UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES
université PARIS-SACLAY

Université de Versailles Saint-Quentin-en-Yvelines — École Doctorale "STV"

Combiner Approches Statique et Dynamique pour Modéliser la Performance de Boucles HPC

Combining Static and Dynamic Approaches to Model Loop Performance in HPC

THÈSE

présentée et soutenue publiquement le 21 Septembre 2015

pour l'obtention du

Doctorat de l'Université de Versailles Saint-Quentin-en-Yvelines

(spécialité informatique)

par

Vincent Palomares

Composition du jury

Président :	François Bodin	- Professeur, Université de Rennes
Rapporteurs :	Denis Barthou	- Professeur, Université de Bordeaux
	Henri-Pierre Charles	- Directeur de Recherche, CEA
Examinateurs :	Alexandre Farcy	- CPU Architect, Intel
	David J. Kuck	- Fellow, Intel
Directeur de thèse :	William Jalby	- Professeur, Université de Versailles

PERFCLOUD

COLOC

Thanks

I would first like to thank William Jalby, my advisor, for the help and guidance he provided throughout these past few years. His healthy doses of both optimism and skepticism motivated me to keep aiming higher.

I would also like to thank David Wong and David Kuck, with whom collaborating was very pleasant and stimulating. Working on Cape with them was an enriching experience.

I am grateful to Denis Barthou and Henri-Pierre Charles, the reporters, for reviewing my work and making suggestions on how to improve my manuscript. Their input was very helpful and contributed to making this document clearer.

My thesis work was extremely fun, in no small part thanks to Zakaria Bendifallah and José Noudohouenou. We certainly had our share of good laughs in our office. Let's just hope the quotations we've collected along the years[1] never get in the wrong hands.

I would like to thank all the members of the lab for their dynamism, competence and friendliness. I do not want to make a comprehensive list here (for fear I may forget a name or two[2] and cause jealousy between those mentioned and those not). Surely, a blanket statement should be enough to prevent that?!
Some of them are however definitely worthy of a special mention. First, Emmanuel Oseret, with whom I often discussed microarchitectural details on Intel CPUs, and who agreed to implement some features in his CQA tool that were specially tailored to my needs. He also proofread my manuscript, helping me rid it of various mistakes. I also want to mention Mathieu Tribalat, who helped me navigate some of MAQAO's intricacies, and whose work in taking over DECAN made getting application measurements considerably easier.

The end-of-thesis rush and the tension that came with it were made easier by chatting (and complaining!) with other students in the same situation. I want to thank Nicolas Triquenaux for finding and sharing information about thesis completion procedures, and Zakaria (once again), whose perpetual struggle with paperwork made it easier to see mine was quite mild after all.

On a more personal side, I would like to thank Anna Galusza, my fiancée, for supporting me throughout the writing of this manuscript and reviewing some of its key parts, as well as my parents and the rest of my family for helping me get this far.

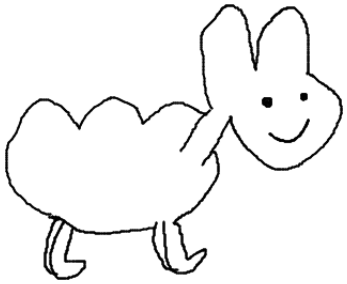
Finally, I would like to thank you, the reader: you are the reason why I wrote this manuscript[3]. This is particularly true if you read it until the end, at which point you should feel free to fill the following blank with your name[4]:
THANK YOU, !

[1] Some quotations? What quotations? Move along, there is nothing to see here!

[2] Or three...

[3] You, and getting my Ph.D., that is.

[4] NB: Not all libraries are very fond of this practice, so you might want to be discreet if using a borrowed copy.



Fluffy Canary
I am told I can do anything I want with the Thanks *section...*
Let's see if it's true!

Résumé:

La complexité des CPUs s'est accrue considérablement depuis leurs débuts, introduisant des mécanismes comme le renommage de registres, l'exécution dans le désordre, la vectorisation, les préfetchers et les environnements multi-coeurs pour améliorer les performances avec chaque nouvelle génération de processeurs. Cependant, la difficulté a suivi la même tendance pour ce qui est a) d'utiliser ces mêmes mécanismes à leur plein potentiel, b) d'évaluer si un programme utilise une machine correctement, ou c) de savoir si le design d'un processeur répond bien aux besoins des utilisateurs.

Cette thèse porte sur l'amélioration de l'observabilité des facteurs limitants dans les boucles de calcul intensif, ainsi que leurs interactions au sein de microarchitectures modernes.

Nous introduirons d'abord un framework combinant CQA et DECAN (des outils d'analyse respectivement statique et dynamique) pour obtenir des métriques détaillées de performance sur des petits codelets et dans divers scénarios d'exécution.

Nous présenterons ensuite PAMDA, une méthodologie d'analyse de performance tirant partie de l'analyse de codelets pour détecter d'éventuels problèmes de performance dans des applications de calcul à haute performance et en guider la résolution.

Un travail permettant au modèle linéaire Cape de couvrir la microarchitecture Sandy Bridge de façon détaillée sera décrit, lui donnant plus de flexibilité pour effectuer du codesign matériel / logiciel. Il sera mis en pratique dans VP3, un outil évaluant les gains de performance atteignables en vectorisant des boucles.

Nous décrirons finalement UFS, une approche combinant analyse statique et simulation au cycle près pour permettre l'estimation rapide du temps d'exécution d'une boucle en prenant en compte certaines des limites de l'exécution en désordre dans des microarchitectures modernes.

Mots Clés: codelet, analyse de boucle, analyse statique, analyse dynamique, calcul intensif, HPC, optimisation, modélisation rapide, performance, exécution dans le désordre, simulation au cycle près

Abstract:

The complexity of CPUs has increased considerably since their beginnings, introducing mechanisms such as register renaming, out-of-order execution, vectorization, prefetchers and multi-core environments to keep performance rising with each product generation. However, so has the difficulty in making proper use of all these mechanisms, or even evaluating whether one's program makes good use of a machine, whether users' needs match a CPU's design, or, for CPU architects, knowing how each feature really affects customers.

This thesis focuses on increasing the observability of potential bottlenecks in HPC computational loops and how they relate to each other in modern microarchitectures.

We will first introduce a framework combining CQA and DECAN (respectively static and dynamic analysis tools) to get detailed performance metrics on small codelets in various execution scenarios.

We will then present PAMDA, a performance analysis methodology leveraging elements obtained from codelet analysis to detect potential performance problems in HPC applications and help resolve them.

A work extending the Cape linear model to better cover Sandy Bridge and give it more flexibility for HW/SW codesign purposes will also be described. It will be directly used in VP3, a tool evaluating the performance gains vectorizing loops could provide.

Finally, we will describe UFS, an approach combining static analysis and cycle-accurate simulation to very quickly estimate a loop's execution time while accounting for out-of-order limitations in modern CPUs.

Keywords: codelet, loop analysis, static analysis, dynamic analysis, HPC, optimization, fast modeling, performance, out-of-order, cycle-accurate simulation

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The growing complexity behind modern CPU microarchitectures [1, 2] makes performance evaluation and modeling a very complex task. Indeed, modern CPUs will typically implement features such as pipelining, register renaming, speculative and out-of-order execution, data prefetchers, vectorization, virtual memory, caches and multiple execution cores. While each of them can be beneficial to performance, they also make performance analysis more difficult.

On the consumer side of the CPU design process, users want to know which product fits their needs best in terms of performance, energy consumption and/or price. Software developers will be more concerned with adjusting their applications to make the best use of existing features, especially when performance represents a direct competitive advantage.

On the designer side, CPU manufacturers need to build microarchitectures offering the characteristics wanted by users while keeping costs low. Furthermore, as product improvements may have to rely on complex mechanisms, they have to guide software developers on how to use them while also having the contrary objective of preventing competitor plagia by controlling the exposure of their performance recipes.

In this context, performance modeling can be used to cost-effectively:

1. Help users find out which hardware would best fit their applications (without actually buying all the considered hardware first).
2. Expose optimization opportunities to software developers (without first testing them).
3. Offer CPU architects insights on which hardware improvements would speed user applications the most (without first implementing them).

This chapter will describe why performance modeling is important in the field of High Performance Computing (HPC) and proceed to present the objectives and contributions of this thesis. It will also provide a quick overview of the document.

1.1 High Performance Computing (HPC)

HPC represents the use of large-scale machines called *supercomputers* to process compute workloads extremely quickly. It is used (and needed) in areas as diverse as aerodynamic simulations, cryptanalysis, engine design, oil and gas exploration, molecular dynamics or weather forecasting.

It is an environment with very interesting characteristics:

1. Performance is a primary objective and can result in hefty monetary gains. For instance, a faster numerical simulator will be able to provide more results, or/and results of a better quality, in domains as various as the design of cars, plane wings, nuclear plants or the development of new drugs.

2. The used supercomputers can have millions of execution cores [3], offering potentially tremendous calculation speeds and making energy consumption an unavoidable (and expensive) concern: a poorly used machine is a costly machine.

3. Users and software developers can be strongly tied, or even be the same entities. It creates an interesting dynamic where developers are strongly motivated to optimize their code to make the best use of existing resources, and may also have a say in which machines to buy next.

As HPC applications typically spend very large amounts of time in computational loops due to processing large data sets, loop analysis is a primary go-to approach for HPC performance analysis, optimization and modeling.

1.2 Objectives and Contributions

This thesis focuses on increasing the cost-effective observability of potential bottlenecks in HPC computational loops and how they relate to each other. It aims to do so by combining static and dynamic approaches to identify, quantify, and model both the bottlenecks and their interactions.

Its main contributions are:

1. PAMDA, a performance evaluation methodology using a blend of static and dynamic analyses to find bottlenecks and quantify their impact. Its main purpose is to expose optimization opportunities.

2. An adaptation of the Cape linear model to the Sandy Bridge microarchitecture as well as a direct application thereof with VP3, a vectorization gain predictor.

3. Uop Flow Simulation (UFS), a loop performance modeling technique combining static analysis and cycle-level simulation to account for out-of-order limitations at a very low execution cost.

Other less significant contributions include:

1. A loop performance measurement framework combining static and dynamic analysis tools to evaluate loop performance from different angles.

2. An empirical approach to quantify out-of-order resources.

1.3 Overview

Chapter 2 will present some background information to familiarize readers with CPU microarchitectural details and nomenclature, performance analysis approaches and tools, as well as with the use of small benchmarks called *codelets*.

We will introduce a framework combining CQA and DECAN (respectively static and dynamic analysis tools) in Chapter 3. Its objective is to get detailed performance metrics on small codelets given various execution scenarios.

We will then present PAMDA, a performance analysis methodology, in Chapter 4. It leverages elements obtained from codelet analysis to detect potential performance problems in HPC applications and help resolve them.

Chapter 5 will describe a work extending the Cape linear model to better cover Sandy Bridge and give it more flexibility for HW/SW codesign purposes. It will be directly used in Chapter 6 with VP3, a tool evaluating the performance gains vectorizing loops could provide.

Chapter 7 will introduce UFS, an approach combining static analysis and cycle-accurate simulation to very quickly estimate a loop's execution time while accounting for out-of-order limitations in modern CPUs, and better identifying out-of-order related issues than PAMDA or Cape modeling.

We will finally conclude in Chapter 8, summarizing our contributions and presenting future work.

Chapter 2

Background

This chapter will focus on presenting the technical context for this thesis and introduce some of the nomenclature used throughout this manuscript.

It will first describe modern Intel microarchitectures detailedly, before presenting different performance analysis approaches and tools. It will also explain some of the advantages and limitations of *codelets*, small benchmarks which we will use for modeling purposes in later chapters.

2.1 Recent Microarchitectures

Microarchitectures are the result of different design choices and incremental improvements carried over CPU generations. They are typically pipelined and feature the following components:

1. Front-End (FE): component of reading and decoding instructions, making them available to the rest of the execution pipeline.
2. Back-End (BE): executes the instructions provided by the Front-End.
3. Memory Hierarchy: caches can be used to improve the effective speed of memory accesses for both data and instructions.

We will present some of the microarchitectures particularly relevant to HPC here, using information from official sources [4, 5, 6, 7, 8], technical news articles [9, 10, 11, 12, 13, 14, 15], test-based reports [16] and our own observations.

2.1.1 Sandy Bridge

Sandy Bridge (SNB) is a microarchitecture used in the performance-oriented *Big Core* family of Intel CPUs. It is a *tock* in the manufacturer's *tick-tock* development cycle [17], meaning it keeps the same 32 nm lithography as its *Westmere* predecessor, but brings important microarchitectural changes.

It will be the microarchitecture this thesis most focuses on. We will present it in details and later use it as a base point to describe the incremental improvements brought by its *Ivy Bridge* and *Haswell* successors.

SNB Stock Keeping Units (SKUs) can have from 1 to 6 cores.

2.1.1.1 Front-End

Sandy Bridge's decode pipeline (also called *legacy decode pipeline*) is in charge of fetching instructions from the memory hierarchy and decoding them, producing uops more easily interpretable by the Back-End. It is the component the most directly affected by the complexity of the x86 instruction sets, and can produce up to 16 bytes of uop *or* 4 uops per cycle, whichever is the most restrictive. Furthermore, it has branch prediction abilities, and can decode and provide uops speculatively.

While it can typically only decode up to 4 instructions per cycle, it implements extra features to increase its effective bandwidth:

1. Macrofusion: allows a simple integer instruction and a following branch instruction to be decoded as a single uop in certain circumstances. This actually brings the maximum theoretical number of decoded instructions to 5 per cycle in favorable corner cases.

2. Microfusion: complex instructions may need to get divided in smaller logical operations (or *components*) when decoded to simplify the Back-End's work. Microfusion allows instructions having both an arithmetic and a memory components to be fit in a single uop for part of the pipeline despite this constraint, potentially doubling the effective FE bandwidth. For instance, *MULPD (%rax), %xmm0* will occupy a single uop slot until each component needs to be executed separately.

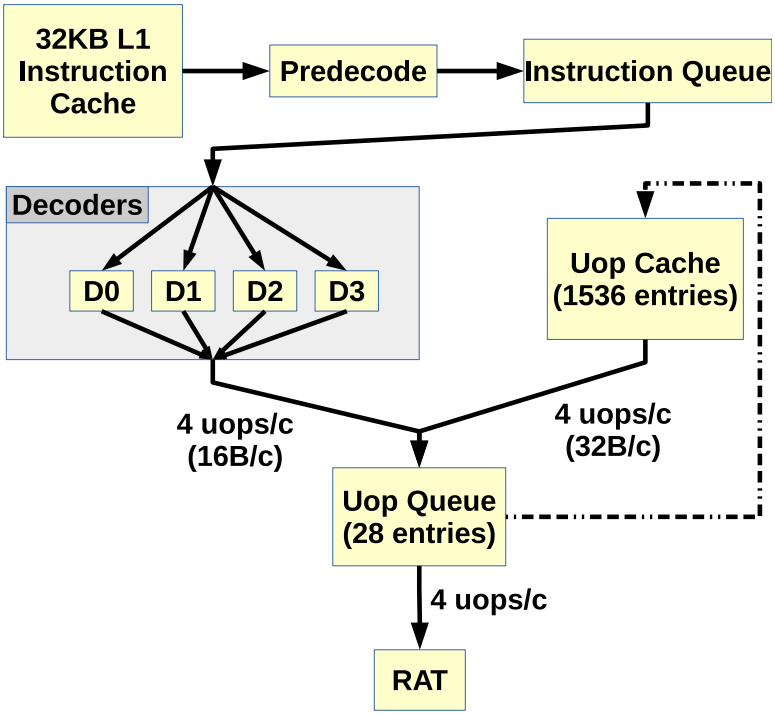


Figure 2.1: Simplified Sandy Bridge Front-End

Sandy Bridge's Front-End can produce up to 4 uops using any of three different generation mechanisms: the decoders (legacy pipeline), the Uop Cache and the Uop Queue (when iterating over small loops). Only one uop source may be active at a time.

Furthermore, as decoding is a slow and expensive process, Intel CPU architects designed extra mechanisms to prevent instructions from having to be constantly re-decoded, reducing the pressure on the legacy pipeline and increasing the FE's bandwidth (see Figure 2.1):

1. The *uop queue*: queues uops right before the RAT, allowing some FE or Back-End stalls to be absorbed. A loop detection mechanism called *Loop Stream Detector* detects when uops currently in the queue are part of a loop, and can decide to a) stop taking uops from the legacy pipeline, b) not destroy the uops

it sends to the Back-End and c) replay them as many times as necessary.
While its peak bandwidth is still 4 uops per cycle, there is no limit on the number of transferred bytes anymore, increasing the effective FE bandwidth when lengthy uops are present.
It has a maximum capacity of 28 uops on SNB.

2. The *uop cache* (or *Decoded ICache*): it saves uops decoded by the legacy pipeline, and can serve as an alternative uop provider for the uop queue.
As with the legacy pipeline and the uop queue, its peak bandwidth is 4 uops per cycle, though with a maximum of 32 bytes of uop being generated per cycle.
It is extremely large compared to the uop queue's capacity and can contain up to 1536 uops in ideal conditions.

2.1.1.2 Execution Engine

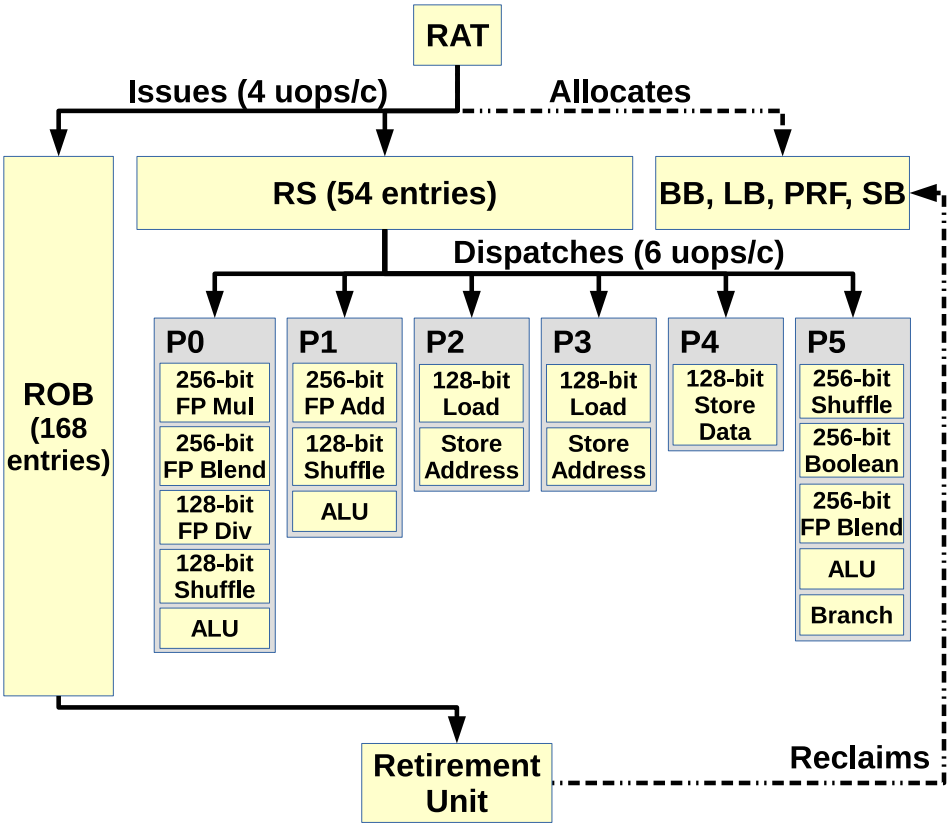


Figure 2.2: Simplified Sandy Bridge Execution Engine

The RAT issues *uops from the Front-End to the Back-End after having allocated the necessary out-of-order resources and renamed their operands.*

The ROB keeps track of all in-flight uops (both those pending execution and the ones waiting for retirement), while other resources are more specific (e.g. the LB keeps track of load entries). All resources are allocated at issue time and only reclaimed at retirement, with the notable exception of RS entries (which are released once uops are dispatched to compatible execution ports).

The RS can dispatch up to 6 uops per cycle (one to each port) out-of-order.

The Resource Allocation Table (RAT) is the gateway component between the

Front-End the FE to the Back-End. It *issues* uops in-order, performs register renaming and allocates the resources necessary to their out-of-order execution. While all uops need an entry in the ReOrder Buffer (ROB) to be issued, other out-of-order buffers are allocated on a per-case basis.
Interestingly, not all uops need to be sent to the Reservation Station to wait for execution: Sandy Bridge processes *nop* uops (which have no input nor output) and *zero-idioms* (whose output is always zero, and hence have no relevant input) directly in the RAT.
Other resources such as the Branch Buffer, Load Buffer, Physical Registers and Store buffer are intuitively only allocated for respectively branches, loads and software prefetches, uops with a register output and stores.
Furthermore, with the exception of Reservation Station entries, resources are only released at the retirement step.

The Reservation Station holds uops until their input operands are ready, and then *dispatches* them to adequate execution ports. The latter will forward them to the proper Functional Units where they will begin their *execution*.

Most Functional Units are fully pipelined, often giving them a throughput of 1 uop per cycle.

Fully executed uops are *retired* in-order, at which point their output is committed to the architectural state and their resources freed.

Figure 2.2 summarizes our description of Sandy Bridge's execution engine.

2.1.1.3 Memory Hierarchy

The role of the memory hierarchy is to dampen the impact of RAM's limited bandwidth and latency by acting as intermediaries to the main (RAM) memory. Indeed, caches can be much faster than RAM in both regards due to their being much smaller: as a general rule, the smaller the memory unit is, the faster it can perform. CPUs may consequently have several levels of cache, each offering different levels of capacity and performance.

Sandy Bridge's memory hierarchy is summarized in Figure 2.3.

Load and Store Units can each transfer up to 16 bytes from/to the L1 data cache per cycle, though there are 2 of the former and only 1 of the latter. While they can work concurrently (for an aggregated bandwidth of 24 bytes per cycle), this can only be achieved when using AVX 32-byte vector transfers due to port restrictions (32-byte transfers keep memory units busy for 2 cycles, allowing store address uops to use ports 2 and 3 without penalizing loads).

All of Sandy Bridge's data caches are write-back: upper cache levels are only made aware of memory writes (or *stores*) when cache lines from lower levels are evicted. Sandy Bridge's 32-KB L1 data cache is 8-way associative and virtually indexed. Its 256-KB L2 cache also has an associativity of 8, but is physically indexed and interestingly neither inclusive nor exclusive in regards to L1. The L3's size is SKU-dependent and can range from 1 to 20 MB. Its associativity is also variable, and is between 12 and 16 depending on the model.

All three cache levels use an NRU (*Not Recently Used*, a variant of *Least Recently Used*) replacement policy.

