148–157, 2002. ISBN 0-7695-1605-X.

[20] PTLsim SPEC 2000 Benchmark Suite Steps for building and using the benchmarks. Ptlsim, http://www.ptlsim.org. 2006.

[21] Philip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, SIGPLAN '86, pages 11–16, 1986. ISBN 0-89791-197-0.

[22] Rajiv Gupta, Eduard Mehofer, and Youtao Zhang. A representation for bit section based analysis and optimization. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 62–77, London, UK, 2002. Springer-Verlag. ISBN 3-540-43369-4.

[23] J.L. Hennessy, D.A. Patterson, and D. Goldberg. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2006.

[24] Wen Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *The Journal of Supercomputing*, 7:229–248. ISSN 0920-8542.

[25] Intel. Intel®64 and ia-32 architectures software developer manuals, volume 1 : Basic architecture. 2007.

[26] Intel. Intel®64 and ia-32 architectures software developer manuals, volume 2a : Instruction set reference, a-m. 2007.

[27] Intel. Intel®64 and ia-32 architectures software developer manuals, volume 2b : Instruction set reference, a-m. 2007.

[28] V. Kathail, S. Aditya, R. Schreiber, B. Ramakrishna Rau, D.C. Cronquist, and M. Sivaraman. Pico: automatically designing custom computers. *Computer*, 35(9):39 – 47, sep 2002. ISSN 0018-9162. doi: 10.1109/MC.2002.1033026.

[29] Alexander Klaiber. The technology behind crusoe^{TM}processors, low-power x86-compatible processors implemented with code morphing^{TM}software. *White paper*, 2000.

[30] Masaaki Kondo and Hiroshi Nakamura. A small, fast and low-power register file by bit-partitioning. *High-Performance Computer Architecture, International Symposium on*, 0: 40–49, 2005. ISSN 1530-0897.

[31] William C. Kreahling, David Whalley, Mark W. Bailey, Xin Yuan, Gang-Ryung Uh, and Robert van Engelen. Branch elimination by condition merging. *Softw. Pract. Exper.*, 35: 51–74, January 2005. ISSN 0038-0644.

[32] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *SIGPLAN Not.*, 39:473–489, April 2004. ISSN 0362-1340.

[33] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 145–156, 2000. ISBN 1-58113-199-2.

[34] Kevin M. Lepak and Mikko H. Lipasti. Silent stores for free. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, pages 22–31, New York, NY, USA, 2000. ACM. ISBN 1-58113-196-8.

[35] Bengu Li and Rajiv Gupta. Bit section instruction set extension of arm for embedded applications. In *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '02, pages 69–78, New York, NY, USA, 2002. ACM.

[36] Gabriel H. Loh. Exploiting data-width locality to increase superscalar execution bandwidth. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 35, pages 395–405, 2002. ISBN 0-7695-1859-1.

[37] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood. Bitwidth cognizant architecture synthesis of custom hardware accelerators. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(11):1355–1371, 2001.

[38] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: log-based transactional memory. In *12th International Symposium on High-Performance Computer Architecture, HPCA-12 2006, Austin, Texas, February 11-15, 2006*, HPCA, pages 254–265. IEEE Computer Society, 2006.

[39] A. Moshovos and G.S. Sohi. Speculative memory cloaking and bypassing. *International Journal of Parallel Programming*, 27(6):427–456, 1999.

[40] A.I. Moshovos. *Memory dependence prediction*. PhD thesis, Citeseer, 1998.

[41] Andreas Moshovos and Gurindar S. Sohi. Read-after-read memory dependence prediction. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 32, pages 177–185, 1999. ISBN 0-7695-0437-X.

[42] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. Cacti 6.5. 2009.

[43] Emre Özer, Andy P. Nisbet, and David Gregg. Stochastic bit-width approximation using extreme value theory for customizable processors. In *Proceedings of the 13th International Conference on Compiler Construction*, CC '04, London, UK, 2004. Springer-Verlag.