Four data prefetchers are also present, whose role is to predict which cache lines are going to be needed in the future and request them ahead of time:

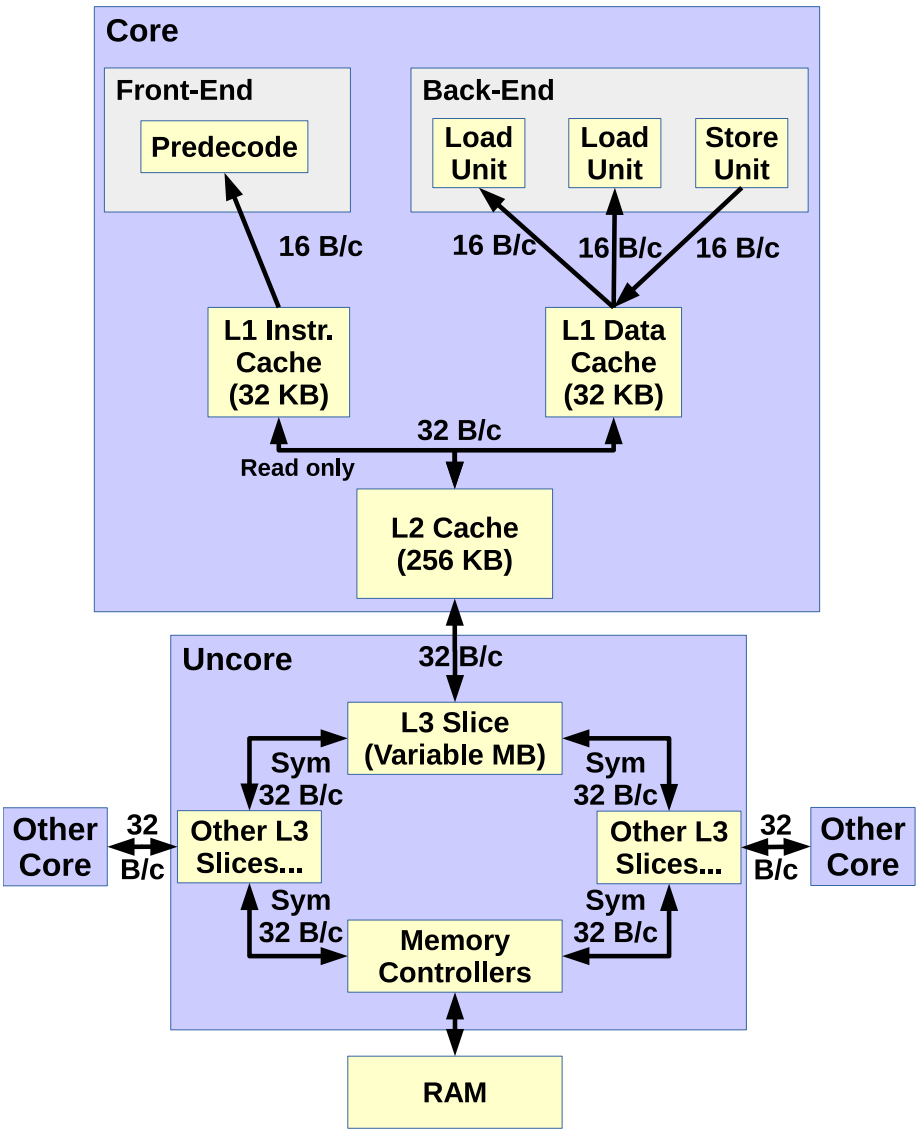


Figure 2.3: Simplified Sandy Bridge Memory Hierarchy

The data L1 has a combined bandwidth of 48 bytes per cycle, as load and store accesses can be performed in parallel. The L2's 32 bytes per cycle bandwidth is shared for all fetch and store accesses from the L1s. However, only the data L1 can write data back to the L2. Each L3 slice also has a dedicated bandwidth of 32 bytes per cycle which is shared for read and write accesses from the L2.

The L3 is distributed over all the cores, allowing each core to have their own dedicated access to L3. The bi-directional data ring connecting the slices allows each core to access the entirety of L3, though latency may vary a bit depending on far the relevant slice is.

L3 slices share the same memory controllers for RAM accesses.

1. The *DCU Prefetcher* (operates in L1): detects ascending-address loads within the same cache line and fetches the following cache line.

2. The *Instruction-Pointer-based Prefetcher* (operates in L1): detects access stride patterns for individual load instructions and fetches cache lines accordingly.

3. The *Spatial Prefetcher* (or *Adjacent Cache Line Prefetcher*; operates in L2): pairs contiguous cache lines in 128-byte blocks. Accesses to the first cache line trigger the fetching of the whole block.

4. The *Stream Prefetcher* (operates in L2): tries to predict and fetch futurely-used cache lines based on previous accessed addresses. It can keep track of up to 32 different access patterns.

Sandy Bridge has a 2-level Translation Lookaside Buffer system:

1. The L1-TLB is 4-way associative, and can contain up to 64 4KB page entries, 32 2MB entries and 4 1GB entries.

2. The L2-TLB is also 4-way associative, and can contain up to 512 4KB page entries. It cannot hold larger page entries.

2.1.2 Ivy Bridge

Ivy Bridge is the tick improvement of Sandy Bridge, carrying the microarchitecture to a 22 nm lithography but only bringing moderate microarchitectural changes.

IVB CPUs feature from 1 to 15 cores.

2.1.2.1 Front-End

Sandy Bridge's uses two physical 28-entry uop queues to support hyper-threading. Ivy Bridge improves over it by fusing the queues into a single physical one with 56 entries, and virtually splitting it only when hyper-threading is actually used. It improves its ability to absorb pipeline stalls and increases the maximum size of loops replayable with the Loop Stream Detector, helping improve performance and lower power consumption.

2.1.2.2 Execution Engine

Ivy Bridge introduces 0-latency register moves: in some cases, register moves can be achieved by merely making the named register point to the source physical register, which can be done by the RAT.

The architects also improved the divider / square root unit, likely taking advantage of the finer lithography to widen it and improve its bandwidth significantly (as well as its latency to a lesser extent).

2.1.2.3 Memory Hierarchy

While the sizes of L1 and L2 are the same for IVB as for SNB, the maximum L3 size was increased from 20 MB to 37.5.

Furthermore, L3 seems to use an adaptive replacement policy [18].

Ivy Bridge also introduces the *Next-Page Prefetcher* (NPP), which detects memory accesses nearing the beginning or the end of a page to fetch the matching page translation entry. It is not clear whether the NPP fetches entries to the L2 data cache or directly to the TLBs, though Intel patent [19] suggests the latter.

2.1.3 Haswell

Haswell is the tock after Ivy Bridge, focusing once again on microarchitectural changes and still using the same 22 nm lithography.

2.1.3.1 Front-End

Haswell's Front-End is largely the same as Ivy Bridge's.

2.1.3.2 Execution Engine

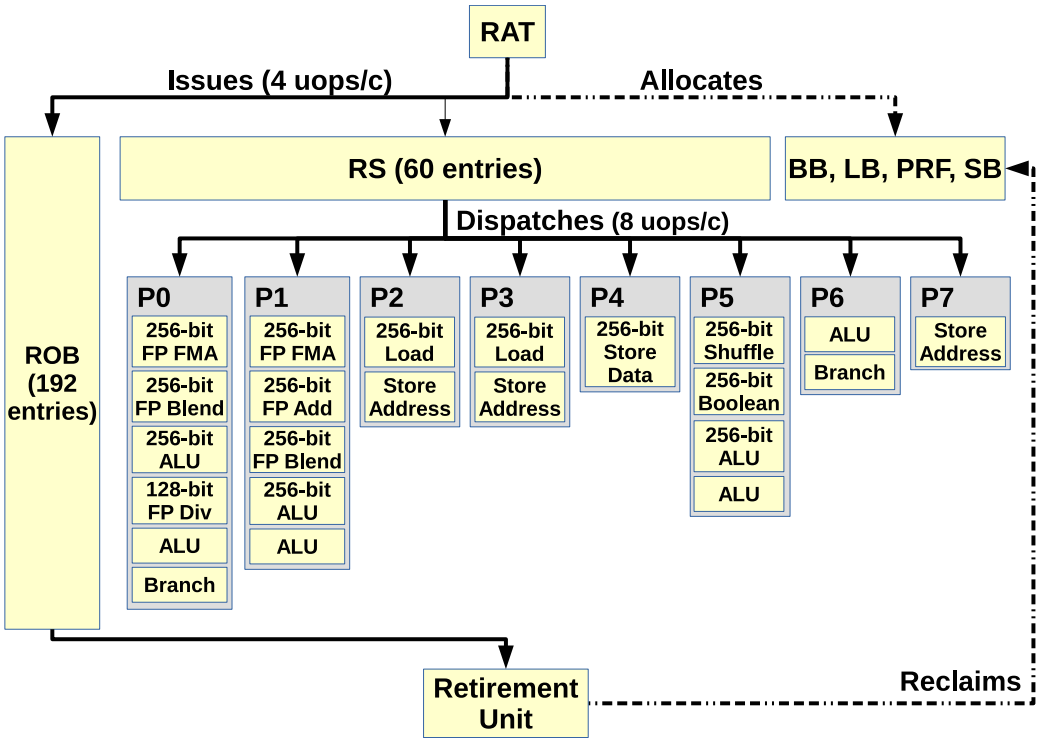


Figure 2.4: Simplified Haswell Execution Engine

The Haswell execution engine brings various improvements over Sandy Bridge, such as more execution ports, new Fused Multiply-Add functional units, larger out-of-order buffers and memory units able to process 32-byte transfers in a single cycle.

The execution engine has some important changes, which are summarized in Figure 2.4. Some of the most important ones include:

1. There now being 8 dispatch ports (against 6 previously), one of which is dedicated to handle store address uops.

2. Pipelined Fused Multiply-Add units being placed behind ports 0 and 1, doubling the potential number of FP operations per cycle. Indeed, they can each execute vector operations such as $result = vector1 * vector2 + vector3$ with a throughput of 1 per cycle (note: in Haswell's implementation, the result register must be one of the inputs).

3. The sizes of most out-of-order resources are increased.

2.1.3.3 Memory Hierarchy

The L1 bandwidth is doubled, allowing the two load units and the store unit to each transfer up to 32 bytes per cycle (for a combined bandwidth of 96 bytes per cycle).

The L2 bandwidth was also doubled, allowing a full cache line (64 bytes) to be transferred between L1 and L2 every cycle.

The minimum and maximum L3 sizes were increased to reach respectively 2 MB and 45 MB.

Furthermore, some SKUs are equipped with *Crystalwell* embedded DRAM acting as a 128MB L4 victim cache, providing important bandwidth and latency bonuses.

2.1.4 Silvermont

Unlike SNB, IVB and HSW, *Silvermont* (SLM) is a microarchitecture used for *Atom* processors, for which emphasis is on low power consumption. Its *Bay Trail* variant targets the mobile sector, while *Avoton* micro-server versions were also designed.

Silvermont uses a 22 nm lithography, just like main line processor, and comprises between 1 (in Bay Trail) and 8 cores (in Avoton).

It is a particularly interesting x86 microarchitecture due to how energy consumption considerations impacted its design. Furthermore, future high-performance *Knights Landing* chips will feature around 70 Silvermont-inspired cores, making it very relevant in the HPC sphere.

2.1.4.1 Front-End

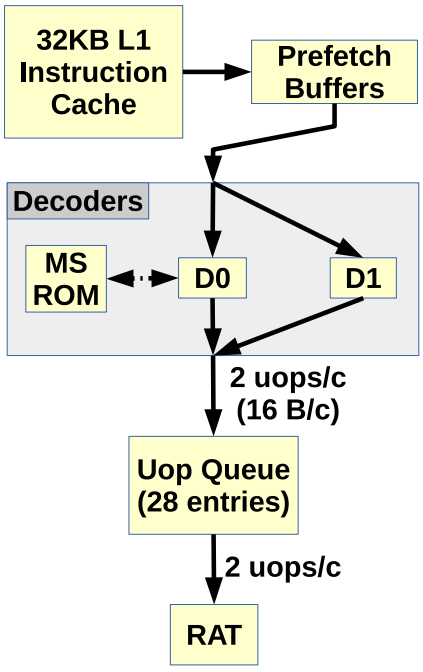


Figure 2.5: Simplified Silvermont Front-End

Silvermont's Front-End can provide the RAT with up to 2 uops per cycle. However, its decoders are limited, and D1 *can only decode simple instructions.*

A feature called Loop Stream Detector *can help compensate for the decoders' weaknesses when executing loops with small loop bodies, pinning down uops in the Uop Queue and replaying them for as long as possible. The Front-End can then consistently reach its peak bandwidth.*

Silvermont's Front-End supports speculative execution and branch prediction. It is 2-wide (see Figure 2.5), meaning it can decode and provide up to 2 uops to the Back-End per cycle. This is twice less than what Big Core microarchitectures can

do, and is further aggravated by SLM's individual instruction decoders being less potent than those in the main line products. They are however more power-efficient.

There is also a Loop Stream Detector in the uop queue, which takes a very high importance due to the decoders' weaknesses. While for SNB/IVB/HSW using the LSD is done rather opportunistically, it is an important factor for SLM performance.

2.1.4.2 Execution Engine

The *RAT / Allocation / Rename* cluster bridges the FE with the BE, inserting uops in-order after a) allocating some of the necessary resources for their execution and b) renaming their input and output registers.

All uops need an entry in the ROB. It is also not clear when other resources (e.g. Load Buffer) are allocated, as Intel claims Silvermont uses a late allocation / early resource reclamation scheme [20].

The scheduler system is distributed over different Reservation Stations, each handling a specific execution port (see Figure 2.6), and their own in-order or out-of-order dispatch policy:

1. FP RS 0: handles FP and vector additions, as well as some other arithmetic and logic operations. Dispatches uops in-order (only in regards to uops in FP-RS-1).

2. FP RS 1: handles FP and vector multiplications, divisions, shuffles and other operations. Dispatches uops in-order (only in regards to uops in FP-RS-2).

3. Int RS 0: handles integer arithmetic, logic and shift operations. Dispatches uops out-of-order.

4. Int RS 1: handles integer arithmetic and logic as well as branches. Dispatches uops out-of-order.

5. Mem RS: handles address generations, loads and stores. Dispatches uops in-order, but allows accesses to complete out-of-order to absorb latency.

Fully executed uops are retired and committed in-order.

2.1.4.3 Memory Hierarchy

Silvermont has a 24KB L1 with an associativity of 6. The load and store units can transfer up to 16 bytes per cycle from/to it, though they likely cannot do so simultaneously (as only one address per cycle can be generated for loads and stores). It adopts a random cache line replacement policy.

Single-core Silvermont SKUs have a 512KB L2.
SKUs with more than 1 core are organized in pairs of cores called *modules*, with each module having 1 MB of dedicated 16-way associative L2. The overall cache size for 8-core Silvermont CPUs hence reaches 4 MB, though each core is constrained to only use the L2 slice from its own package.
The L1 and the L2 can exchange up to 32 bytes per cycle, though this link is shared with all the cores in the package. It uses an NRU cache line replacement policy.

The L1 and L2 caches are both write-back.

There are two data prefetchers, the L1 Spatial Prefetcher and an L2 "advanced" prefetcher. They are likely respectively inspired by the L1 DCU and the L2 Streamer prefetchers from the Big Core CPUs, but not many details are given.

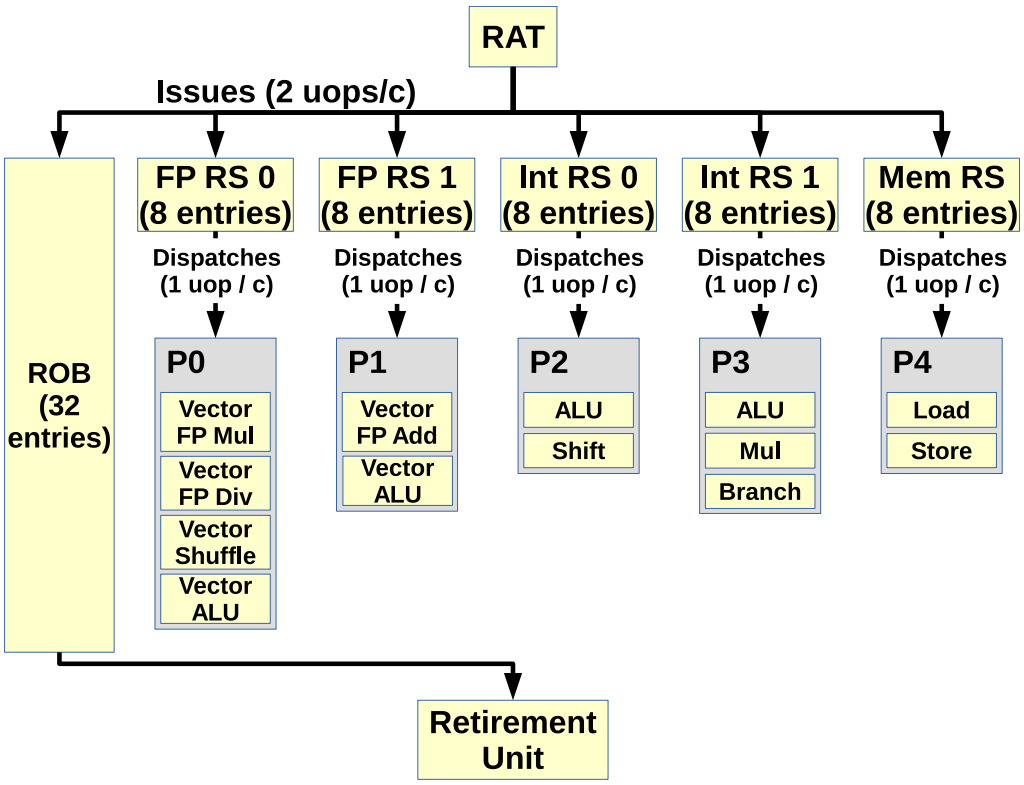


Figure 2.6: Simplified Silvermont Execution Engine

Unlike Big Core microarchitecture, there is no unified reservation station. Furthermore, each small RS acts with its own rules:

1. *The FP Reservation Stations have to dispatch their uops in-order.*
2. *The Mem RS also has to dispatch uops in-order (to help memory scheduling be as simple as possible), but they are non-blocking and can be be completed out-of-order.*
3. *Integer Reservation Stations can dispatch uops out-of-order.*

The ports presented here might not actually have a discrete existence, (as absent from any official documentation we could find), but we decided to put them here anyway to simplify the figure. We labelled them in a manner consistent with Big Core CPUs.

Silvermont also has a 2-level TLB cache:

1. The L1 TLB has 48 4KB entries and is fully associative.
2. The L2 TLB has 128 4KB entries and 16 2MB entries, and is 4-way associative. It is not inclusive (nor exclusive) in regards to the L1.

Figure 2.7 summarizes this memory hierarchy.

2.2 Performance Analysis

Different approaches can be adopted to analyze performance, each working with their own tradeoffs. This section will present some of them.

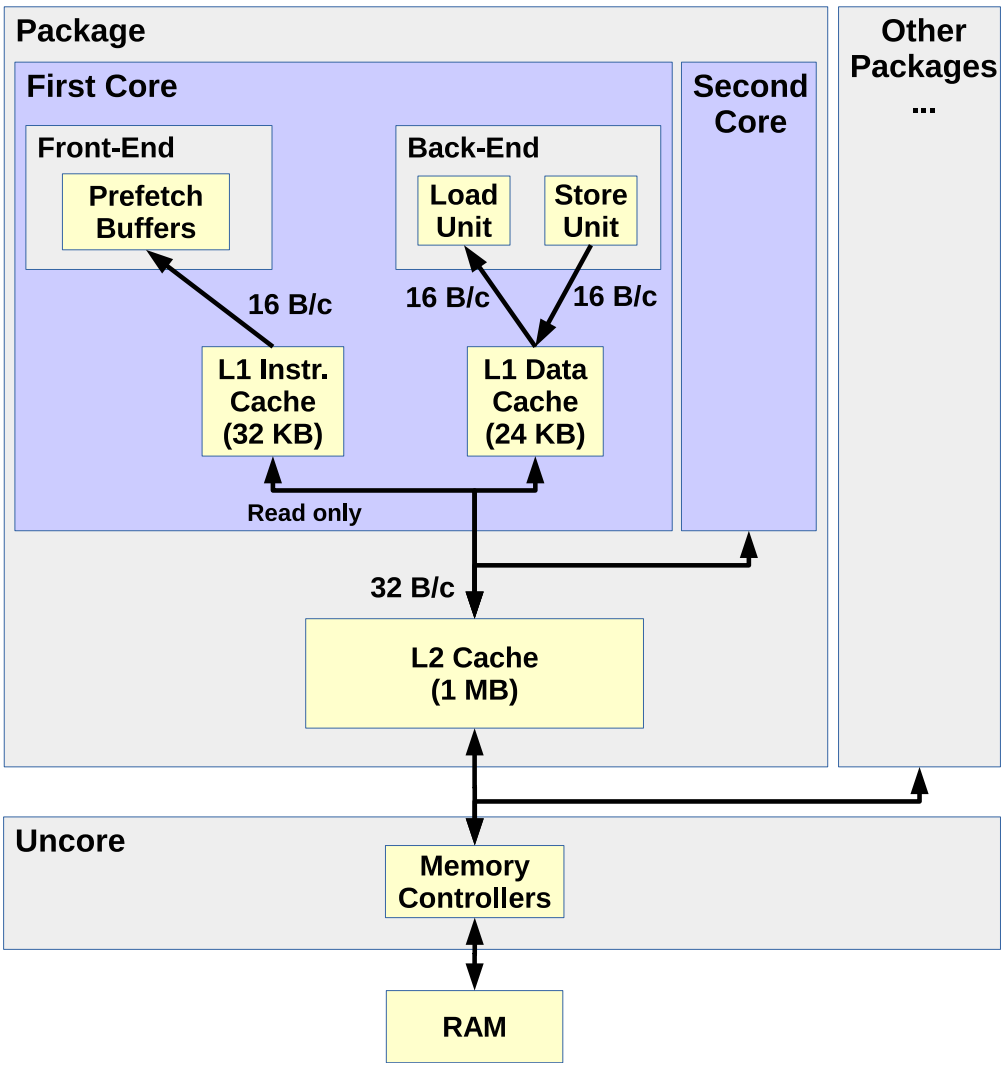


Figure 2.7: Simplified Silvermont Memory Hierarchy

The Load and the Store units cannot access L1 in parallel, so the data L1's bandwidth is of 16 bytes per cycle.

The L2 cache space and bandwidth are shared between cores from a same package.

Finally, the memory controllers are shared by all cores on the CPU.

2.2.1 Static Analysis

Static analysis consists in evaluating software without executing it. This offers the advantage of being particularly fast, at the cost of working with a limited amount of information.

We will present tools using this approach in this section.

2.2.1.1 Code Quality Analyzer (CQA)

CQA [21] evaluates the quality of assembly loops and projects their potential peak performance. It is developed inside the MAQAO [22] framework, which handles both the disassembling of target compiled binaries and the detection of the loops within.

Operating at the assembly level provides several advantages:

1. The evaluated code is the code executed by the machine. This is not (necessarily) when working at the source level due to the optimizations the compiler can apply.

2. Loop instructions can be tightly coupled to known microarchitecture features and functional units, making performance estimates solidly grounded in reality.

However, assembly-level static analysis does not (consistently) allow to detect and account for issues such as poor memory strides, which could have been observed in the source code.

CQA's metrics include:

1. Peak performance in L1 (assuming no memory-related issues, infinite-size buffers and an infinite number of iterations).

2. Front-End performance.

3. Distribution of the workload across the different ports and functional units.

4. Vectorization-efficiency metrics.

5. Impact of inter-iteration dependencies.

CQA can also provide suggestions on how to fix detected performance problems.

2.2.1.2 Intel Architecture Code Analyzer (IACA)

IACA is also a static analysis tool working at the assembly level. Unlike CQA, it uses markers inserted at the source level to locate the code to analyze. It allows users to easily target the parts they want analyzed, but forces them to place these markers and recompile their application.

Two types of analyses are possible:

1. Throughput analysis: the target block of code is treated as if it were a loop for the purpose of detecting inter- iteration dependencies, and evaluates performance in terms of instruction throughput. (This analysis is very close to CQA's, though with the extra assumption that the Front-End can always deliver 4 uops per cycle.)

2. Latency analysis: IACA evaluates the number of cycles needed go executed the target block once.

In both cases, IACA can highlight the assembly instructions that are are part of the performance bottleneck.

2.2.2 Dynamic Analysis

Dynamic analysis uses runtime information to evaluate performance. It can provide large amounts of information not obtainable through purely static means, but typically requires the target program to be executed at least once.

Furthermore, it introduces problems of its own, such as measurement stability and precision.

2.2.2.1 Hardware Support

Hardware can implement features specifically intended for performance evaluation. They can allow dynamic analysis to be faster, easier, more precise and/or collect more relevant information.

The Time Stamp Counter [23] (TSC) provides a very precise way of measuring time at a very low cost. Indeed, it counts the number of spent cycles, and is readable in only around 30 cycles using the *RDTSC* instruction on Sandy Bridge, making it very cost-effective.

Performance Monitoring Counters [24] can also count events other than just cycles, such as the number of retired instructions, the number of cache lines read from RAM, the number of stalls at different stages of the execution pipeline or even power consumption. All this information can be obtained through other means (value tracing, simulation, hardware probes), but at a much higher complexity and cost.

Furthermore, different methods can be used to access these counters:

1. Sampling: counters are parameterized to keep track of certain events and raise an interrupt when a certain threshold is reached. Sampling software can then identify the last retired instruction and credit it for the overflown counter's events, reset said counter to 0 and resume the program's execution. It is a very cheap and non-intrusive way to collect performance counter data.

2. Tracing: sampling's precision is not perfect as it essentially works by "blaming" the instruction that caused the overflow for the entirety of the event sample without any guarantee that it is indeed responsible for a majority of them. Furthermore, it cannot dissociate events associated to the same code but in different execution contexts (e.g. different function parameters). Tracing addresses these concerns by inserting *start* and *stop* probes in strategic parts of the code (e.g. right before and after a loop), allowing to a) precisely count events caused by the loop, and only by the loop and b) separate events caused by the same code area but at different times. It however comes at a higher cost because probes need to be inserted in the first place, and is not a working solution for tiny pieces of code.

3. Multiplexing: multiplexing consists in using the same hardware counter to monitor more than one event. It is done by making the counter alternate between the watched events during the measurement phases. It allows to collect more counters simultaneously (as the number of monitoring units is limited), but can degrade the precision of results.

However, the overhead for accessing PMCs can be important and should be considered when using them [25].

2.2.2.2 Finding Hotspots

Finding an application's *hotspots* (i.e. parts of the program needing the most time to execute) is important for performance analysis and optimization as it allows tools and developers to focus on parts of the code that have an important impact on the overall execution time. Different techniques can be used to find them.

For instance, Intel compilers have an option to insert RDTSC probes in the assembly code, tracing the time spent in different loops and functions [26]. The

main drawback of this method is that it requires having access to the source code (and recompiling it for performance monitoring purposes).

Tools such as VTune [27] and Perf [28] can detect hotspots using sampling. The Perf module of MAQAO [29] can also do so in applications using OpenMP or OpenMPI frameworks. Though this technique is not perfect [30], it is very efficient and does not require any changes to be made to the target application.

2.2.2.3 Counter-based Analysis

Programs or frameworks such as Oprofile [31], PAPI [32] and Likwid [33] only perform PMC measurements, leaving the analysis to other tools.

VTune [27] is a special case and combines both measurement and analysis abilities. Among them, the Top-Down approach [34] evaluates the performance contributions of the Front-End, Back-End, retirement and memory accesses.

Levinthal [35] offers counter-based performance metrics to evaluate performance bottlenecks.

HPC oriented frameworks [36] can also detect and categorize performance issues in parallel applications, as well as suggest potential fixes such as in-lining functions or changing data structures.

2.2.2.4 Differential Analysis

Differential Analysis [37] consists in a) transforming a given target code to isolate some of its characteristics and b) comparing the execution times of the original and the modified codes to evaluate the impact of the targeted characteristics.

DECAN is a tool implementing this approach at the assembly loop level. It was developed within MAQAO [22], using the framework's binary patching features [38] to generate new binaries with modified loops.

Figure 2.8 show-cases its main transformations:

1. *LS* (Loads/Stores): only keep memory instructions, address calculations and control flow instructions (which are necessary for the loop to iterate normally). Running this variant allows to see the contribution of memory accesses to the original loop's execution time.

2. *FP* (Floating Point): only keep floating point instructions and the control flow. It isolates the impact of FP operations in the original loop.

DECAN can be used on sequential, OpenMP and MPI programs alike.

One of the drawbacks of this approach is overhead, as each of the the target loop's variants need to be executed.

2.2.3 Simulation

Simulation can bring information that is difficult or impossible to otherwise get, and is often used as a means of validating performance models when the modeled hardware does not exist, or is hard to access.

However, it is a complex process for which one of the trade-offs is nearly always execution time, though other factors (e.g. memory consumption) may also get in the picture.

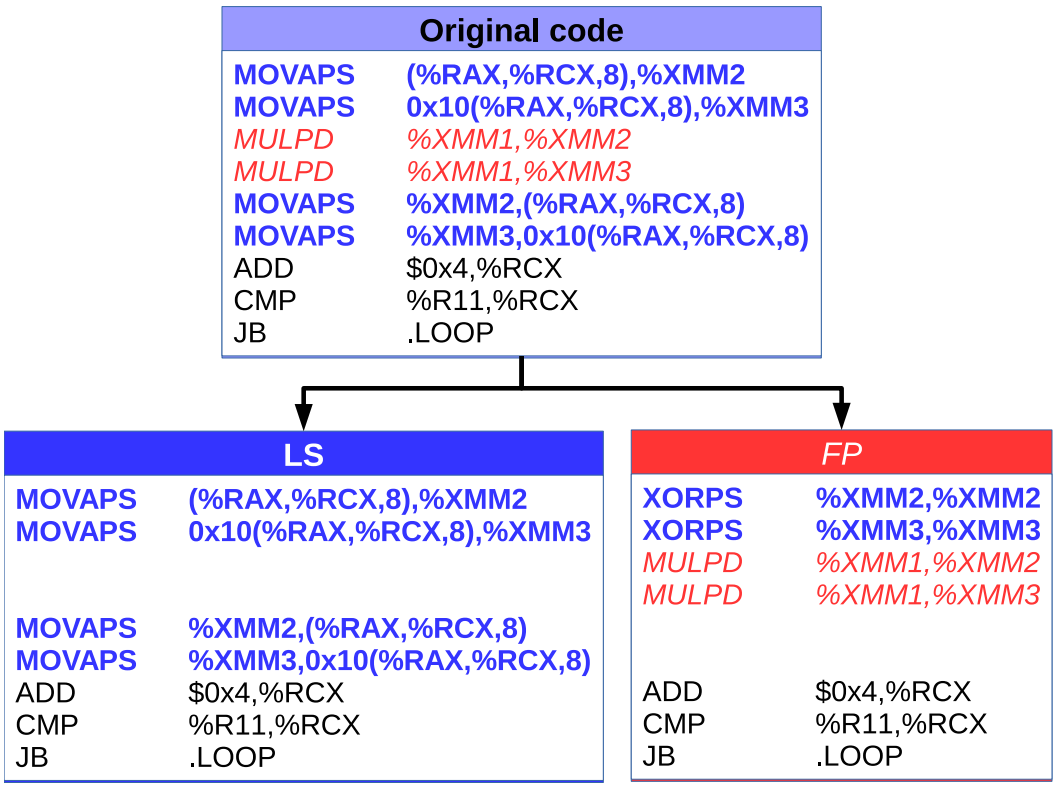


Figure 2.8: Example of DECAN Loop Transformations

This example assembly code multiplies all elements from an array by a constant.

DECAN can be used to patch loops to isolate some of their performance characteristics.

In the LS version, the MULPD *floating point instructions were removed.*

In the FP version, loads were transformed into zero-idiom XORPS *instructions to avoid creating new dependencies on the XMM registers. Stores were simply removed.*

2.2.3.1 Simulation Techniques

Different techniques can be used depending on the objectives of the simulation, or the targeted performance trade-off.

Execution-driven simulators [39, 40] simulate both the semantic of a program and its behavior on the modeled system. It allows them to tackle problems from scratch, but at high cost.

Trace-driven simulation offsets part of the problem by first collecting relevant execution details [41] (e.g. the list of all instructions executed for a program, as well as their outputs), and then using these traces to reconstitute the program's semantics. The simulation then only targets the behavior of the traced instructions, allowing it to be both less complex and faster than an execution-driven equivalent. However, trade-offs for using this technique include a) having to run the target program on a real machine (or an execution-driven simulator...) at least once to collect the traces, b) storing and using

trace files which can be extremely large and c) not being able to see the impact of e.g. randomness or different instruction sets without getting different traces.

Simulators can also have varying scopes. Full-system simulators [42, 43, 44] will simulate not just the CPU, but also other components such as graphical cards or

network cards so as to be able to run entire operating systems. However, their main objective is more likely to enable e.g. driver development for not-yet-existing hardware or system-level performance analysis, taking their focus further away from pure CPU performance analysis. PTLsim [45] adopts an interesting approach combining virtualization and simulation for performance analysis, allowing users to use virtualization's near-native execution speed when speed is important, and switching to simulation mode for performance analysis.

2.2.3.2 Cycle-Accurate Simulation

Detailed cycle-accurate simulation is the only way to get a perfect knowledge of a system's performance. It offers a full visibility on all relevant mechanisms without any impact on the results (unlike hardware or software measurement probes).

However, it comes with several drawbacks:

1. Execution time: simulators modeling everything perfectly will typically simulate at rates of a few thousand cycles per second [46], which represents a slowdown in the order of millions.

2. Amount of data: it can be hard to know which parts of the modeled system are actually relevant, and tell the cause from the consequence: simulation does not supersede analysis.

3. Purely simulation-based analysis can be very slow, as each tested potential bottleneck has to be tested separately and result in whole new simulation run.

Furthermore, cycle-accurate simulators modeling all the details of modern CPUs' hardware implementations are not publicly available, limiting their applicability for performance analysis for general application developers.

2.2.3.3 Improving Simulation Speeds

Different approaches have been developed to improve the speed of simulation:

1. Sampled simulation [47, 48]: simulating only parts of the execution in details allows to use faster techniques (e.g. emulation) for most of the time. The quality of the results will then depend on the sampling rate, and the quality (representativeness) of the samples.

2. Using dedicated simulation hardware [49, 50]: using FPGAs can greatly help with achieving better simulation speeds, as the hardware running the simulation is specifically fine-tuned for this task.

3. Interval simulation [51]: focuses on modeling the impact of *miss events* (e.g. cache miss, branch misprediction) and uses a simplified, fast model for the rest of the execution.

2.2.3.4 Functional Simulation

Specialized functional simulators can also provide interesting information. Such tools focus on reproducing realistic behaviors with no particular regards for timing.

For instance, tools like Cachegrind [52] track memory accesses and replay them in a realistic cache structure, evaluating how and whether programs use caches. Other tools [53] can also characterize multi-threaded workloads.

2.3 Using Codelets

Tackling important problems can often be simplified by decomposing them into smaller ones, which are more easily understood and fixed. The *codelet* approach applies this principle to modeling and optimization problematics, focusing on smaller problems to then approach the bigger ones.

2.3.1 Codelet Presentation

Codelet definitions may vary depending on their granularity and purposes. We will hence clarify our use of the term, as well as some of codelets' main advantages and drawbacks.

2.3.1.1 Definition

A *codelet* is a small piece of code that may be evaluated independently. Codelets can be coupled to *drivers*, intendedly minimal code complements allowing them to be run in a stand-alone manner.

In the context of this thesis, codelets will always be loop nests. However, coarser-grain codelets like functions could also be used.

2.3.1.2 Why use codelets?

Whether or not codelets are interesting depends on their intended uses:

1. For modeling, they are particularly interesting if they display some novel or/and unexpected behavior, as their being small helps pinpoint the issue.

2. For optimization, codelets can isolate significant performance problems, allowing potential solutions to be tested in a vacuum.

3. HPC scheduling: codelets could be used as a finer schedulable workload than threads [54], allowing to make better use of supercomputers' very high numbers of cores.

Codelets can offer important advantages in both cases:

1. Evaluating the impact of the environment: codelets' behavior variations depending on the execution environment (e.g. different host machines or CPU operating frequencies) can be determined in a cost-effective way due to their stand-alone nature and their normally small execution times.

2. Varying data sets: some codelets may allow for arbitrary input data sets to be used, making it possible to see how their behavior evolves with varying cache localities.

3. Optimization tests: codelets being stand-alone programs allows programmers to test different optimizations (or compiler optimization flags, compiler versions, etc.) and only impact the intended code. It would not be as simple with e.g. real applications, in which potential optimizations may have an impact on the whole program.

2.3.1.3 Drawbacks of Codelets

While codelets may be very convenient, their *stand-alone* property may raise problems such as:

1. Static interference: the compiler may alter or reorder statements to try to produce more efficient code (e.g. loop interchanging, in-lining...), or certain optimizations may be overly optimistic (e.g. optimizing data structures for individual loops whilst they are used for a whole program). Studying a piece of code out of its original (or a realistic) context may hence create unnatural circumstances.

2. Cold/warm cache [55]: the state of the caches when running a codelet may have a strong impact on its observed behavior. Several possibilities exist (completely cold caches, completely warm ones, or various degrees in-between). As there is no universal strategy to create a single most-appropriate execution environment, users have to make this potentially complex decision on a per-case basis.

3. Dynamic influence from and on other loops: in a real application, a loop may have a strong influence on how other loops behave and in turn be influenced by other loops' execution. For instance, a previous loop could warm up caches or on the contrary trash them, and instructions from different loops can also cohabit in the execution pipeline due to superscalar mechanisms. Combining the codes of two codelets together may hence produce different results than when running them separately.

These issues may ultimately cause modeling or optimization projection errors. They represent a limitation that codelet users should keep in mind.

2.3.2 Artificial Codelets

Developing artificial codelets allows for a fine control of experiments. However, they require codelet creators to already have a clear idea about what they want to test, and how.

2.3.2.1 Hand-Crafted Codelets

Some codelets can be handmade to test something specific. Furthermore, small benchmarks do fit our definition of codelet.

On the hardware side, [56, 57] use simple benchmarks to characterize the effective bandwidth on a machine. [57] also covers other potential bottlenecks such as memory latency or network bandwidth.

On the software side, [58, 59] use handmade (and legally vectorizable) loops to test auto-vectorizing compilers' abilities.

2.3.2.2 Automated Generation

Codelets can also be generated automatically, following patterns of (potential) interest. Microtools [60] facilitate this process to evaluate the impact of e.g. different memory access patterns, unrolling or vectorization. They can also measure the execution times of the resulting codelets.

Henri Wong [61] generates loops using pointer-chasing to expose the number of out-of-order resources in the ROB and the Physical Register File in Intel microarchitectures, as well as test the impact of zero-idiom and register move instructions on register consumption.

2.3.3 Extracted Codelets

Codelet *extraction* is the process of isolating a program's hotspot as a codelet.

Such codelets offer various advantages over synthetic ones:

1. They represent realistic workloads. For instance, on SNB, there is an important penalty for using both SSE and AVX instructions in the same loop. While a synthetic codelet will be able to exactly address this problem, mixing such instructions is not a mistake compilers would do in realistic circumstances, hence much lowering its relevance. Extracted codelets will expose real day-to-day problems and bottlenecks.
   Having realistic workloads allows models to be trained on codes directly relevant to their objectives (e.g. using HPC codelets to model HPC applications).

2. They can be used to study each of an application's hotspots separately, allowing said application's performance to be modeled in small bricks rather than as a whole [62].

2.3.3.1 Manual Extraction

Codelets can be extracted manually, e.g. by copying the source code of a loop. However, it is a slow and error-prone process which can only be reasonably used when the number of interesting codelet candidates is low.

However, it is a fairly simple process, and allows for implementation liberties e.g. to allow arbitrary data sets to be processed by the loop.

2.3.3.2 Automated Extraction

Automating the extraction process greatly reduces the costs of codelet extraction, as well as the hazards inherent to manual processing. It also makes it realistic to *cover* the majority of the execution time in HPC applications potentially comprising hundreds of hotspots.

Codelet extraction tools can operate at different levels. For instance, [63, 64, 65] extract hotspots at the source level, also saving the runtime state of the memory so that the resulting codelet can be replayed faithfully. A drawback of this method is that there may be discrepancies between the generated compiled codes for original and extracted versions.

On the other end of the spectrum, [66] identifies codelet-like structures called *simulation points*. They are pieces of code pinpointed at the assembly level for the purpose of speeding up simulation. This technique is extremely faithful in terms of assembly code and memory states, but limits the adaptability and portability of the identified workloads.

An in-between approach is adopted in [67], where hotspots are extracted at a compilation-time Intermediate Representation (IR) level. It offers advantages in terms of extraction complexity as the IR may be language-independent, but creates a dependency on the used compiler instead. Unlike source-level extraction, it does not

allow for codelets to be easily modified, but is more flexible than Simulation Points as it allows for different optimization flags and data sets to be tried. [68] adapts this approach to parallel codes, operating at a coarser granularity to quickly evaluate the scalability of OpenMP parallel regions.

Chapter 3

Codelet Performance Measurement Framework

The complexity of modern CPUs makes loop performance be the result of many factors simultaneously coming into play. Identifying these factors, their individual impact and how they interact with one another correspondingly becomes increasingly difficult.

In this chapter, we approach this issue by producing fine-grained data on various -but mostly simple- loops in a controlled in vitro environment. The aim is to produce reliable data allowing to single out certain components' behavior, check implementation hypotheses and validate low-level performance-evaluation tools.

3.1 Introduction

Modern CPUs being developed in a highly competitive and industrial environment, pushing companies into different strategies to protect their competitive advantage:

1. Patenting: while offering legal protection against plagia, patents have the downsides of a) only protecting *implementations*, b) requiring the publication of the material to protect and c) being temporary. Furthermore, as they can be costly to establish, not all innovative solutions may be deemed worth the matching cost.

2. Secrecy: confidentiality can protect *ideas* (outside the realm of patentable materials), but very little can be done against plagia if the material gets leaked or stolen. This forces companies to be conservative in terms of releasing technical implementation details.

In this environment, performance researchers have to deal not just with the complexity brought by state-of-the art CPUs comprising billions of transistors, but also with the unknowns induced by hardware characteristics not disclosed by CPU manufacturers.

We specifically target Intel microarchitectures, for which the manuals provided by the manufacturer still provide a sizeable amount of information on microarchitectures' pipeline and hence allow for a reliable overview of the core pipelines. Further to this, information from patents can provide interesting leads on implemented hardware mechanisms, though patents can exist without the mechanisms they describe being actually implemented in real-world products.

Empirical data can be used to:

1. Independently validate information from manuals.

2. Test implementation hypotheses (and empirically expose undocumented characteristics).

3. Validate performance models and find unexpected, unmodeled or/and undocumented phenomena.

This chapter will focus on the production of such reliable, low-level data, from which other chapters of this thesis will draw information quite extensively.

We will present the target codes, the followed methodology and the experimental parameters we played with to capture the codes' performance in different contexts. We will also present PCR, the data repository we developed to store and exploit the generated data.

3.2 Target Loops: Numerical Recipes Codelets

The choice of the loops to study is important as it determines not just the breadth of captured phenomena but also their relevance. For instance, picking codes that are too similar would expose only a few of the hardware components' behavior. Furthermore, if said components only have an impact on performance in extremely corner cases, one can question the importance of modeling them in details.

We hence picked loops from the Numerical Recipes [69], which present the advantages of being relatively diverse while also tackling real and significant numerical challenges.

We will describe how we obtained these loops and why, as well as present the main implementations of the codelets we selected.

3.2.1 Obtention and Target Properties

Our NR codelets collection is the result of both automatic loop extraction (using [64]) and manual cherry-picking. The desired properties were as follows:

1. Single innermost loop: there should only be a single innermost loop in the compiled version of the codelet. This allows us to attribute all measured performance characteristics to a narrowly defined piece of code, and helps simplify performance analysis.

2. No innerloop branching: the only branch instruction should be the one allowing the loop to iterate (in compilation terms, the loop comprises a single *basic block*). Branches can add a lot of complexity to performance analysis, especially if they depend on input data values.

3. Data-value invariant iteration test: it may (and should) vary depending on the dataset size, but should not depend on a dataset's *values*. This makes the code more DECAN tolerant, allowing it to suffer more semantic loss when getting patched without causing unwanted side effects (e.g. infinite loop). While some workarounds can be implemented around such issues, we want to keep the simplicity we aim with these codelets.

4. Work on arbitrary dataset sizes (allowing to study the codelet's interaction with caches and the main memory).

5. Perform FP operations. Mere memory transfers, for instance, would not be as interesting to analyze with DECAN's classical variants. Furthermore, while integer computation can be present in hotspots, FP performance (i.e. FLOPS)

is often the baseline to evaluate a system's (or an application's) performance in scientific computing and we hence focused on it.

On top of this, we wrote several versions of the same extracted loops, altering characteristics such as the FP precision, the unroll factor, the instruction set to use or whether or not to vectorize.

3.2.2 Presentation and Categories

We succinctly present our main versions of the NR codelets in Table 3.1.