[44] Emre Özer, Andy P. Nisbet, and David Gregg. A stochastic bitwidth estimation technique for compact and low-power custom processors. *ACM Trans. Embed. Comput. Syst.*, 7: 34:1–34:30, May 2008. ISSN 1539-9087.

[45] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. *SIGARCH Comput. Archit. News*, 25:206–218, May 1997. ISSN 0163-5964.

[46] S.J. Patel and S.S. Lumetta. replay: A hardware framework for dynamic optimization. *Computers, IEEE Transactions on*, 50(6):590 –608, jun 2001.

[47] Gilles Pokam, Olivier Rochecouste, André Seznec, and François Bodin. Speculative software management of datapath-width for energy optimization. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '04, pages 78–87, 2004. ISBN 1-58113-806-7.

[48] Rahul Razdan. *PRISC: programmable reduced instruction set computers*. PhD thesis, Cambridge, MA, USA, 1994. UMI Order No. GAX95-00124.

[49] Rahul Razdan and Michael D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th annual international symposium on Microarchitecture*, MICRO 27, pages 172–180, 1994. ISBN 0-89791-707-3.

[50] Rahul Razdan, Karl S. Brace, and Michael D. Smith. Prisc software acceleration techniques. In *Proceedings of the1994 IEEE International Conference on Computer Design: VLSI in Computer & Processors*, ICCS '94, pages 145–149, 1994. ISBN 0-8186-6565-3.

[51] Robert Schreiber, Shail Aditya, Scott A. Mahlke, Vinod Kathail, B. Ramakrishna Rau, Darren C. Cronquist, and Mukund Sivaraman. Pico-npa: High-level synthesis of nonprogrammable hardware accelerators. *VLSI Signal Processing*, 31(2):127–142, 2002.

[52] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Flexible decoupled transactional memory support. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 139–150, 2008. ISBN 978-0-7695-3174-8.

[53] Darko Stefanovic and Margaret Martonosi. On availability of bit-narrow operations in general-purpose applications. In *Proceedings of the Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications*, FPL '00, pages 412–421, 2000. ISBN 3-540-67899-9.

[54] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 1–12, 2000. ISBN 1-58113-232-8.

[55] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bidwidth analysis with application to silicon compilation. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 108–120, 2000. ISBN 1-58113-199-2.

[56] Sriraman Tallam and Rajiv Gupta. Bitwidth aware global register allocation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pages 85–96, 2003. ISBN 1-58113-628-5.

[57] Benchmarking tools and assessment environment for configurable computing: benchmark specification documentversatility stressmark. Submitted by honeywell technology center to usa intelligence center and fort huachuca under contract no. dabt63-96-c-0085. 1999.

[58] Perry H. Wang, Jamison D. Collins, Christopher T. Weaver, Blliappa Kuttanna, Shahram Salamian, Gautham N. Chinya, Ethan Schuchman, Oliver Schilling, Thorsten Doil, Sebastian Steibl, and Hong Wang. Intel®atom™processor core made fpga-synthesizable. In *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '09, pages 209–218, 2009. ISBN 978-1-60558-410-2.

[59] Fu-Ching Yang and Ing-Jer Huang. An embedded low power/cost 16-bit data/instruction microprocessor compatible with arm7 software tools. In *Design Automation Conference, 2007. ASP-DAC '07. Asia and South Pacific*, pages 902–907, 2007.

[60] Minghui Yang, Gang-Ryung Uh, and David B. Whalley. Improving performance by branch reordering. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 130–141, 1998. ISBN 0-89791-987-4.

[61] Minghui Yang, Gang-Ryung Uh, and David B. Whalley. Efficient and effective branch reordering using profile data. *ACM Trans. Program. Lang. Syst.*, 24:667–697, November 2002. ISSN 0164-0925.

[62] M.T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 23–34. IEEE, 2007.