Column *Codelet* gives the name by which we commonly refer to these codelets, and which we will use throughout this thesis manuscript. The names are composed as follows:

1. The first part of the name (e.g. *balanc* for *balanc_3_de*) refers to the function from which the codelet was extracted.
2. The following number (*3* here) is an ID given by Astex [64] during the extraction process.
3. Finally, the suffix (*se* in our example) presents our re-implementation details: the first letter presents the FP precision used in computations ('*d*' for double precision, '*s*' for single precision and '*m*' for mixed precision code), and the second letter represents the used instruction set ('*e*' for *SSE 4.2* and '*x*' for *AVX*). Although they are not the versions we use by default, we also generated AVX versions of all the codelets presented here.

Column *NR Context* explains the purpose of the code from which the codelet was extracted.

We also show the type of loop nest (column *Loop Structure*) in the codelet:

1. *1D* codelets do not have "active" nested loops (if they exist, they may be executed once but are not iterated).
2. *2D* have "active" nested loops and always operate on square matrices.
3. *2DT* are similar to 2D loops, except they follow triangular access patterns (and may consequently have varying numbers of innermost iterations for a given data set), making data accesses and branches harder to predict.

As loop performance being often dictated by memory, and memory performance by data access patterns within the loop, we decided to also show codelets' memory access strides (i.e. the distance between accessed elements) in column *Access Pattern* to help appreciate codelets' memory access efficiency:

1. Stride *1* loops make good use of the cache structure by fully using loaded cache lines.
2. *LDA* (Leading Dimension Access) codelets' innermost loop accesses data following a 2D array's leading dimension, causing them to access elements not contiguous in memory. The effective stride depends on the data set.
3. *CLDA* (Constant LDA) codelets' stride is not 1, but is constant. This allows for memory performance to be more consistent across data sizes, and happens due to 1D loops exploring rectangular matrices column-wise (in Fortran).

Table 3.1: NR Codelet Suite

Codelet	NR Context	Loop Structure	Access Pattern	Description
balanc_3_de	Matrix balancer	1D	1	Array scaling
elmhes_10_de	Hessenberg form reduction	1D	1	Daxpy-like
elmhes_11_de		1D	CLDA	Daxpy-like
four1_2_me	FFT	1D	1	Complex operations on an array
hqr_12_se	Eigenvalues finder	2DT	1	Reduction with absolute value
hqr_13_de		1D	1	Reduction with absolute value
hqr_15_se		1D	LDA	Subtracting constant from matrix diagonal
jacobi_5_se	Jacobi method to find eigenvalues and eigenvectors	2DT	1	Reduction with absolute value
lop_13_de	Nonlinear elliptic equation solver	2D	LDA	5-point stencil computation
ludcmp_4_se	Lower upper matrix decomposition	2DT	1	Complex reduction
matadd_16_de	Nonlinear elliptic equation solver	2D	1	Adding 2 matrices into a 3rd
mprove_8_me	Linear equation solution improvement	2D	1	Reduction of matrix and vector
mprove_9_de		1D	1	Array subtraction
realft2_4_de	Inverse FT	1D	1	Complex operations on an array
realft_4_de		1D	1	Realft2_4_de w/ truncated dependencies
relax2_26_de	Gauss-Seidel matrix relaxation	2D	LDA	5-point stencil computation
rstrct_29_de	Half-weighting restriction	2D	LDA	5-point stencil computation
svbksb_3_se	Singular value backsubstitution	2D	1	Reduction of matrix and vector
svdcmp_6_de	Matrix singular value decomposition	1D	CLDA	Reduction with absolute value
svdcmp_11_de		1D	CLDA	Array scaling
svdcmp_13_de		1D	1	Dividing array elements + reduction
svdcmp_14_de		1D	1	Dividing array elements, storing results in a separate array
toeplz_1_de	Toeplitz matrix solver	1D	1	Reduction on 3 arrays
toeplz_2_de		1D	1	Daxpy-like
toeplz_4_de		1D	1	Daxpy-like
tridag_1_de	Tridiagonal matrix solver	1D	1	Complex operations on 3 arrays
tridag_2_de		1D	1	Daxpy-like

List of the 26 in vitro codelets extracted from the Numerical Recipes [69, 70, 71], and realft_4_de *(which we crafted by truncating realft2_4_de).*

On an important note, this characterization was applied *after* first experiments were run, allowing us to account for modifications applied by the compiler. For instance, the compiler interchanges loops in *matadd_16_de*, causing it to be a stride 1 codelet even though accesses follow the leading dimension in the source code.

3.3 Measurement Methodology

A rigorous measurement methodology is particularly important for our intended purpose, as the worth of our measured data is entirely dictated by how reliable it is.

3.3.1 Placing Probes

Fine-grain computer measurements raise several general problems:

1. Observability: watching the experiment may change the result. Software probes are convenient to measure the CPU's activity, but the CPU processing them incurs a measurement overhead affecting the result value. This can be mitigated with event sampling, but at the cost of precision. Hardware debuggers may also help in this regard as a large part of the observation cost is externalized, though even they may impact the observed behavior.

2. Reproducibility: while many factors can be controlled, some cannot, and it is impossible to guarantee the same intended experiment will consistently produce the same output. This is especially true with parallel programs (whose scheduling is partially random), and memory states (e.g. virtual memory mapping).

3. In pipelined machines, and even more so in out-of-order ones, it is hard to exactly pinpoint when the loop to measure starts, and when it ends. Is it when the first instruction of the loop gets decoded, executed, or retired? What do we do about instructions foreign to the loop cohabiting in the pipeline?

It is hence particularly important to use probes adequately. Here, we will describe our probe setup for time on the one hand, and general event counters on the other.

3.3.1.1 Measuring Time

We use the *RDTSC* [23, 72] probe provided by DECAN [37] to measure time. As it is particularly inexpensive, we can place it precisely before and after the target binary loop (see Figure 3.1) while still retaining an acceptable overhead.

We define the "cycles per iteration" metric derived from RDTSC as:

$$CycPI\_R = \frac{elapsed\_time\ (RDTSC)}{nb\_assembly\_level\_iterations}$$

Note: when talking about time, we always use *reference cycles*, whose length is independent from the active core frequency. On Intel microprocessors, they usually tick at the same rate as the maximum non-turbo frequency available on the CPU.

3.3.1.2 Measuring Hardware Events

The probe for hardware counters (or *HWC*) is considerably more expensive as they require to leave the Linux userland to get initialized and read. This is particularly true when measuring memory controller related events, as different intermediary steps need to be taken (both for client CPUs [73] and for server versions [74]). We hence had to place it at a considerably coarser level to keep the probe overhead low, and measure the whole program in lieu of just the target loop (see Figure 3.1).

This is only acceptable under the following conditions, which we ensure are met during measurements (see Section 3.3.2.1):

1. Initialization costs are negligible.
2. The target binary loop is the only significant active loop in the codelet at runtime.

Certain counters can also be used to report time in reference cycles (for instance, *CPU_CLK_UNHALTED.REF* in Sandy Bridge): we define the "cycles per iteration" derived from them as:

$$CycPI\_H \;=\; \frac{elapsed\_time\ (HWC)}{nb\_assembly\_level\_iterations}$$

Tools like Likwid [33] are used for such HWC measurements.

3.3.1.3 Combined Time Metric (ECPI)

We also define an "enhanced" time metric combining information from both the RDTSC and the HWC time values, and which we will use as our default time metric throughout this manuscript:

$$ECPI \;=\; min\ (CycPI\_R,\ CycPI\_H)$$

This metric makes us get the best from both worlds: if the neighboring loops are important, CycPI_R is closer to the time actually taken by the loop than its HWC counterpart. On the other hand, for very small loops where the overhead from the RDTSC probe might not be negligible, CycPI_H may be more accurate.

3.3.2 Measurement Quality and Stability

Measurement quality is a more abstract property, aiming to make sure measurements are representative of what an experiment is meant to measure. Measurement stability is important to ensure results are a) reproducible and b) not showing a statistically aberrant behavior.

In this section, we will explain how we pursue these two orthogonal objectives.

3.3.2.1 Measurement Quality

Controlling noise is a good way to increase the quality of results:

1. The proportion of iterations spent in the target loop (in all innermost loop iterations inside the codelet) is checked to make sure only one loop is significant. This metric needs to be close to 1 to reasonably attribute all hardware counter events to the target loop (due the probe placement described in Section 3.3.1.2).

2. All measurements are performed on dedicated machines, limiting the noise created by other processes in the system. Noise could otherwise be created due to certain resources being shared on the system, such as RAM bandwidth, cache space or execution time slices.

3. A repetition loop ensures that the caches are properly heated [55, 75] for all but the first codelet call. It also makes sure that the process startup, array initialization costs and the first codelet call -likely to be an outlier- contribute only insignificantly to the overall result in both time and counter measurements. The number of repetitions to perform is calibrated to make the program run for at least 1 second and be no less than 100.

4. The program is *pinned* to an execution core, forbidding the system to migrate it to different ones. Migrations can impact performance due to e.g. some of the caches are core-specific, causing some data locality to be lost. The first core (for each present CPUs) is avoided, as sometimes more solicited by the OS than others.

5. All CPU cores are set to a constant frequency, avoiding the variance induced by on-demand or turbo frequency scaling. It also makes it easier to interpret results.

3.3.2.2 Measurement Stability

Measurement stability is obtained by the use of *meta-repetitions*: the whole experiment is run several times, allowing us to get different sets of measurements. The median value is then calculated independently for each metric, and used as the reference result. A "D" is suffixed on the name of metrics computed this way to clarify their origin (e.g. CycPI_R_D.).

The number of meta-repetitions favoured in literature is 31 [76, 77, 78], though we did not notice significant differences when using only 11 -or even 5- in our in-vitro experiments. This is partly due to the quality controls explained earlier, as well as to the absence of parallel code in our target codelets. Hence, some of the results presented may have been produced with numbers of meta-repetitions ranging from 5 to 31. The practical consideration in keeping this number low is the total experimental time needed: a tradeoff with must be met.

3.3.2.3 Codelet Program Structure

Figure 3.1 presents the overall structure of our codelet processes (as well as where we place our different software probes). The codelet is isolated in its own function and compiled separately, preventing unwanted compiler optimizations (e.g. finding out the codelet's results are never used and removing all codelet calls from the binary). A *driver* program handles the initialization of the codelet's arguments as well as the repetition loop described earlier.

It is important to note that the repetition loop does little in terms of bettering hardware counter measurements when the instructions represented as "..." in the figure represent a sizeable part of the function's execution time, as they too get repeated. It can happen for very small data sets (in which peel and tail loops can process a significant part of the workload) or for cases of loop splitting (where the compiler decides to distribute the source loop's statements over several assembly

```
HW Counters (start) // Only when measuring hardware counters
'driver' process
{
        read arguments (data size, nb repetitions)
        allocate and initialize codelet arrays

        repetition loop
        {
                codelet function (arrays, data size, ...)
                {
                        ...
                        RDTSC (start) // Only when measuring time
                        target binary loop
                        {
                                instructions of interest
                        }
                        RDTSC (stop) // Only when measuring time
                        ...
                }
        }

        free allocated memory
}
HW Counters (stop) // Only when measuring hardware counters
```

Figure 3.1: Codelet Structure and Probe Placement

loops), regardless of whether there is only one innermost loop at the source level (as is the case for all the NR codelets we selected).

3.3.3 CQA Reports

Detailed CQA [21] reports are generated for the studied loop, providing valuable insights on its theoretical peak performance and a good performance decomposition for L1 data sets.

Such reports can also help confirm the measurements are sense-making, i.e. aberrant values are not produced.

3.4 Varying Experimental Parameters

Codelets can be studied from many different angles: we apply various DECAN [37] transformations to them, make them run for diverse data sizes, on different machines (see Table 3.2), core frequencies or memory loads.

We will present the experimental variations supported by the framework and show result examples in this section.

Table 3.2: Machine List

Machine Alias	CPU						Note
	SKU	Uarch	Nb of Cores	LLC	Frequencies		
					Min	Max	
HSW	E3-1270 v3	HSW	4	8 MB	0.8 GHz	3.5 GHz	
IVB	I7-3770	IVB	4	8 MB	1.6 GHz	3.4 GHz	
SLM	C2750	SLM	4	4 MB	1.2 GHz	2.4 GHz	
SNB1	E5-4640	SNB	8	20 MB	1.2 GHz	2.4 GHz	Transparent Huge Pages enabled
SNB2	E5-2640	SNB	6	15 MB	1.2 GHz	2.5 GHz	Poor RAM configuration
SNB3	E5-2640	SNB	6	15 MB	1.2 GHz	2.5 GHz	
SNB4	E3-1240	SNB	4	8 MB	1.6 GHz	3.3 GHz	

This table summarizes some of the essential characteristics of the machines we used to run our experiments, and which will be referred to throughout this manuscript.

All machines used a 64-bit version of Linux, with a kernel version greater or equal to 2.6.

Hyper-threading was disabled on all machines (except for Silvermont ones as they do not support it in the first place).

The poor RAM configuration on SNB2 *is due to not all RAM slots being populated on the motherboard, and the ones that are not taking advantage of the CPU and the motherboard's multi-channel capabilities.*

3.4.1 DECAN Variants

DECAN is a tool allowing to patch binary loops so as to extract some of their performance characteristics.

The main transformations (or *variants*) we use are:

1. FP: removes memory accesses from the loop, exposing the time taken by floating point operations.

2. LS: removes floating point operations, exposing the time needed to perform memory accesses.

3. REF: serves as reference point for the original loop's behavior.

An example of performance decomposition using DECAN (and this framework) is presented in Figure 3.2. This kind of information is valuable for optimization, as it allows developers to know what aspects they should prioritize. For instance, here, it is clear that any significant performance gains can only be achieved by optimizing memory accesses, and optimizing FP operations should not even be considered.

3.4.2 Data Sizes

An interesting property of the codelets we selected is they can be easily run for arbitrary zero-filled data sets as their control flow does not depend on the data values. This allows us to observe how their behavior evolves depending on where the data is held in the memory hierarchy.

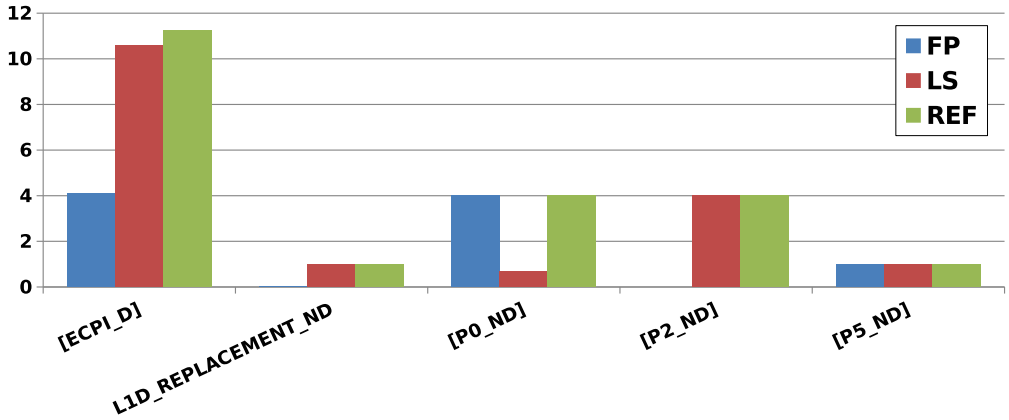


Figure 3.2: DECAN Performance Decomposition Example (balanc_3_de)
The codelet was run on SNB1 (see Table 3.2), for a data set fitting in L3. All metrics are normalized per iteration. [ECPI_D] *represents the measured execution time,* L1D_REPLACEMENT_ND *the measured number of cache lines coming into L1 (using hardware counters), and* P0_ND, P1_ND *and* P5_ND *represent the numbers of uops respectively sent to ports 0, 1 and 5 (according to CQA).*

We can see that the patched FP loop preserves the port load on P0 and P5 (ports mostly responsible for FP operations and control flow) while discarding memory activity (cache traffic is null, and memory port P2 has no activity).

We can also see that the patched LS loop preserves the memory traffic and P2 activity while significantly decreasing the number of operations on P0. The activity on P5 is kept and is due to preserving the control flow.

Running the different patched loops makes it easy to see that memory operations are responsible for most of the execution time here.

Each codelet works on arrays whose size are determined by an input N variable, which we vary to change data set sizes. However, as codelets are free to do as they wish with this value (e.g. 1D codelets can allocate N elements per array, whilst 2D codelets allocate N^2 of them), may work with elements in single or/and double precision, and can operate on different numbers of arrays anyway, a given value of N does not necessarily correspond to the same data set size across codelets.

An example of the advantage obtained by varying data sizes is given in Figure 3.3. Here, the codelet shows a different behavior depending on where the data is held in the memory hierarchy. It can give interesting insights as to how the codelet can be optimized depending on the target workload: if the codelet is typically called for small data sets, then FP operations should be looked into; otherwise trying to optimize memory accesses should take priority.

3.4.3 Machines and Microarchitectures

Performance may vary depending on the microarchitecture, the CPU itself or the machine's configuration. It can then be interesting to see how codelets' behavior changes depending on the machine.

An example is given in Figure 3.4. Here, the optimization impact is more hardware than software related. Developers can use these experiments to know which microarchitecture best fits their needs.

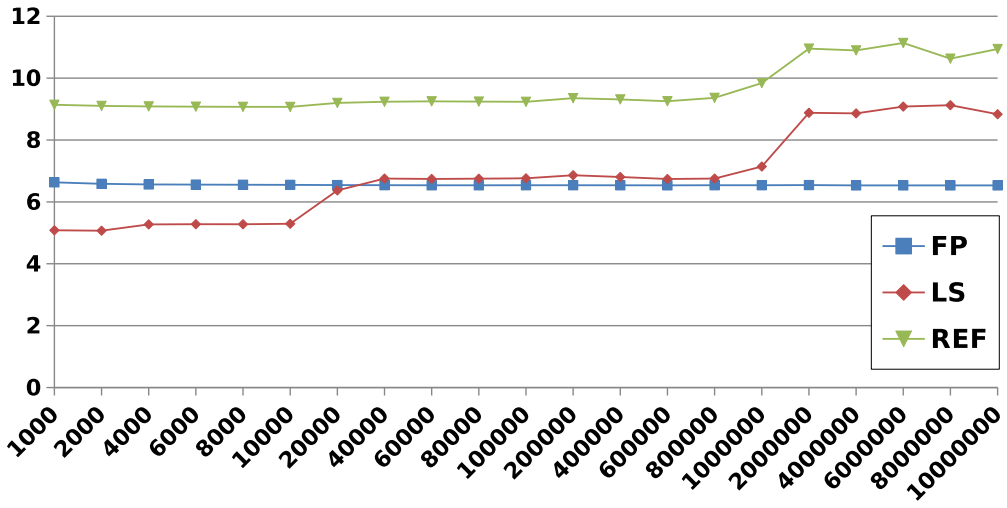


Figure 3.3: Example of Codelet Behavior Across Dataset Sizes (toeplz_4_de)

The codelet was run on SNB1 (see Table 3.2). The y-axis represents cycles per iteration (ECPI_D), while the x-axis shows the input values for N *(which controls the size of the dataset the codelet works on). The input values were chosen arbitrarily with the objective of sweeping through the different behaviors codelets can exhibit depending on data cache locality.*

We can see the relative contribution of LS to REF's performance level increases as the size of the data set grows, pushing the data further and further in the memory hierarchy. Thanks to LS, we can see the data likely fits in L1 for data points 1000 and 2000, then in L2 until point 10 000 (included), then in L3 until point 800 000, transitioning to RAM on point 1 000 000, and fully in RAM on later points.

The performance for FP is constant across data sets, which makes sense considering this DECAN variant does not read or write data from/to the memory hierarchy, and is hence unimpacted by cache limitations.

We can consequently see FP operations are the bottleneck in L1 and L2, but have a lesser impact in L3 and RAM.

3.4.4 Frequencies

Energy consumption is one of the three classical considerations for performance (with time and memory space), and can be optimized by adjusting the operating frequency of the CPU cores [79, 80]. Indeed, power consumption can be approximated by the following formula [81, 82], thanks to which we can see it scales with frequency:

$$Power = Capacitance * Voltage^2 * frequency$$

Studying how well codelets scale with frequency changes can help find cases in which frequency scaling is profitable, hence the framework allows for measurements to be done at different operating frequencies using the CPUFreq [83] system.

Figure 3.5 shows an example of such frequency scaling analysis using our framework. In this case, there may be room for frequency-driven energy savings for RAM data sets.

Note: all experiments are performed in the highest base frequency available (i.e. the "reference" frequency in the studied CPUs) unless otherwise specified.

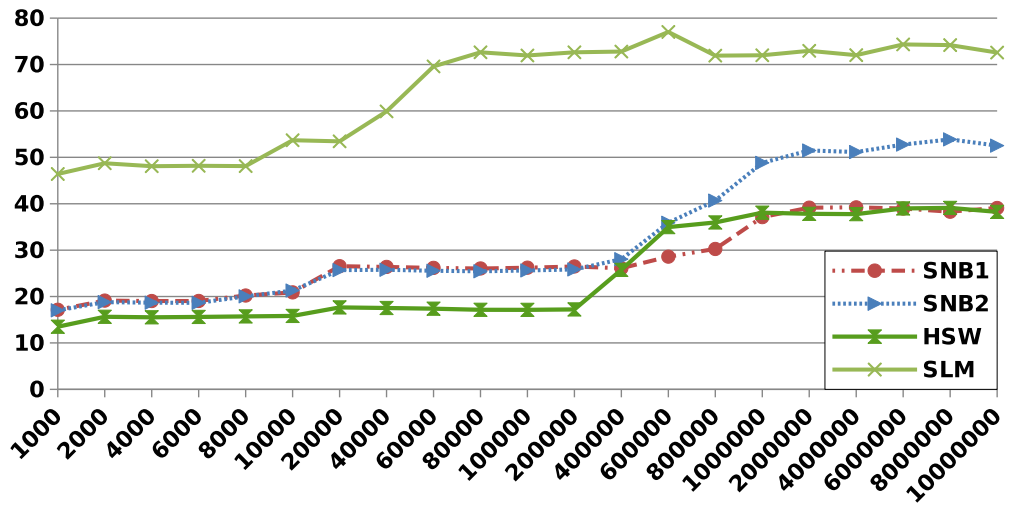


Figure 3.4: Behavior across Machines (toeplz_1_de)

The y-axis represents cycles per iteration (ECPI_D). The x-axis shows the input values for the N *variable, which governs the size of the dataset.*

Detailed descriptions for the machines are available in Table 3.2. As a quick summary, SNB1 is a machine with a Sandy Bridge CPU with 20 MB of L3 cache and operating at 2.4 GHz. SNB2 is a machine with a Sandy Bridge CPU with 15 MB of L3 cache, operating at 2.5 GHz, and with poorly populated RAM slots (reducing the peak RAM bandwidth). HSW is a machine with a Haswell CPU with 8 MB of L3 cache and operating at 3.5 GHz. SLM is a machine with a Silvermont CPU with 4 MB of L2 cache and operating at 2.4 GHz. Note: only 2 MB are accessible by a given Silvermont core, so the effective L2 cache is 2 MB here.

We can see that performance per clock varies depending on the microarchitecture, with Silvermont behind far behind Sandy Bridge and Haswell even for RAM data sets. The effect of cache sizes is also particularly visible, with Silvermont's 2 MB not holding the data anymore as early as point 60 000.

We can also see Haswell brings a noticeable improvement over Sandy Bridge when the data still fits in L3 (points below 200 000), though performance for SNB1 is shortly better for points 600 000 and 800 000 (likely due to its L3 being significantly bigger). Also, one should keep in mind HSW is running at 3.5 GHz (against respectively 2.4 and 2.5 GHz for SNB1 and SNB2), contributing to make RAM look slow on this graph.

Finally, we can see SNB1 works better than SNB2 when RAM gets in the picture, which is to be expected considering it is configured better in this regard.

3.4.5 Memory Load (using Memload)

Our in vitro codelets are run on a single core, with the rest of the machine being made to be as little active as possible to get stable and clean measurements. However, it also puts them in an ideal scenario where they can access all shared resources with no competition. We developed *Memload*, a small tool to allow us to retain the ability to observe codelets' behavior when other programs make use of the (shared) RAM bandwidth, providing a more realistic execution environment.

Memload is a multi-threaded program that streams through arrays for no other purpose than to consume bandwidth from the memory hierarchy. Its threads get pinned on the cores not used by the codelet so as to only impact CPU-wide resources,

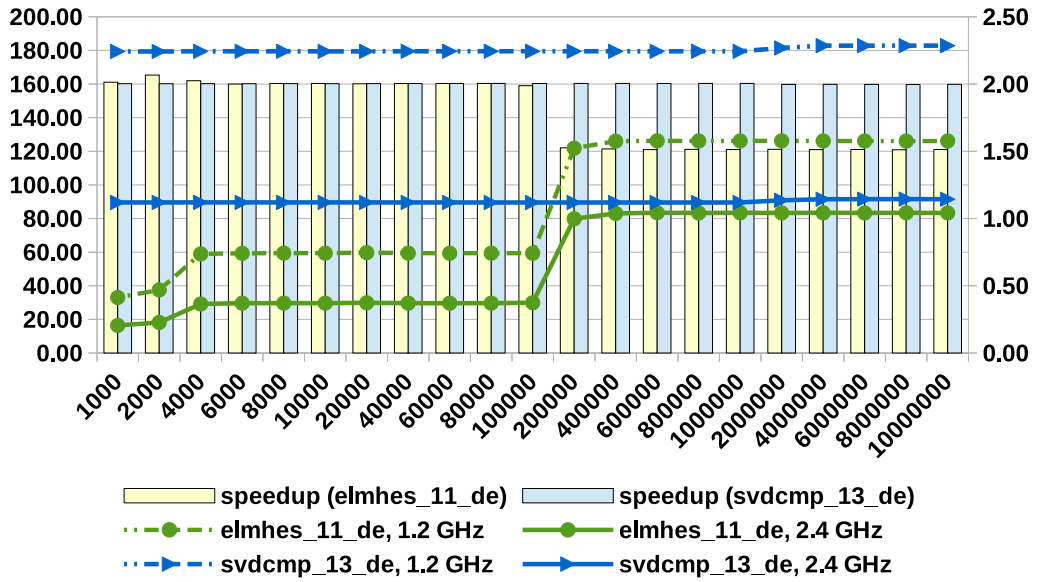


Figure 3.5: Frequency Scaling Example (elmhes_11_de and svdcmp_13_de)

The codelets were run on a Sandy Bridge CPU. The y-axis represents cycles per iteration (ECPI_D). The speedup values are graded on the secondary y-axis, and represent the speedup achieved when switching from the lower frequency to the higher one. The x-axis shows the input values for the N *variable, which governs the size of the dataset.*

Here, we can see svdcmp_13_de scales perfectly with the frequency change across all data sets: running it in high frequency pays off. The speedup is also important for elmhes_11_de, but in a less considerably manner when the data comes from RAM: it falls to 1.5x (from 2) on point 200 000. There may be room for energy consumption savings here.

and their bandwidth consumption is frequently monitored and regulated.

Inputs can be used to control Memload's activity:

1. Size of the array to stream through: this allows the user to target a given element of the memory hierarchy. For instance, if the array fits in L2, then Memload will not impact the effective bandwidths for L3 and RAM.

2. Target bandwidth consumption: the user specifies the bandwidth Memload should try to consume (in MB per second). The tool may fail to reach this number (if the target is unrealistic or the bandwidth reached a saturation point), but will throttle itself if its exceeds it.

3. Access mode: regular loads can be used to impact both the effective bandwidths and available cache spaces, or non-temporal stores (a.k.a. *streaming stores*) to impact the bandwidth without consuming cache space. In the latter case, RAM bandwidth will be impacted regardless of the size of the array specified in 1).

We set the Memload tool to consume as much RAM bandwidth as possible using non-temporal stores: this allows to see the codelet's behavior in the worst case scenario while keeping regular and *memloaded* results directly comparable for

a given data size (i.e. the codelet's data does not get shifted to a higher level due to cache space being occupied by Memload).

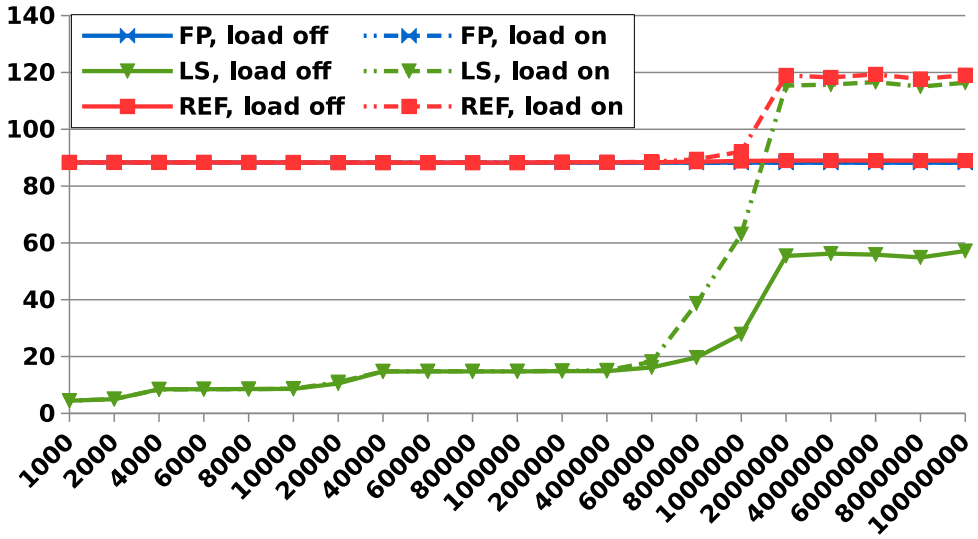


Figure 3.6: Memory Load Example (svdcmp_14_de)

The codelets were run on a Sandy Bridge CPU. The y-axis represents cycles per iteration (ECPI_D). The x-axis shows the input values for the N *variable, which governs the size of the dataset.*

Memload has no impact on the FP variant as it makes no use of the (data) cache hierarchy. LS starts getting impacted on point 800 000 (when not all data fits in L3 anymore), reaching a peak ECPI_D on point 1 000 000. REF only gets impacted on point 1 000 000 onwards.

Svdcmp_14_de is strongly dominated by FP operations (due to divisions being very slow) in all cases when its execution core has free access to the CPU's RAM bandwidth. However, a strong memory activity makes it memory bound that when most of its working data sets do not fit in L3 anymore (starting from point 1 000 000), which we would not be able to see with purely single-threaded experiments.

We can see an example in Figure 3.6. Such data can help developers know what their code's bottleneck really is when parallel workloads coexist on the CPU, and hence guide their optimization effort.

3.4.6 Overall Structure

The framework handles varying the data size, running frequency and Memload parameters. However, it is up to the user to run the framework on different host machines. Other parameters not described here (e.g. which compiler or compiler version to use, or whether CPU features such as hyper-threading and prefetchers should be active) are also left to the user.

The framework's general structure is summarized in Figure 3.7.

3.5 Results Repository: PCR

Our codelet performance measurement framework allows us to produce massive amounts of data:

```
compile codelet
identify target loop with DECAN
{
  run the program and collect iteration counts
  select loop with the highest iterations count
}
generate DECAN variants (LS, FP, etc.) for the target loop
{
  CycPI\_R version (with RDTSC probe)
  HWC version (without RDTSC probe)
}
static analysis for patched loops
{
  extract assembly code
  CQA report generation
}
// Changing experimental parameters
for all target data sizes
{
  change input file
  run the program and collect iteration counts
  ensure the target loop is still the most important one
  for all target frequencies
  {
    change core frequency
    for all target memory loads
    {
      configure Memload (if needed)
      for all DECAN variants
      {
        for all meta-repetitions
        {
          run pinned RDTSC version
          get HWC measurements
          {
            start HWC collection
            run pinned HWC version
            stop HWC collection
          }
        }
        collect median measurements
      }
    }
  }
}
```

Figure 3.7: Framework Structure

- There are 27 codelets in our NR collection.
- The number of collected metrics per individual experiment is around 30 (for the measured hardware counters and CycPI_R).
- The codelets are run on several machines (roughly 8 different ones for the NRs).
- The number of DECAN variants usually exceeds the 3 classical ones, and tends to be more around 5 instead.
- Each codelet is run for 21 different data sizes.
- We typically run codelets on two different frequencies (the lowest and the highest available).
- Different memory load configurations can be tried, though this is only rarely done. We can assume the Memload parameter is not significant for the purpose of giving a rough estimate.

The number of data points for the NRs' DECAN entries alone (disregarding CQA-generated values) then conservatively reaches $27*30*8*5*21*2 \simeq 1.3\ million$. This is also ignoring re-runs (due to tool updates) and runs with atypical experimental variations (e.g. more DECAN variants, more collected metrics...).

We hence developed *Perfcloud Results Repository* (PCR), a database tool to keep track of the large volumes of data produced by our framework. Indeed, a major challenge when having so much data is being able to navigate it. PCR allows users to retrieve data and easily draw comparisons, as we did throughout Section 3.4.

3.5.1 Features

PCR's main features include:

1. Remote access through a web interface.
2. Manual exploration of the results in case users are interested in a very specific piece of data.
3. Fetching important amounts of data through a combination of filters (e.g. "get *all data* about codelet *balanc_3_de* from the *Numerical Recipes* which was obtained on machine *SNB1*"). Usable filters include application names, codelet names, machine names or CPU types, data sizes, operating frequency, DECAN variants, etc.
4. User-specified layouts of data: while the data is always presented in tables, users can choose how to structure them. With the example above, the user could decide to have data for different DECAN variants show up in different tables, data for different data sizes in different rows, and data for different frequencies in different columns.
5. On-the-fly generation of "aggregate" metrics computed using stored data. For instance, users can specify they want an extra *ECPI_D* metric to be computed as the minimum of the *CycPI_R* and *CycPI_H* values.
6. Exportation of fetched data to CSV or XLS files.

7. Data importation using a simple specialized input format, enabling other frameworks to produce compatible input files.

It can also be used by several people at the same time, allowing other researchers to store and share experimental results of their own. Some of the in vivo data presented later in the manuscript was made available to us this way.

Most of the data used in this manuscript went through PCR to facilitate their storage and formatting. At the time of writing, our repository indexed over 17.5 million measured values, over more than 4300 codelets (and loops) and 22 machines.

3.5.2 Technical Details

PCR can be accessed through a web Front-End, allowing it to be used remotely with any web browser. It was entirely developed in PHP, except for a Java-written module used to generate XLS files.

It is backed with a SQL database, which handles most of the complexity behind indexing and retrieving data and ensures the integrity of the repository.

3.5.3 Acknowledgments

We would like to thank Clément Malet and Roméo Azangue for contributing to the development of PCR during their stay at the laboratory.

3.6 Related Work

The Microtools [60] also focus on studying codelets at a low level, also using using techniques such as pinning and meta-repetitions to get accurate and stable results. However, they mostly focus on purely synthetic workloads, and do not use static analysis. Hardware counters are also not currently supported.

Vtune [27] integrates many different functionalities similar to those used in our framework, but focuses entirely on program optimization. Our framework focuses on a single codelet, analyzing it from many different angles.

The Codelet Tuning Infrastructure [84] (CTI) project measures loops from whole applications, using both static analysis and dynamic measurements. However, as with VTune, they are not studying individual loops as extensively as our framework allows and focus instead on providing an overview at the application level. They also offer to share experimental results in a multi-user environment, furtherly providing all necessary files to re-run experiments.

3.7 Future Work

A stability metric can be computed for each measurement, using the following formula:

$$stability = (median - min)/min$$

Adding this feature to the framework would allow for a very fine-grained evaluation of how stable the results are.

The framework in general could be improved to support parallel codelets, better tackling the evergrowing need in this domain. Supporting loops containing conditional code (other than for iterating) would also open more codelets to further scrutiny.

Some work is still needed to determine how codelets with several loops could best be supported with the described approach. A possible lead is to extend DECAN's patching capacity to single out a single loop and disable others, allowing each relevant loop to be studied in separate runs.

Better measurement speeds could also be achieved by using less strict methods than "start and stop" for counter measurements, such as sampling. Furthermore, inserting probes around the repetition loop to better target the region to measure would allow us to reduce the number of repetitions needed to get quality results.

Finally, while the framework was tested on "big core" Intel microarchitectures (Sandy Bridge, Ivy Bridge, Haswell; both on desktop and server versions) and on Silvermont (Avoton), work on both the tools it uses (CQA, DECAN, etc.) and itself will be needed to support later microarchitectures, or completely different ones such as ARM. Mobile computing with smartphones and tablets represents a large chunk of today's computing use, and studying the microarchitectures lying underneath could bring interesting insights on possible optimization opportunities.

3.8 Conclusion

We have presented the measurement methodology behind our Codelet Performance Measurement Framework, combining static and dynamic analysis tools such as CQA, DECAN and Likwid to produce quality low-level measurements and insights.

We have also presented our NR codelet collection, which we tested extensively through our measurement framework and will make repeated appearances in the rest of this manuscript.

We will use this data to help create PAMDA (Performance Assessment Methodology using DECAN) in Chapter 4, to extend the Cape modeling tool and refine CQA and DECAN variants (see Chapter 5), and as validation data for our static Uop Flow Simulation loop modeling (see Chapter 7).

Chapter 4

PAMDA: Performance Assessment Methodology Using Differential Analysis

Identifying performance bottlenecks in applications is crucial to improve their efficiency, but it may be difficult to precisely assess their impact on performance: in particular, two performance problems can interact making it difficult to isolate and therefore to correct them. We propose PAMDA [85], a methodology to single out performance problems through hierarchical bottlenecks detection. Important potential performance issues are classified in a 'Performance Breakdown Tree' which is used to drive our iterative analysis cycle, prioritizing the most relevant problems. Our system relies on MAQAO toolset and code's differential analysis. While MAQAO is a performance analysis and optimization tool suite, the differential analysis approach, which is implemented through DECAN tool, consists in quantifying performance changes when applying controlled transformations to the target code. Our focus will be on performance issues raised by processors and memory subsystems in multicore architectures. We will demonstrate the approach on loops extracted from real life HPC applications.

4.1 Introduction

The recent progress of high performance architectures generates new challenges for performance evaluation tools: more complex processors (larger vectors, many cores), more complex memory systems (multiple memory levels including NUMA, multiple-level prefetch mechanisms), more complex systems (large increase in core counts up to several hundred of thousands now) are all key issues which need to be simultaneously optimized to get a decent performance level.

To work properly, all of these mechanisms require specific properties from the target code. For example, good exploitation of memory hierarchies relies on good spatial and temporal locality within the target code. The lack of such properties induces variable performance penalties: such combinations (mismatch between hardware and software) are denoted performance pathologies. Most of them have been identified (cf. Table 4.1) and efficient workarounds are well known. The current generation of performance tools (TAU [86], PerfExpert [87], VTune [27], ThreadSpotter [88], Scalasca [89], Vampir [90]) is excellent at detecting such pathologies although some are fairly specialized: for instance, Scalasca/Vampir mainly addresses MPI/OpenMP issues, requiring the combined use of several tools to get a global overview of all of the performance pathologies present in an application.

Most of the current tools do not provide any direct insight on the potential cost of a pathology. Furthermore, the user has no idea about what the potential benefit of optimizing his code to fix a given pathology is. These two points prevent him from

focusing on the right issue. For example, let us consider a program containing two hot routines A and B, respectively consuming 40 % and 20 % of the total execution time. Let us further assume that the potential achievable performance gain on A is 10 % while on B it is up to 60 %. The overall performance impact on B is up to 60 % * 20 % = 12 % while on routine A, it is at best 10 % * 40 % = 4 %. As a consequence, it is preferable to focus on routine B. Additionally, the user has no clue of what the current performance level is, compared with the best one achievable, i.e. he may not know when optimizing is worth the investment.

In general, the situation is even worse since a simple loop may simultaneously exhibit several performance pathologies. In such cases, most of the tools cited above give no hint to the user of which ones are dominant and really worth fixing. For instance, a loop can suffer from both a high miss rate and the presence of costly Floating-Point (FP) operations such as div/sqrt: trying to improve the hit rate does not improve the performance if the dominant bottleneck is the div/sqrt operations.

In this paper, we present a coherent set of tools (MicroTools [60], CQA [91, 92, 22, 21], DECAN [37], MTL [93]) to address this lack of user's guidance in the tedious and difficult task of program optimization. These tools are integrated in a unified methodology (PAMDA) to help the user to quickly identify performance pathologies and to assess their cost and impact on the global performance. The different techniques (static analysis, value profiling, dynamic analysis) appear to be more appropriate and give a more accurate answer depending upon performance pathologies to be fixed: for example, detecting a badly strided access is immediate through value tracing of array addresses while the same task is extremely tedious when only using static analysis or hardware counters. Anyway, such array access tracing should only be triggered when necessary due to its high cost. In this paper, we focus on providing performance insight at the core level and parallel OpenMP structures. Our analysis can be combined with MPI analysis provided by tools such as Scalasca, TAU or Vampir.

Through the integrated methodology PAMDA, we aim at providing the following contributions:

- Getting a global hierarchical view of performance pathologies/bottlenecks
- Getting an estimate of the impact of a given performance pathology taking into account all other present pathologies
- Demonstrating that different specialized tools can be used for pathology detection and analysis
- Performing an hierarchical exploration of bottlenecks according to their cost: the more precise but expensive tools are only used on specific well chosen cases.

Section 2 presents a motivating example in detail. Section 3 details the various key components of PAMDA while Section 4 describes the combined use of these different tools. Section 5 describes some experimental use of PAMDA. Section 6 gives an overview of related works and the added value of the PAMDA system. Finally, Section 7 gives conclusions and future directions for improvement.

Table 4.1: A few typical performance pathologies

Pathologies	Issues	Workarounds
ADD/MUL balance	ADD/MUL parallel execution (of fused multiply add unit) underused	Loop fusion, code rewriting e.g. use distributivity
Non pipelined execution units	Presence of non pipelined instructions: div, sqrt	Loop hoisting, rewriting code to use other instructions eg. x86: div and sqrt
Vectorization	Unvectorized loop	Use another compiler, check option driving vectorization, use pragmas to help compiler, manual source rewriting
Complex control flow graph in innermost loops	Prevents vectorization	Loop hoisting or code specialization
Unaligned memory access	Presence of vector-unaligned load/store instructions	Data padding, use pragma and/or attributes to force the compiler
Bad spatial locality and/or non stride 1	Loss of bandwidth and cache space	Rearrange data structures or loop interchange
Bad temporal locality	Loss of perf. due to avoidable capacity misses	Loop blocking or data restructuring
4K aliasing	Unneeded serialization of memory accesses	Adding offset during allocation, data padding
Associativity conflict	Loss of performance due to avoidable conflict misses	Loop distribution, rearrange data structures
False sharing	Loss of bandwidth due to coherence traffic and higher latency access	Data padding or rearrange data structures
Cache leaking	Loss of bandwidth and cache space due to poor physical-virtual mapping	Use bigger pages, blocking
Load unbalance	Loss of parallel perf. due to waiting nodes	Balance work among threads or remove unnecessary lock
Bad affinity	Loss of parallel perf. due to conflict for shared resources	Use numactl to pin threads on physical CPUs
High number of memory streams	Too many streams for hardware prefetcher or conflict miss issues	See conflict misses
Lack of loop unrolling	Significant loop overhead, lack of instruction-level parallelism	Try different unrolling factors, unroll and jam for loops nest, try classical affinities (compact, scatter, etc.)

4.2 Motivating Example

Figure 4.1 presents the source code of one of the hottest loops extracted from POLARIS(MD) [94]: a molecular dynamics application developed at CEA DSV. POLARIS(MD) is a multiscale code based on Newton equations: it has been successfully used to model Factor Xa involved in thrombosis.

This loop simultaneously presents a few interesting potential pathologies:

- Variable loop trip count
- Fairly complex loop body which might lead to inefficient code generation by the compiler
- Presence of div/sqrt operations
- Strided and indirect access to arrays (scatter/gather type)
- Multiple simultaneous reduction operations leading to inter iteration dependencies

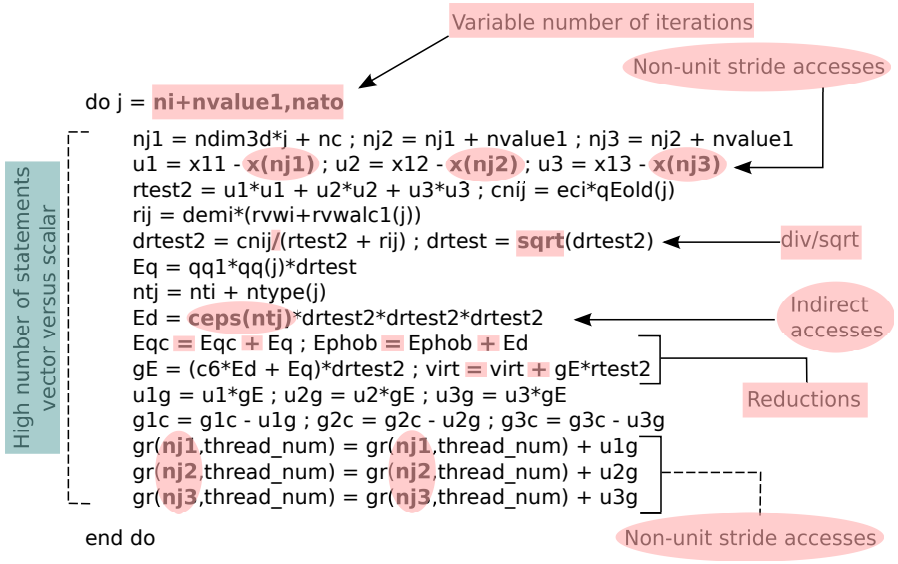


Figure 4.1: Polaris Source Code Sample
The code's main performance pathologies are highlighted in pink.

All these pathologies can be directly identified by simple analysis of the source code. The major difficulty is to assess the cost of each of them and therefore to decide which should be worked on.

A first value profiling of the loop iteration count reveals that the trip count is widely varying between 1 and 2000. However the amount of time spent in the small (less than 150 iterations) loop trip count instances remains limited to less than 10 %. The remaining interval of loop trip counts is further divided into 10 deciles and one representative instance is selected for each of them. Further timings on analyzing loop trip count impact indicate that the average cost per iteration globally remains constant independently from the trip count. Therefore, the data size variation seems to have no impact on performance: the same optimization techniques should apply for instances having a loop trip count between 150 and 2000.

The static analyzer (see Figure 4.3) provides us with the following key information: in the original version, neither Load/Store (LS) operations nor FP ones are vectorized. It further indicates that due to the presence of div/sqrt operations, the FP operations are the main bottlenecks. It also points out that even if the FP operations were vectorized, the bottlenecks due to sqrt/div operations would remain. However this information has to be taken with caution since the static analyzer assumes that all data accesses are ideal, i.e. performed from L1.

Dynamic analysis using code variants generated by DECAN is presented in Figure 4.2. Initially, the original code (in dark blue bars) shows that FP operations (see FP versus LS DECAN variants) clearly are the dominating bottlenecks. Furthermore, the good match between CQA and REF clearly indicates that analysis made by CQA is valid and pertinent. Optimizing this loop is simply obtained by inserting the SIMD pragma '`!DEC$ VECTOR ALWAYS`', which forces the compiler to vectorize FP operations. However, the compiler does not vectorize loads and stores due to the presence of strides and indirect access. Rerunning DECAN variants of this optimized version (see light blue bars in Figure 4.2) shows that even for this optimized version FP operations still remain the key bottleneck (comparison between LS and FP). Therefore, there is no point in optimizing data access, the only hope of

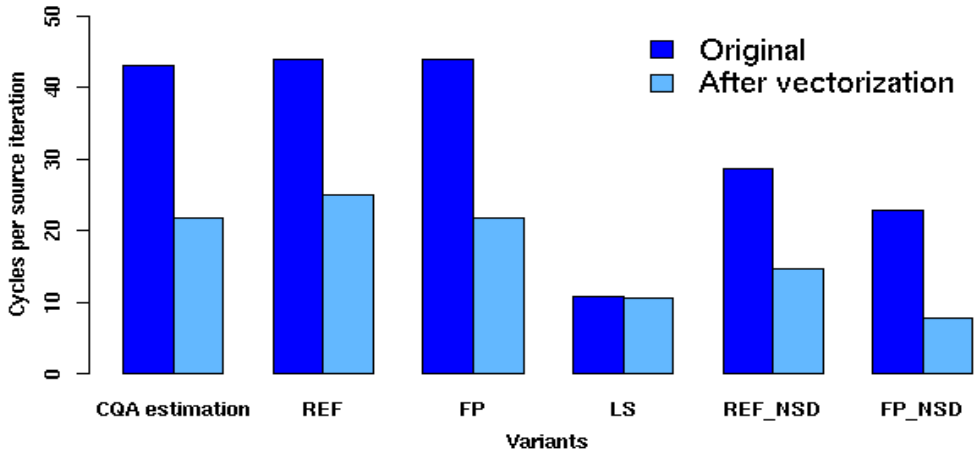


Figure 4.2: DECAN Analysis Example

The y-axis represents the number of cycles per source iteration: lower is better.

Comparing static estimates obtained by CQA with dynamic measurements performed on different code variants generated by DECAN of both of the original and vectorized versions: **REF** *is the reference binary loop (no binary modifications introduced by DECAN),* **FP** *(resp.* **LS***) is the DECAN binary loop variant in which all of the Load/Store (resp. FP) instructions have been suppressed,* **REF_NSD** *(resp.* **FP_NSD***) is the DECAN binary loop variant in which only FP sqrt and div instructions (resp. all of the Load/Store and FP sqrt/div instructions) have been suppressed.*

optimization lies in improving div/sqrt operations: for example SP instead of DP. Unfortunately, such a change would alter the numerical stability of the code and cannot be used.

The major lesson to be drawn from this case study is that a combined use of CQA and DECAN allows us to quickly identify the optimization to be performed and also gives us a clear halt on tackling other pathologies without impacting overall performance.

4.3 Ingredients: Main Tool Set Components

Performance assessment issues require robust methodologies and tools. Therefore, in order to systematically provide programmers with a performance pathology hierarchy and its related costs, the current work considers two toolsets: MicroTools, for microbenchmarking the architecture, and the MAQAO [91, 92, 22] framework, which is a performance analysis and optimization tool suite.

MAQAO's goal is to analyze binary codes and to provide application developers with reports to optimize their code. The tool mixes both static (code quality evaluation) and dynamic (profiling, characterization) analyses based on the ability to reconstruct low level (basic blocks, instructions, etc.) and high level structures such as functions and loops. Another MAQAO key feature is its extensibility. Users easily write plugins thanks to an embedded scripting language (Lua), which allows fast prototyping of new MAQAO-tools.

From MAQAO, PAMDA extensively uses three tools including the Code Quality Analyzer tool (CQA) exposed in section 4.3.2, the Differential Analysis framework

(DECAN) presented in section 4.3.3, and finally the Memory Tracing Library (MTL) in section 4.3.4. We briefly present the main contribution of each of these tools to PAMDA and then describe their major characteristics.

4.3.1 MicroTools: Microbenchmarking the Architecture

Microbenchmarking [95, 96, 97] is an essential tool to investigate the real potential of a given architecture: more precisely, in PAMDA, microbenchmarking is first used to determine both FP units performance and achievable peak bandwidth of various hardware components such as cache/RAM levels, and second to estimate the potential cost of various pathologies (unaligned access, 4K aliasing, high miss rate, etc.).

For achieving these goals, PAMDA relies on `MicroTools` , consisting of two main components: `MicroCreator` tool automatically generates a set of benchmark programs, while `MicroLauncher` framework executes them in a stable and closed environment.

4.3.2 CQA: Code Quality Analyzer

In PAMDA, the CQA framework is used first for providing a performance target under ideal data access conditions (all operands are supposed to be in L1), second for providing a bottleneck hierarchy analysis between the various hardware components of the core (FP units, load/store ports, etc.) and third for detecting some performance pathologies (presence of inter iterations dependencies, div/sqrt operations) which are worth investigating via specialized DECAN variants. The ideal assumption (all operands in L1) is essential for CPU bound codes such as the POLARIS(MD) loop studied in the previous section. For memory bound loops, it needs to be complemented with a dynamic analysis.

CQA is a static analysis tool directly dealing with binary code. It extracts key characteristics, and detects potential inefficiencies. It provides users with general code metrics such as details on basic loop characteristics, the number of instructions, μ`ops`, and used XMM/YMM vector registers. CQA also allows users to obtain more in-depth information on the loop execution on the target architecture. For example, the tool provides a reliable Front-End pipeline execution report, which is an estimated number of cycles spent during each Front-End pipeline stage. The tool gives the same type of report for the Back-End. Finally, CQA provides a cycle estimate of loop body performance under ideal conditions: all operands in L1, no branches and infinite loop count (steady state behavior).

CQA is able to report both low and high level metrics/reports (figure 4.3). For example, when a loop is not fully vectorized, the high level report provides a potential speedup (if all instructions were vectorized) and corresponding hints (compiler flags and source transformations). For the same loop, some low level metrics/reports show the breakdown of vectorization ratios per instruction type (loads, stores, ADDs, etc.) giving the user a more in-depth view of the issue.

CQA supports Intel 64 micro-architectures from Core 2 to Haswell.

4.3.3 DECAN: Differential Analysis

In PAMDA, DECAN is used for quantitatively assessing performance pathologies impact. The general idea is fairly simple: a given pathology is associated with the

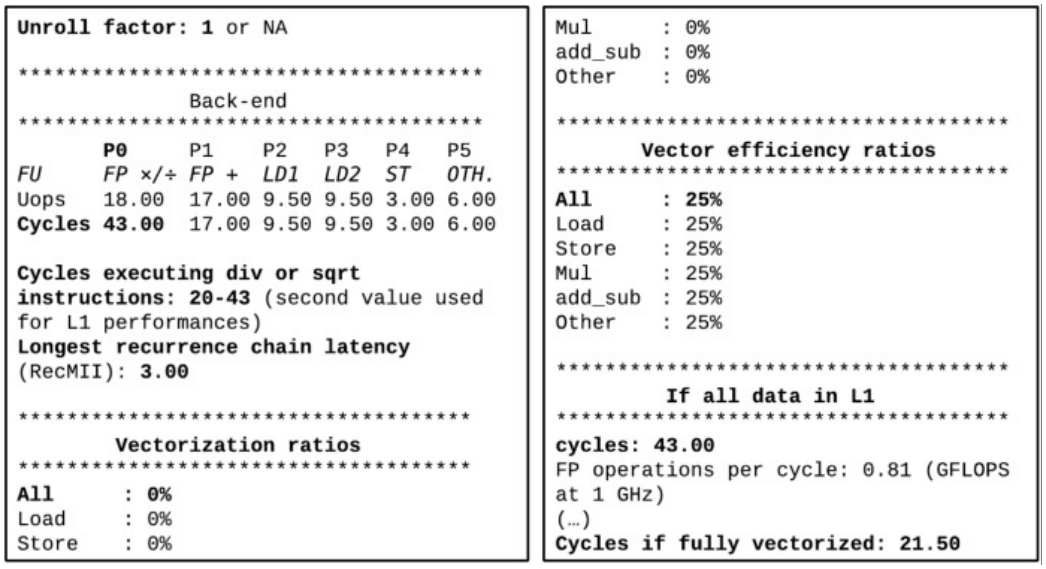


Figure 4.3: Low-Level CQA Output

presence of a given subset of instructions, for example div/sqrt operations, then DECAN generate a binary version of the loop in which the corresponding instructions are deleted or properly modified. This altered binary is measured and compared with the original unmodified version.

The resulting binary does not in general preserve semantics, i.e. numerical values generated with DECAN variants are not identical to the original ones. For our performance analysis objective, this is not a critical issue but for the subsequent program execution, control behavior might be altered. To avoid such problems, the original loop is systematically replayed after the execution of the modified binary in order to restore correct memory values.

DECAN starts by using static analysis on the target loop produced by CQA. The goal is to select instruction subsets to be transformed, as the selection process is driven by the desired type of behavior to highlight. Afterward, instructions are carefully transformed in a manner that minimizes unwanted side effects that may disturb the observations, such as changes in the code layout and instruction dependencies. It also inserts some monitoring probes to be able to accurately compare the modified part of the code with the original one. Also, and as stated earlier, DECAN is built on top of the MAQAO framework, hence, it uses the MAQAO disassembler/patcher to forward modifications on the instructions.

Using DECAN's features, PAMDA generates altered binaries, thereby splitting performance problems between CPU, memory, and OpenMP issues. Table 4.2 presents a range of loop variants used within the methodology discussed in Section 4.4.

4.3.4 MTL: Memory Tracing Library

Within PAMDA, MTL provides specific analysis of pathologies related to data access patterns in particular stride values, alignment characteristics, data sharing issues in multi threaded codes, etc. MTL works by tracing addresses and by generating compact representations of data access patterns. MTL is not limited to innermost

Table 4.2: DECAN variants and transformations

Variant	Type of SSE/AVX Instructions Involved	Transformation
LS	All arithmetic instructions	Instruction deleted
FP	All memory instructions	Instruction deleted
DL1	All memory instructions	Instruction operands modified to target a unique address
NODIV	All division instructions	Instruction operands modified to target a unique addresses
NORED	All reduction instructions	Instruction deleted
S2L	All store instructions	Converted into load instructions
NO_STORE	All store instructions	Instructions are deleted

loops but directly deals with multiple nested loops, allowing to detect more subtle pathologies: for example, row major instead of column major accesses for a Fortran array (stored column wise) are automatically detected. To perform these analyses, MTL uses the MAQAO Instrumentation Language (MIL) [98]. This language makes the development of program analysis tools based on static binary instrumentation easier. In fact, MIL is a specific language for object-oriented and event-directed domains to perform binary instrumentation at a high level of abstraction using structural objects (functions, loops, etc.), events, filters, and probes.

4.4 Recipe: PAMDA Tool Chain

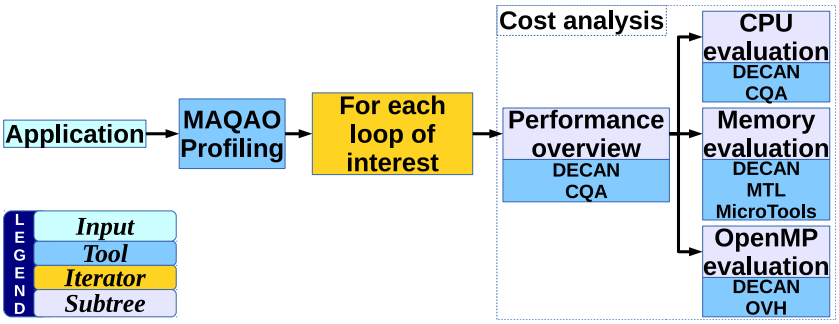


Figure 4.4: PAMDA Overview

Individual tools are the building blocks that PAMDA glue together through a set of scripts. These scripts are under development but most of the principles have been already evaluated. Figure 4.4 presents PAMDA's overall organization, which includes application profiling, cost analysis, structural checks, CPU and memory subsystems evaluation, and finally OpenMP evaluation for parallel applications. The current section describes PAMDA's components.

4.4.1 Hotspot identification

To limit the processing cost, we focus on the most time consuming portions of the code. Our target loops are defined as the loops with a cumulated execution time exceeding 80% of the total execution time. It should be noted that with such an

aggregated measurement, we can end up with a large number of loops with small individual contributions. Such target loops are identified using `MAQAO sampling`.

4.4.2 Performance overview

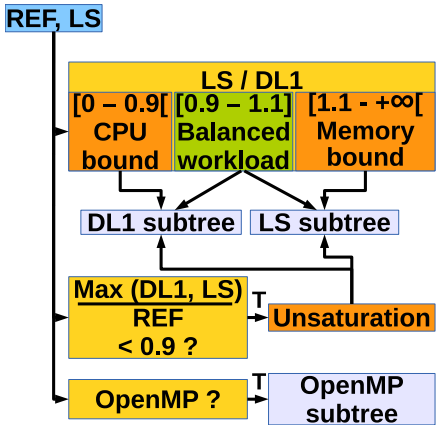


Figure 4.5: Performance Investigation Overview
T *means the condition is true and* F *that it is False.*

The PAMDA approach divides performance bottlenecks into two main categories (Figure 4.5): memory subsystem and CPU. Then, their respective contribution to the overall execution time is quantified using DECAN transformations `LS` (assessing memory subsystem performance) and `DL1` (assessing CPU subsystem performance). The ratio of these contributions reveals whether the loop is memory or/and CPU bound. Ideally, pipeline and out of order mechanisms insure that cycles spent for memory accesses and for arithmetic operations perfectly overlap: as a result, the time taken by `REF` should be the maximum time taken either by `LS` or `DL1`. In such a case, only the slower component needs optimizing. If the time taken by `LS` and `DL1` is similar, the workload is said to be balanced: optimizing both components is necessary to improve the loop's performance. Finally, when cycles taken by the memory and CPU components are poorly covered by one another (`unsaturation`), optimizing either of them can be sufficient to gain overall performance.

4.4.3 Loop structure check

Loop structure issues can be detrimental to performance, and may be detected using DECAN's loop trip counting feature. Indeed, in the case of unrolling or vectorization, peel and tail scalar codes may have to be generated to cover for remaining iterations. If too much time is spent in these peel and tail codes, this might indicate the unroll factor is too high with respect to the source loop iteration count. To detect such cases, loop trip counts for each version (peel/tail/main) are determined, and we check whether the main loop is processing at least 90 % of the source code iterations.

In some cases, the number of iterations per loop instance may not be large enough to fully benefit from unrolling or vectorization. This is easily highlighted by comparing the dynamic execution time of the DL1 DECAN variant with the CQA estimate, as the latter assume an infinite trip count. The difficulty to optimize such loops is exacerbated when the loop trip counts are not constant.

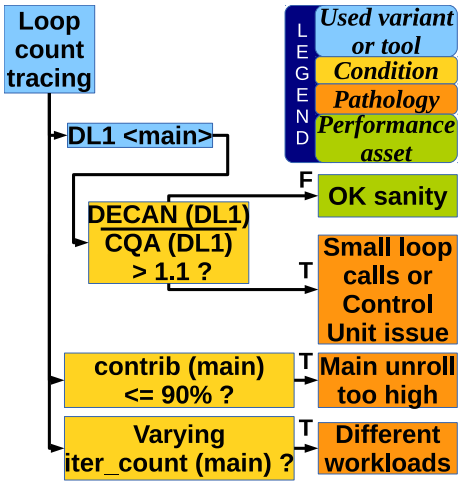


Figure 4.6: Detecting Structural Issues

4.4.4 CPU evaluation

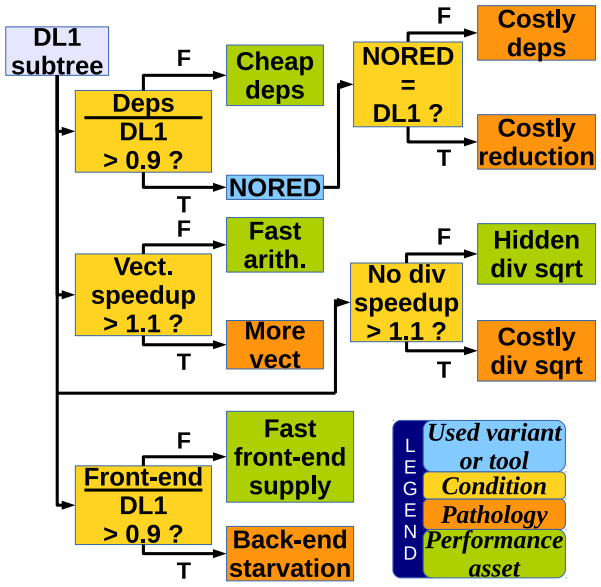


Figure 4.7: DL1 Subtree: CPU Performance Evaluation

Besides data accesses, CPU performance may be limited by other pathologies such as long dependency chains (`deps`), reductions (`RED`), scalar instructions or long latency floating point operations (`div`): these pathologies can be detected through the combined use of CQA and DECAN (Figure 4.7). The Front-End can also slow down the execution by failing to provide the Back-End with micro-operations at a sufficient rate. Comparing their contribution to L1 performance (`DL1`) is a cost-effective way to identify such problems. Finally, CQA can provide us with estimations of the effect of vectorizing a loop. We precisely quantify CPU related issues, enabling us to reliably assess potential for optimizations such as getting rid of divisions, suppressing dependencies or vectorizing. This information can guide the user's optimization decisions.

4.4.5 Bandwidth measurement

Data access rates from different cache levels / RAM highly depend on several factors, such as the instructions used or the access pattern. To account for it, we generate microkernels loading data in an ideal stream case, testing different configurations for load operations, with or without various software prefetch instructions, and/or splitting the accessed data in streams accessed in parallel. We also force misaligned addressing for `vmovups` and `movups`. Finally, we use `Microlaunch` to run these benchmarks for each level of the memory hierarchy.

Table 4.3: Bytes per Cycle for Each Memory Level (Sandy Bridge E5-2680)

Instruction	L1	L2	L3	RAM
vmovaps	31.74	15.05	10.81	5.10
vmovups	31.73	14.96	10.81	5.10
movaps	30.72	18.16	10.80	5.14
movups	29.53	17.07	10.79	5.23
movsd	15.67	11.55	10.61	5.36
movss	7.91	6.65	6.39	4.97

On our target architecture, 128-bit SSE load instructions could roughly achieve the same bandwidth as 256-bit AVX (Table 4.3) throughout the whole memory hierarchy. Except for `movss`, all instructions could attain similar bandwidths in L3 and RAM: only the type of instruction really matters for data accesses from L1 or L2, and data alignment is not as relevant as it once was.

4.4.6 Memory evaluation

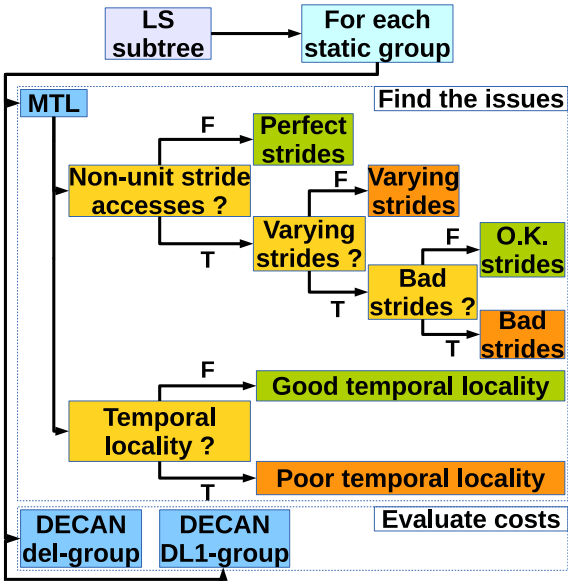


Figure 4.8: LS Subtree: Memory Performance Evaluation

Memory performance can be quite complex to evaluate. In Figure 4.8, we use MTL to find the different access patterns and strides for each memory group (as defined by the grouping analysis [37]). Memory accesses typically are more efficient when targeting contiguous bytes, while discontiguous accesses reduce the spatial

locality of data. The worst case scenario is having large and unpredictable strides, as hardware prefetchers may not be able to function properly. MTL also provides the data reuse distance, allowing the temporal locality evaluation of groups.

Once potential performance caveats are identified, we can use DECAN transformation `del-group` to single out offending groups and quantify their contribution to the `LS` variant global time. Comparing the bandwidth measured for each group with the bandwidth obtained in ideal conditions in the bandwidth measurement phase may then provide us with an upper limit on achievable performance.

4.4.7 OpenMP evaluation

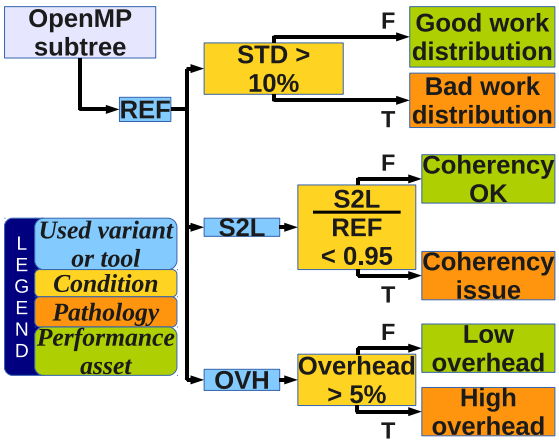


Figure 4.9: OpenMP Performance Subtree

STD *represents the standard deviation between threads, while* OVH *stands for our OpenMP OverHead evaluation.*

Some issues are specific to parallel programs using OpenMP (Figure 4.9). The standard deviation (`STD`) of the execution time for each thread points out workload imbalances. It is particularly important that no thread takes significantly longer than others to compute its working set, as loop barriers may then highly penalizing stalls. Another issue is excessive cache coherency traffic generated by store operations on shared data. Transformation `S2L` converts all stores to loads: we can quantify coherency penalties by comparing `S2L` with `REF`. Furthermore, the OpenMP Overhead (OVH) module of MAQAO is able to measure the portion of time spent in OpenMP routines, providing an OpenMP overhead metric.

4.5 Experimental results

We applied our methodology on two scientific applications: PNBench and RTM. The analysis processes and test results are presented below.

4.5.1 PNBench

PNBench is an OpenMP/MPI kernel used at CEA (French Department of Energy). Hot loops are memory bound and are ideal to stress tools dedicated to memory optimizations.

All tests are performed on a two-socket Sandy-Bridge machine, composed of two Intel E5-2680 processors with 8 physical cores each.

The profiling done on the initial MPI version of the code presents four loops consuming more than 8% of the global execution time each. Because of a lack of space, we only study the first one, but the three other loops have a similar behavior.

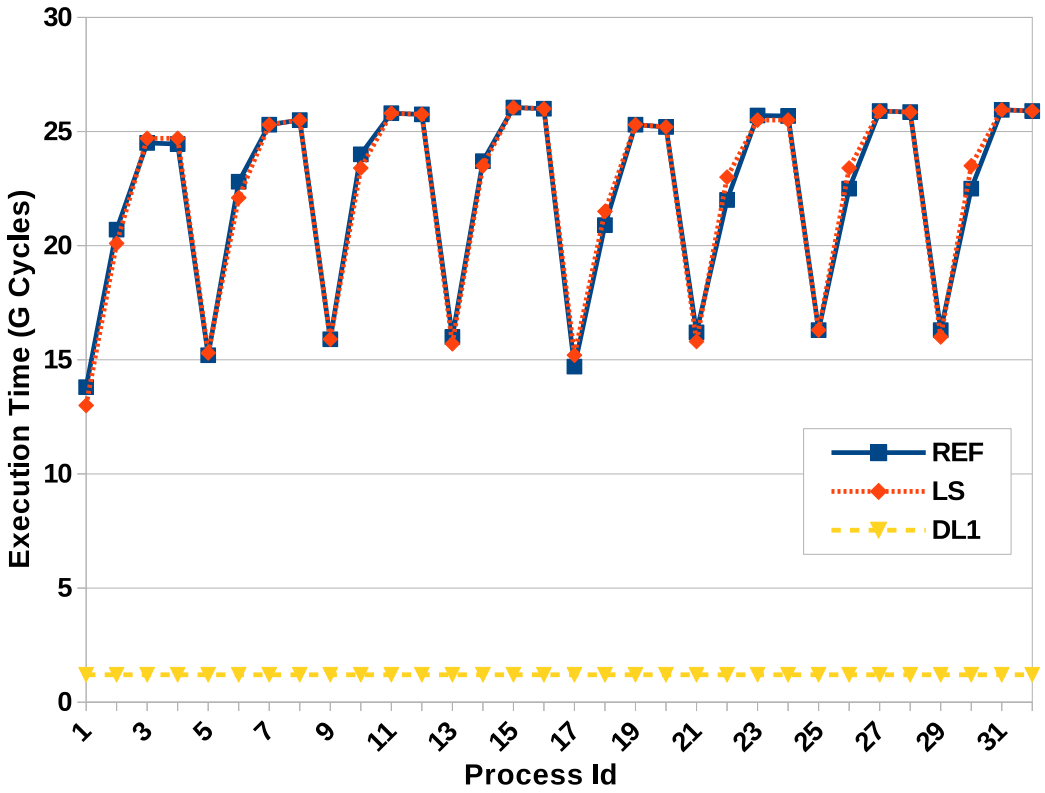


Figure 4.10: Streams Analysis on PNBench

The REF curve corresponds to the performance of the original code. The LS (resp. DL1) curve corresponds to the DECAN variant where all FP instructions have been suppressed (resp. all data accesses are forced to come out of L1).

According to the methodology, the next step consists in gaining more insight on the loop characteristics through *performance overview*, hence, the `LS` and `DL1` DECAN variants are used. The corresponding results shown in Figure 4.10 indicate a strong domination of data accesses, with the `LS` curve being well over the `DL1` curve and matching the `REF` one. Consequently, the investigation follows the `LS` subtree.

Table 4.4: PNBench MTL Results

Group	Instructions	Pattern
G1	Load (Double)	8*i1
G6	Load (Double)	8*i1+217600*i2+1088*i3
G5	Store (Double)	8*i1+218688*i2+1088*i3

This table shows PNBench MTL results for the three most relevant instruction groups.

In order to get more information on data accesses, we use MTL. Six instruction groups are detected but only three of them contain relevant SSE instructions dealing with FP arrays. The MTL results shown in Table 4.4 indicate a simple access pattern

for group G1 (stride 1) and, for groups G6 and G5, more complicated patterns which need to be optimized. As a result, in this step we are able to characterize our memory accesses with precision. Though, it leaves us with two accesses and no possibility to know which one is the most important.

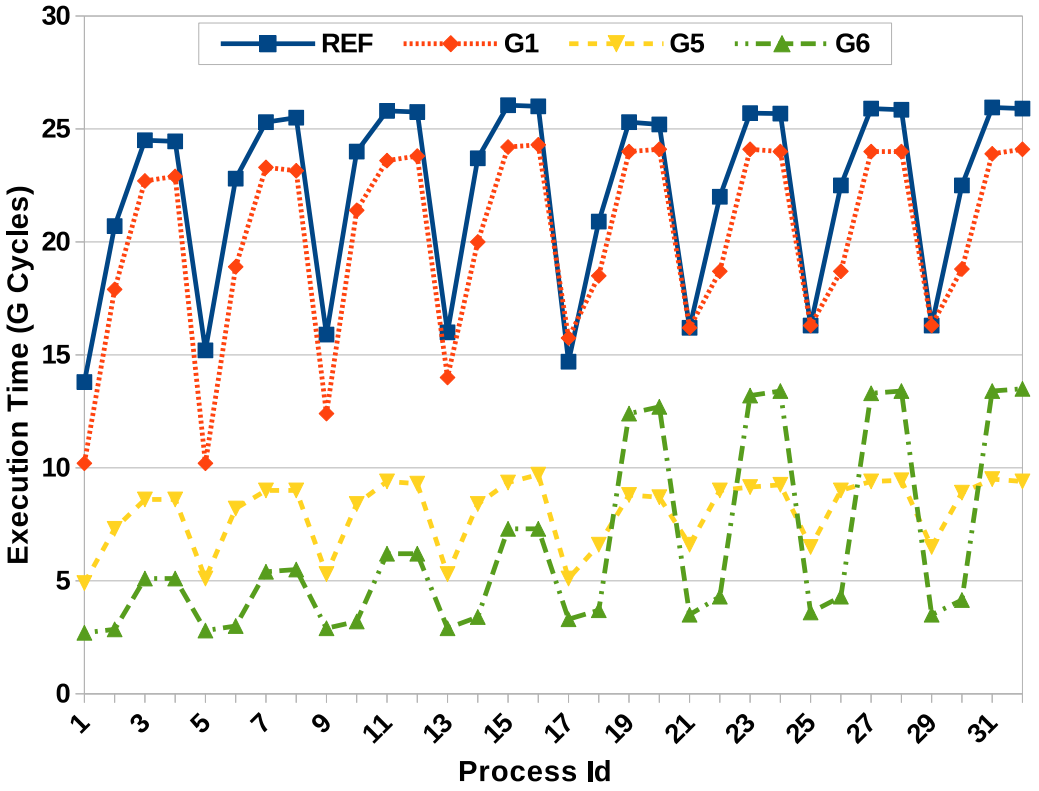


Figure 4.11: Group cost analysis on PNBench

Each group curve corresponds to performance of the loop while the target group is deleted. The original code performance (REF) is used as reference.

At this point, we return to our notion of ROI provided through Differential analysis and apply the DECAN `del-group` transformation for each of the three selected groups. The `del-group` results shown in Figure 4.11 clearly indicate that G6 is the most costly group by far: it should be our first optimization target.

With the finding of the delinquent instruction group, the analysis phase comes to its end. The next logical step is to try and optimize the targeted memory access. Fortunately, the information given by MTL reveals an interesting pathology. The access pattern of the instruction of interest has a big stride in the innermost loop (1088*i3) and a small one in the outermost loop (8*i1). In order to diminish the access penalty we perform loop interchange between the two loops, which results in a considerable performance gain at the loop level with a speedup of *7.7*x and consequently a speedup of *1.4*x on the overall performance of the application.

4.5.2 RTM

Reverse Time Migration (RTM) [99] is a standard algorithm used for geophysical prospection. The code used in this study is an industrial implementation of the RTM algorithm by the oil & gas company TOTAL.

Our RTM code operates on a regular 3D grid. More than 90% of the application

execution time is spent in two functions, `Inner` and `Damping`, which execute similar codes on two different parts of the domain: `Inner` is devoted to the core of the domain while `Damping` is used on the skin of the domain. Standard domain decomposition techniques are used to spread the workload on multicore target machines. Since the grid is uniform, load balancing can be easily tuned by using rectangular sub-domain decomposition and by properly adjusting the sub-domain size.

All experiments are done on a single socket machine, which contains a quad-core Intel Xeon E3-1240 processor with a cache hierarchy of 32KB (L1), 256KB (L2) and 8MB (shared L3).

Step 1: The original version of the code is provided with a default non-optimized blocking. The first analysis on the OpenMP subtree reveals an imbalanced work sharing. A second analysis done at the level of the *performance overview* subtree shows that the code is highly bounded by memory operations. In order to fix this, we focus on the blocking strategy. As a result it turns out that the default block size is responsible for both the load imbalance between threads and the bad memory behavior. We can then select a strategy which provides a good balance at the work sharing level as well as a good trade-off between the LS and FP streams. However, we note that, to obtain an optimal strategy, a more dedicated tool should be used.

Step 2: The second step of the analysis consists in going further in the OpenMP subtree and checking how the RTM code performs in term of coherency. As explained earlier, the structure of the code induces a non-negligible coherency traffic. Figure 4.12 shows experimental results after applying the S2L transformation on RTM. While the x-axis details loops respectively identified from `Inner` and `Damping`, the y-axis represents speedups over the original loops. The results indicate a negligible gain due to canceling potential coherency modifiers and a minimal gain, observed on two loops, due to a complete deletion of the stores. Consequently, we can conclude that maintaining the overall coherency state remains negligible, therefore, there would be no point in going further in this direction.

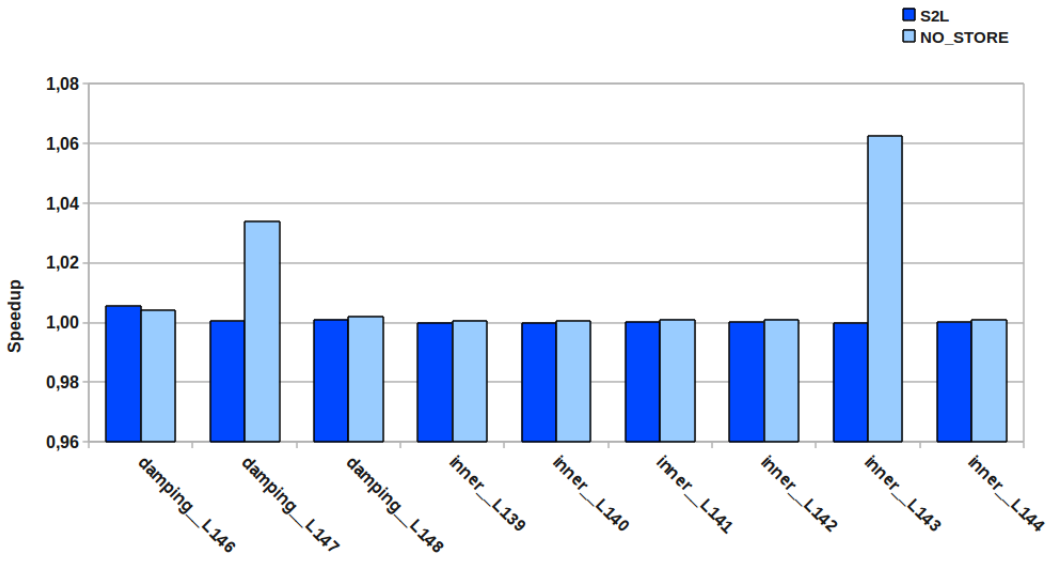


Figure 4.12: Evaluation of the Cost of Cache Coherence Protocol

The S2L variants show similar performance as their corresponding reference versions. The NO_STORE variants are the same, except for two loops which present an observable store cost.

4.6 Related Work

Improving an application's efficiency requires identifying performance problems through measurement and analysis but assessing bottlenecks impact on performance is much harder. To achieve that, most researchers consider a qualitative approach. TAU [86] represents a parallel performance system that addresses diverse requirements for performance observation and analysis. Although performance evaluation issues require robust methodologies and tools, TAU only offers support to the performance analysis in various ways, including instrumentation, profiling and trace measurements.

Tools such as Intel VTune [27], GNU profiler (Gprof) [100], Oprofile [31], MemSpy [101], VAMPIR [90], and Scalasca [89] provide considerable insight on the application's profile. In term of methodology Scalasca, for instance, proposes an incremental performance-analysis procedure that integrates runtime summaries based on event tracing. While these tools help hardware and software engineers find performance pathologies, significant manual performance tuning remain for software improvements, for example, selecting instructions in particular part of a program.

PerfExpert [87], HPCToolkit [102], and AutoSCOPE [103] pinpoint performance bottlenecks using performance monitoring events. Furthermore, while PerfExpert suggests performance optimizations, AutoSCOPE extends PerfExpert by automatically determining appropriate source-code optimizations and compiler flags. Contrary to PAMDA, the considered tools do not provide a methodology presenting the cost related to the identified bottleneck. ThreadSpotter also helps a programmer by presenting a list of high level advice without addressing return on investment issues: what to do in case of multiple bottlenecks? How much do bottlenecks cost?

Interestingly in [104, 105], the authors present an automated system that fingerprints the pathological patterns of the hardware performance events and identifies the pathologies in applications, allowing programmers to reap the architectural insights. The proposed technique is close to the current work and includes pathology description through microbenchmarks as well as pathology identification using a decision tree. However, in order to evaluate usual performance pathologies, PAMDA additionally integrates pathology cost analysis.

The above survey indicates that performance evaluation requires a robust methodology, but traditional methods do not help much with coping with the overall hardware complexity and with guiding the optimization effort. Also, previous works focus on performance bottleneck identification providing optimization advice without providing potential gains. The previous factors motivate to consider PAMDA as the only methodology combining both qualitative and quantitative approaches to drive the optimization process.

4.7 Acknowledgments

We would like to thank Michel Masella for the access to his POLARIS(MD) code and Henri Calandra and Asma Farjallah for the access to the RTM code.

This work has been carried out by the Exascale Computing Research laboratory, thanks to the support of CEA, GENCI, Intel, UVSQ, and by the PRiSM laboratory, thanks to the support of the French Ministry for Economy, Industry, and Employment through the PERFCLOUD project. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not

necessarily reflect the views of the CEA, GENCI, Intel, or UVSQ.

4.8 Conclusion and Future Work

Application performance analysis is a constantly evolving art. The rapid changes in the hardware mixed with new coding paradigms force analysis tools to handle as many pathologies as possible. This can only be achieved at the expense of usability. At the end, application developers work with extremely powerful tools but they have to face significant differences and difficulties to use them.

This paper illustrates the usefulness of performance assessment combining static analysis, value profiling and dynamic analysis. The proposed tool chain, PAMDA, helps the user to quickly identify performance pathologies and assess their cost and impact on the global performance.

The goal in using PAMDA is to make sure that the right effort is spent at each step of the analysis and on the right part of the code. Furthermore, we try to create some synergy between different tools by combining them in a unified methodology. We provide some case studies to illustrate the overall analysis and optimization process. Experimental results clearly demonstrate PAMDA's benefits.

The provided methodology is admittedly far from being finished. Our constant challenge is to keep improving it as well as working towards full automation. We also aim to enlarge it for other kind of paradigms through the integration of analyses provided by complementary tools such as Scalasca, Vampir and TAU. Additionally, refining optimization investigations is crucial in order to make it more user-friendly.

Chapter 5

Extending the Cape Model

Cape is a model answering the need for a global view of a machine's performance parameters, and allowing users to evaluate the impact of a HW or SW change across the whole system.

In this chapter, we will present the work we have done to help improve and extend the Cape modeling capabilities on modern Intel microarchitectures. These improvements range from sanitizing Cape inputs to adding key model components.

This work focuses mainly on Sandy Bridge, but we will also mention characteristics from Ivy Bridge and Haswell when they differ.

5.1 Presentation of Cape

Cape is a loop-centric performance model that is mostly aimed at hardware / software codesign, but can also be used to precisely decompose performance for optimization purposes.

While its principles were used and validated in the context of frequency scaling in [70], the work presented in this chapter helped refine its approach, bringing more modeling details, precision and matching the hardware more closely.

5.1.1 Core Principles

We will begin our presentation of Cape by introducing the concept of *nodes*, small linear models targeting specific hardware components or characteristics.

Each node has a *bandwidth*, i.e. a certain amount of work it can process every cycle, and is essentially a hardware characteristic. However, how much these nodes are solicited (which we will call the node's *workload*) is program dependent, and hence a software attribute. From a node's point of view, the time taken to process a certain workload is defined by:

$$node\ time\ =\ workload\ /\ node\ bandwidth$$

As an example, a "branch node" could model the impact of branch instructions on Sandy Bridge by having a workload equal to the number of branches needed to run the code, and a bandwidth of 1 branch per cycle (as there is only one branch unit in the the SNB microarchitecture). Branch instructions would then necessarily take at least $number\ of\ branches\ /\ bandwidth$ cycles to get executed.

Cape combines these individual nodes, assuming they are completely independent from one another:

$$cape\ time\ =\ \max_{all\ nodes}\ (node\ time)$$

The Cape model is consequently a bandwidth-centric model comprising a linear equations system. Modifying hardware characteristics is hence particularly easy (as only about modifying bandwidth values for individual nodes), and presenting

a trivial numerical complexity: exploring potential designs is hence made a rather computationally trivial task. Another interesting property of this model is that it allows the implementation of a fast inverse performance function, giving a list of hardware changes needed to reach a target performance.

5.1.2 Identifying Nodes and their Bandwidths

Nodes may vary both in nature and in bandwidths depending on the target microarchitecture. For instance, one may question the relevance of a FP division node in the context of old CPUs not supporting them natively. Which nodes to use, and what their bandwidth should be set to is hence completely hardware-specific and can be obtained through various means:

1. Documentation: manuals such as [106] can give clear information on a microarchitecture's main components and potential bottlenecks.
2. Benchmarks: empiric data (e.g. [107]) can complement (and sometimes supplement) the official documentation, especially to fine-tune bandwidths.
3. Using DECAN variants. We will discuss this in more details in Section 5.1.4.
4. Trial and error: running Cape through various codes can reveal important error cases, which may be due to overlooked components.

Previous implementations of Cape modeled a restricted number of components, mostly targeting functional units and the CPU's ability to compute floating point operations. This work is about complementing and refining previously existing nodes to better cover the complexity behind modern microarchitectures such as Sandy Bridge.

5.1.3 Getting Node Capacities

Node capacities are workload-dependent, and can be obtained with:

1. Static analysis: CQA [21] can provide detailed instruction and uop counts for a binary loop, precisely quantifying workloads.
2. Dynamic analysis: certain information is hard or impossible to obtain purely statically, and may be obtained through dynamic runs. This is the case e.g. for memory workloads.

We will see this in more details in Section 5.4.

5.1.4 Isolating the Memory Workload

The memory subsystem can respond in surprisingly diverse ways to a given workload, depending on characteristics such as the number and type of used instructions (scalar vs. vector, single precision vs. double precision), the access strides or the number of memory streams being accessed simultaneously. The consequence is that setting a constant bandwidth for memory performance is a difficult task and is still a work in progress.

The Cape model we worked with hence uses codelet-specific bandwidths to characterize the memory system. While it goes against the original premise that nodes'

bandwidth should be entirely defined by the hardware, it allows us to put this particular issue aside until a better solution is found. Some memory-related nodes still have workload-independent bandwidths, as we will see in Section 5.4.

We hence use DECAN [37]'s "LS" variant to isolate the memory workload from the target loop, and then measure the memory bandwidth obtainable with the loop's specific access pattern.

A practical consequence of this lack of "global" bandwidth for some of the memory nodes is that only codelets with similar memory profiles can be considered simultaneously without introducing importance modeling errors.

5.1.5 Saturation Evaluation

Measurements can be used for validation purposes, allowing to check Cape's time projections' validity for the machine in question. This is typically done through the evaluation of *saturation*: at the individual node level, saturation is defined as:

$$node\ saturation\ =\ node\ time\ /\ measured\ loop\ time$$

At the Cape level, *system saturation* is defined as:

$$system\ saturation\ =\ \max_{all\ nodes}\ (node\ saturation)$$

Cape relies on the system being fully saturated to properly decompose performance and estimate the impact of bandwidth changes. On the one hand, *undersaturation* (i.e. the system saturation being below 100%) will cause Cape to underestimate the time taken to process a given workload, and hence be too optimistic. This may happen when an important node is missing, or when a node's bandwidth was incorrectly set to too high a value. On the other hand, *oversaturation* (i.e. the system saturation *exceeding* 100%) will cause Cape to overestimate the needed execution time for a workload and give pessimistic results. This can happen when having set a node's bandwidth to an incorrectly low value.

DECAN can also be used for refining this validation process, isolating the workload for FP or LS related nodes, and hence allowing for saturation checks to be done at a finer level.

5.1.6 Cape Inputs

We use inputs generated using the framework presented in Chapter 3.

DECAN variant LS is used to get memory bandwidths for workload-dependent node bandwidths and fine-grain validation data for the workload-independent nodes. Variant REF is used to get reference times for the target loop, used for Cape time projections' validation. Variant FP is used to get fine-grain validation data for FP nodes' validation.

5.2 DECAN Variant Refinements

As Cape depends on DECAN variants both to get bandwidth inputs for memory nodes and to get validation comparison points, it is particularly important that their transformation rules be as faithful as possible to their intended purpose.

We will describe a few refinements to the main DECAN variants we use (REF, LS and FP) that were made with this intent.

5.2.1 Tackling Partial Vector Register Loads

Register loads fill part of a register with data coming from the memory hierarchy. The amount of data copied this way depends on the instruction's type; for instance, *MOVSD* copies a Single Double (i.e. 8 bytes), and *MOVAPS* copies an entire 128-bit register's worth (i.e. 16 bytes).

Perhaps counter-intuitively, most register loads actually set the part of the register that was not fetched from memory to 0. For instance, loading value "5.6" with a *MOVSD* in a 128-bit register formerly containing DP values "1.2 3.4" will produce a new register value of "0.0 5.6" (and not "1.2 5.6").

This presents the important advantage of causing load operations not to depend on the state of the target register, increasing the potential Instruction Level Parallelism.

Special load instructions can be used so as not to erase the not-loaded part of the register, for instance *MOVHPD* (which loads 8 bytes in the upper half of a 128-bit register, preserving the lower one) or *VINSERTF128* (which does the same with 16 bytes and a 256-bit register). However, this comes at the cost of a dependency on the previous register value, and an extra uop is needed to stitch the loaded values and the old register value together.

We hence modified DECAN variant transformation rules to take it into account:

1. For LS: load instructions preserving parts of the target register should be treated like fused load + arithmetic instructions, and the instruction transformed to only keep the load component. For instance, "*MOVHPD (%rax), %xmm0*" should be transformed into a "*MOVSD (%rax), %xmm0*".

2. For FP: such instructions should not be removed altogether, and instead transformed to only keep the arithmetic component. We chose *MOVLHPS* for this purpose for SSE instructions due to being similar to the original instruction's arithmetic uop's expected behavior (same dispatch port and same expected latency on SNB, IVB and HSW [107]). For AVX, we had to pick a different instruction due to *MOVLHPS* not having an AVX version, and chose *VANDPS* for the same reasons as above.

The same process should be applied to these special instructions when they are employed as stores, as an extra uop is still needed to extract the intended value.

This transformation helps keep only memory workloads on LS, and all of the needed arithmetic in FP.

5.2.2 The Case of Floating Point Divisions

Some additions and multiplications can be more or less complex depending on the operands (even taking aside the thorny issue of denormalized numbers). For instance, it is obvious by human standards that adding 0 to a number, or multiplying it by 1 is simple. Similarly, multiplying by 10 is also trivial when using the decimal notation.

This applies in base 2 as well, leaving some optimization opportunities for hardware designers. Interestingly, using pipelined functional units makes it both harder to implement such tricks while also making them less relevant: if a functional unit is going to produce 1 result per cycle anyway, why even bother with operand-dependent optimizations?

However, divisions (and in particular, floating point ones) are not (fully) pipelined on Intel microarchitectures, meaning such data-dependent optimizations can (and do) still actually improve their throughput [108]. The main factor in determining how long a division is going to take is the size of the mantissa (i.e. the position of the least- significant bit) in the divisor.

The effective division bandwidth being data-dependent is an issue for Cape, as the bandwidth for floating point divisions cannot be reduced to a simple constant anymore. Furthermore, the transformations applied in the FP DECAN variant can have an impact on input operands for divisions, creating a discrepancy between the workload's complexity for REF and for FP. We address these two issues by modifying division instruction operands so that they always fall in the worse case (which we assume to be more frequent in real-world work loads) in both variants. For instance, "*DIVPD %xmm0, %xmm1*" would be preponed with "*MOVAPS global variable containing 1.0, %xmm0*" and "*MOVAPS global variable containing an approximation of pi, %xmm1*".

This is however not a perfect solution as it disrupts the original loop's semantics, but is effective at ensuring the complexity for divisions is always the same.

The very same reasoning and fix apply for other variable-latency operations such as square roots.

5.3 Front-End Modeling Subtleties

The x86 and x86-64 instruction sets are complex, in good part due to Intel preserving backward-compatibility with previous CPU versions: they can only evolve by growing bigger, and cannot be reworked for efficiency purposes.

This complexity is carried over in the Front-End, limiting its operation speed.

First overlooked in our Sandy Bridge Cape model implementation, some of its caveats forced us to dive deeper in its limitations to create a realistic FE node.

5.3.1 Accounting for Unlamination

With a theoretical bandwidth of 4 uops per cycle, the FE should rarely be the bottleneck in SNB. Indeed, *microfusion* allows two different operations to be combined in a single FE uop in some cases, increasing the effective FE bandwidth [109]:

1. Stores: stores require both a "store data" and a "store address' components, which need to be dispatched on different ports. Microfusion allows them to occupy a single slot until they reach the Back-End.

2. Arithmetic instructions fetching data from memory (e.g. *MULPS (%rax), %xmm0* multiplies the contents of the address pointed by %rax with the value of %xmm0): as with stores, the FE can often make the two components be sent to the Back-End as a single uop.

It is hence possible possible to keep all 6 of SNB's execution ports busy with its FE bandwidth:

- A fused vector multiply and load uop will use ports 0 (for the arithmetic component) and one of ports 2 or 3 (for the load).

- A fused store uop will use port 4 (for the "store data" component), and whichever port (among ports 2 and 3) that was not used for the load for the "store address" component.

- A branch uop will use port 5.

- A vector addition will use port 1.

Unfortunately, limitations on microfusion force some uops to be split prematurely (or *unlaminated*) in the uop queue, degrading the effective FE bandwidth [110].

Table 5.1: Finding Unlamination Rules

Codelet	Measured		Unlamination Criteria			
	SNB2	*HSW*	**>= 3 regs.**	**>= 3 regs., except stores**	**>= 4 regs.**	**Never**
elmhes_10_de	26	18	**26**	22	*18*	*18*
elmhes_10_dx	34	30	**34**	*30*	*30*	22
svdcmp_6_dx	49	49	***49***	***49***	***49***	41
svdcmp_11_dx	8	8	***8***	***8***	***8***	***8***

Results are in post-unlamination uops per iteration. Measurements values were obtained by measuring the UOPS_ISSUED.ANY *hardware event. The codelets ending with an* x *are AVX variations of the main codelets, and are interesting here as they can use non-destructive 3-address statements. Presented codelets were selected to cover a wide range of possible unlamination scenarios.*

Unlamination criteria values in bold (with a blue background) match the measured values for Sandy Bridge; those in italic (with a red background) match measurements for Haswell. Values in both bold and italic (with a purple background) match either microarchitecture.

The >= 3 regs. unlamination criterion matches the limitations described in the official documentation for Sandy Bridge. However, it is clearly pessimistic for Haswell, so we tested relaxed criteria to try to expose the new unlamination rules. Criterion >= 4 regs. is the only one matching Haswell measurements on all studied codelets.

The main restriction seems to be related to the number of register operands uop need, as can be seen in Table 5.1:

1. On SNB and IVB, fused uops using more than 2 register slots will be unlaminated. For instance, *MULPS (%rax), %xmm0* will remain fused, but *MULPS (%rax, %r10, 8), %xmm0* will not (3 register operands).

2. On HSW, we observed this behavior was improved (through the use of the *UOPS_ISSUED.ANY* hardware counter which counts the number of uops issued by the RAT and the number of usable register slots was increased to 4. As a consequence, SSE instructions are not unlaminated anymore (for this specific reason), but the problem still exists for AVX instructions such as *VMULPS (%rax, %r10, 8), %xmm0, %xmm0.*

Other cases of unlamination exist, but in more fringe circumstances, limiting their impact.

Unlamination makes the FE be suddenly relevant again in performance modeling: we hence added a new node whose bandwidth is 4 uops per cycle, but for which the workload is based on post-unlamination uop counts.

5.3.2 Ceiling Effect

We also stumbled against another restriction in Sandy Bridge and Ivy Bridge's FE that prevents uops from different iterations from getting sent to the Back-End in the same cycle.

Table 5.2: Front-End Stress Experiment Example

#Line	Instruction	Nb Uops to Issue
1	VXORPS %xmm1, %xmm1, %xmm2	1
...	VXORPS %xmm1, %xmm1, %xmm2	1
n	VXORPS %xmm1, %xmm1, %xmm2	1
n + 1	SUB $1, %rdi	0.5 (macrofused)
n + 2	JG .LOOP	0.5 (macrofused)

We can adjust the number of uops to be issued (per iteration) by modifying the number of $VXORPS$ instructions. Getting a loop with x uops would be achieved by having $n = x - 1$ $VXORPS$ and the macrofused uop.

Note: $VXORPS$ instructions operating on the same input registers (zero-idiom) are handled in the RAT and made to point to a zero-filled register. They do not need to get dispatched, so execution ports are only involved for the macrofused branch uop.

We developed simple microbenchmarks (see Table 5.2) to expose this phenomenon.

Figure 5.1 shows our experimental results on SNB and HSW. IVB has identical results to SNB though they are not presented separately there. Fixing this behavior could hence be achieved by applying an integer ceiling when computing the saturation for the FE node, i.e. issuing x uops per iteration should be estimated as taking $ceiling(x\ /\ FE\ bandwidth)$ instead of just $x\ /\ FE\ bandwidth$. For instance, issuing 5 uops with a FE bandwidth of 4 should be counted as taking 2 cycles.

However, ceiling operations can cause issues for evaluating linear equation systems as they are discontinuous on integers. We instead shifted the burden on the workload, i.e. a loop body containing x uops should be counted as having $x\ +\ x\ \%\ FE\ bandwidth$ uops instead.

Table 5.1 also shows this issue was addressed in Haswell, so the ceiling does not have to be applied anymore in HSW's FE node.

5.3.3 New Front-End DECAN Variant

The different limitations the FE suffers increased the interest in isolating its role in explaining loop performance. We could hence create an *FES* DECAN variant substituting all (non-control flow) instructions with an equivalent number of multibyte-NOP micro-ops. Multibyte NOPs are preferred over regular one-byte ones to avoid complications with the uop cache when too many tiny instructions are placed contiguously [111].

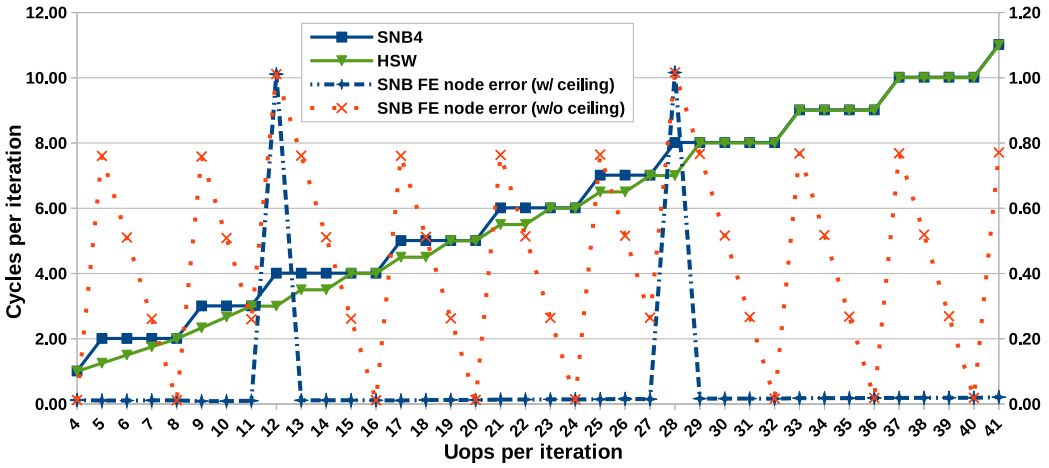


Figure 5.1: Exposing the Front-End Ceiling Effect

*Experiments start from 4 uops per iteration (representing the point where the FE starts being a bottleneck for our generated codelets) and stop at 41, where the worst case misprediction for not taking into account the ceiling effect (*0.75 *cycles) would only represent a* 0.75 / 11 = 7% *error.*

First, we can notice steps on SNB4's curve confirming the existence of the ceiling effect on this microarchitecture. The SNB FE node error (w/ ceiling) *curve captures it well, with errors only on point 12 and 28 due to unfortunate instruction alignments preventing the macrofusion of the CMP and JB instructions and increasing the effective uop count by 1. On the other hand, the* SNB FE node error (w/o ceiling) *curves does not, causing errors of up to nearly* 0.8 *cycle (against the expected* 0.75*).*

HSW results are interesting, and show the cycles per iteration increase in a nice linear fashion until 11 uops. It evolves in an in-between manner between 12 and 28 uops, before joining back SNB's curve starting from 29 uops onwards. It is not clear why 29 is the turning point as the available uop queue's size should be of 56 in Haswell, but the improvement tackles the cases in which the phenomenon is most important.

IVB results are extremely close to SNB's on the presented range (1% difference in the worst case) and were consequently left out to simplify the graph.

It is hence better to use a ceiling-adjusted formula to model the FE's bandwidth in SNB and IVB, and one without for HSW.

Single-byte NOPs are inserted when necessary to account for unlamination. (E.g. a store with a complex address computation will be replaced by both a multibyte NOP *and* a single-byte one).

Just as with the FP variant, FES can be used to provide validation numbers for Cape modeling.

5.4 Back-End Modeling Improvements

The Back-End part of the core pipeline was also significantly overhauled by adding fine-grain nodes for the execution engine and the memory hierarchy.

While improving the model's precision was the primary motivation for adding most of these new nodes, some were added with the model's flexibility in mind.

We will present these changes and their purpose in this section.

5.4.1 Dispatch

Dispatch-related nodes cover execution ports and the handling of inter-iteration dependencies.

5.4.1.1 Execution Ports

Execution ports act as access points (or gateways) for the functional units (FUs) they manage. Each FU is dedicated to a given port.

There are 6 execution ports in SNB and IVB, and 8 in HSW.

Each port can forward one uop to an FU per cycle, and so even if one of their FU is currently active: they consequently all have a bandwidth of 1 uop / cycle.

Their workload is obtained thanks to CQA's port distribution metrics, but capacities could also be alternatively obtained by using hardware counters. For instance, $UOPS\_DISPATCHED\_PORT.PORT\_5$ counts the number of uops dispatched on port 5 in SNB.

5.4.1.2 Inter-iteration Dependencies

Inter-iteration dependencies are dependencies that carry over from one iteration to the other, potentially severely reducing the Instruction Level Parallelism exploitable by the out-of-order engine.

They are governed more by instruction latencies than by their bandwidth (as the execution of instructions from the same dependency chain cannot be pipelined), making them hard to evaluate for a bandwidth-centric model such as Cape. The length of inter-iteration dependency chains represent an upper bound for the time needed to complete an iteration.

Our solution was to add a pseudo-node keeping track of such dependencies: it can figuratively eliminate inter-iteration dependencies at a rate of 1 cycle of latency per cycle; and its workload is simply the length of the biggest inter-iteration dependency chain as determined by CQA.

It could be improved by keeping track of the different instructions involved in all the inter-iteration dependency chains and giving users the ability to modify the latency for each instruction type (allowing Cape to recompute its effective workload on the fly), but is helpful in its current form as a first way of tackling this issue.

5.4.2 Functional Units

Key functional units (e.g. FP adder) were part of the base Cape nodes and did not need adding.

However, they could be split into more detailed nodes to add more flexibility to the model and improve the overview of performance bottlenecks. For instance, on SNB, the FP adder:

1. Can process 1 FP add uop per cycle (original bottleneck).
2. Can execute a maximum of 1 SP or DP scalar addition per cycle (scalar performance bandwidth).
3. Can execute up to 4 SP (2 DP) additions when using SSE vector instructions.
4. Can execute up to 8 SP (4 DP) additions when using AVX vector instructions.

Operation	Assigned Workload							
	Add node 1	Add node 2	Add node 3	Add node 4	Add node 5	Add node 6	Add node 7	Add node 8
256-bit vector add	1	1	1	1	1	1	1	1
128-bit vector add	1	1	1	1				
DP scalar add	1	1						
SP scalar add	1							
	Physical adder length (256-bit)							

Figure 5.2: Modeling the FP Add Functional Unit

This figure represents the modeling of the "FP add" functional unit in Sandy Bridge.

All FP additions are theorized to be performed by a functional unit whose full potential is only harnessed for 256-bit vector additions. Different nodes hence allow to account for the maximum theoretical bandwidth of scalar additions being of 1 per cycle (on Sandy Bridge) while keeping track of how well the unit is used.

Furthermore, it makes it possible for users to model the rate of scalar and vector operations separately.

The adder is hence virtually split in 8 nodes (as illustrated in Figure 5.2), representing each of the possible addition "lanes" in the abstracted adder:

1. An AVX full-vector addition uses all lanes.
2. An SSE full-vector addition uses only the first 4 ones.
3. A DP addition uses the first two lanes.
4. An SP addition only uses the first one.

Having more nodes allows Cape to tackle the same issue from different points of view, and e.g. evaluate how often vector units really are used for the considered loops, and how relevant the unit's width really is.

This flexibility will also be used in Chapter 6 when evaluating loops' vectorization potential.

5.4.3 Memory Hierarchy

Memory modeling went through important changes with the introduction of detailed memory nodes for each step of the memory hierarchy and an explicit handling of TLB misses.

5.4.3.1 L1 Accesses

L1 accesses can be approached in the same fashion as scalar and vector FP operations seen earlier. There are two load units, each behind separate ports (P2 and P3).

On SNB and IVB, each of these units can fetch:

1. 128 bits per cycle in case of vector loads.
2. 8 bytes per cycle in case of scalar DP load.
3. 4 bytes per cycle in case of scalar SP load.

Interestingly, an implication is that a 256-bit load keeps a load unit busy for 2 cycles.

We can hence split the 128-bit data paths in 4 lanes of 4 bytes each, and create as many matching nodes.

Operation	Assigned Workload			
	Store node 1	Store node 2	Store node 3	Store node 4
256-bit vector store	2	2	2	2
128-bit vector store	1	1	1	1
DP scalar store	1	1		
SP scalar store	1			
	Store data path length (128-bit)			

Figure 5.3: Modeling the Store Functional Unit

This figure represents the modeling of the "data store" functional unit in Sandy Bridge.

The reasoning behind splitting the functional unit in different nodes is exactly the same as for FP additions (seen in Figure 5.2). The main difference resides in the physical data path being of only 128-bit in Sandy Bridge, making 256-bit stores keep each store node busy for twice the regular amount of time. This limit is addressed in Haswell with data paths being extended to 256 bits.

Stores units operate in an identical fashion, though there is only one in the studied microarchitectures (behind P4). The node modeling for stores is shown in Figure 5.3.

Capacities for each node is obtained with CQA, which classifies and counts memory instructions depending on how many bytes they move.

Note: the width of units' data paths was increased to 256 bits in HSW, doubling the number of 4-byte lanes and allowing 256-bit vector loads and stores to complete in a single cycle.

5.4.3.2 L2, L3 and RAM

The memory hierarchy provides data in small chunks of 64 bytes called "cache lines".

We can thus create nodes representing the read and write traffic for these cache lines between these different levels:

1. *L2RW* for L1 and L2.
2. *L3RW* for L2 and L3.
3. *RAMRW* for L3 and RAM.

The exact number of nodes may vary depending on the microarchitecture and the exact SKU. For instance, SLM does not have an L3 cache, and CPUs equipped with *Crystalwell* on-chip DRAM have the equivalent of an L4.

The workload for these nodes is the number of cache line transfers caused by the loop's execution, which can be hard to get statically for various reasons:

1. Access patterns are not always determinable statically. They may even change depending on the input data set (e.g. Leading Dimension strides in multi-dimension arrays).
2. Hardware prefetchers can add unknowns as to which lines really are fetched.
3. Cache structures may react in unexpectable ways depending on runtime parameters (e.g. associativity conflicts), causing some lines to be evicted prematurely and fetched several times.

We hence use hardware counter measurements to obtain capacities, allowing us to use the actual workload instead of speculated values.

Table 5.3: Traffic Count Formulas for HSW, SLM and SNB / IVB

Uarch	L2RW	L3RW	RAMRW
HSW	L1D.REPLACEMENT + L2_TRANS.L1D_WB	L2_LINES_IN.ALL + L2_TRANS_L2_WB	CAS_COUNT.RD + CAS_COUNT.WR
SLM	MEM_UOPS_RETIRED.L1_MISS_LOADS + PF_L1_DATA_RD + DEMAND_RFO + (COREWB - OFFCORE_RESPONSE.COREWB.L2_MISS.ANY)	N / A	LONGEST_LAT_CACHE.MISS + L2_WB
SNB / IVB	L1D.REPLACEMENT + L1D_WB_RQST.ALL	L2_LINES_IN.ALL + L2_TRANS.L2_WB + L1D_WB_RQST.MISS	CAS_COUNT.RD + CAS_COUNT.WR

These counters and formulas represent both the read and the write traffics at each level of the memory hierarchy. For instance, on SNB, L1D.REPLACEMENT *represents the number of cache lines getting imported into L1 from upper cache levels, and* L1D_WB_RQST.ALL *the number of cache lines* written back *to upper levels after getting modified by the execution engine.*

In SNB, the L3RW formula has 3 components due to writebacks being counted separately depending on whether the involved cache lines are present in L2. Indeed, the L2 is not inclusive, and there is no guarantee a line evicted by L1 is present there.

The SLM L2RW formula is more complicated than its peers due to not having straightforward read and write counters. The number of fetched lines is hence decomposed in: explicit demand loads, loads due to the fetch-on-write *mechanism and prefetched lines. The number of written lines is obtained by counting all writebacks everywhere (L1 and L2 writebacks) and subtracting L2's.*

The CAS_COUNT *counters are only available on the server variants of SNB, IVB and HSW. They can be substituted with* DRAM_DATA_READS *and* DRAM_DATA_READS *[73].*

Adjustments may be necessary in case of non-temporal stores, which we did not or encounter in studied loops.

Official counter descriptions can be obtained from [112].

Table 5.3 shows counters and formulas that can be used for this purpose.

As explained in Section 5.1.4, the difficulty in determining a single global bandwidth for memory nodes pushed us to instead determine bandwidths on a per codelet basis. Bandwidths for the L2RW, L3RW and RAMRW nodes are hence defined using the LS DECAN variant for the considered loops:

$$node\ bw = node\ workload\ /\ LS\ variant\ execution\ time$$

It can be further refined to instead use the maximum effective LS bandwidth across all data sets (when more than one data points are available for a given codelet). This shortcut limits our ability to accurately consider codelets with different memory characteristics at the same time.

5.4.3.3 Translation Lookaside Buffer

Page walking (i.e. fetching a page translation from the memory hierarchy rather than from the Translation Lookaside Buffers) is a costly a) latency-bound b) non-pipelined process with the potential of being a significant bottleneck in loops with large strides.

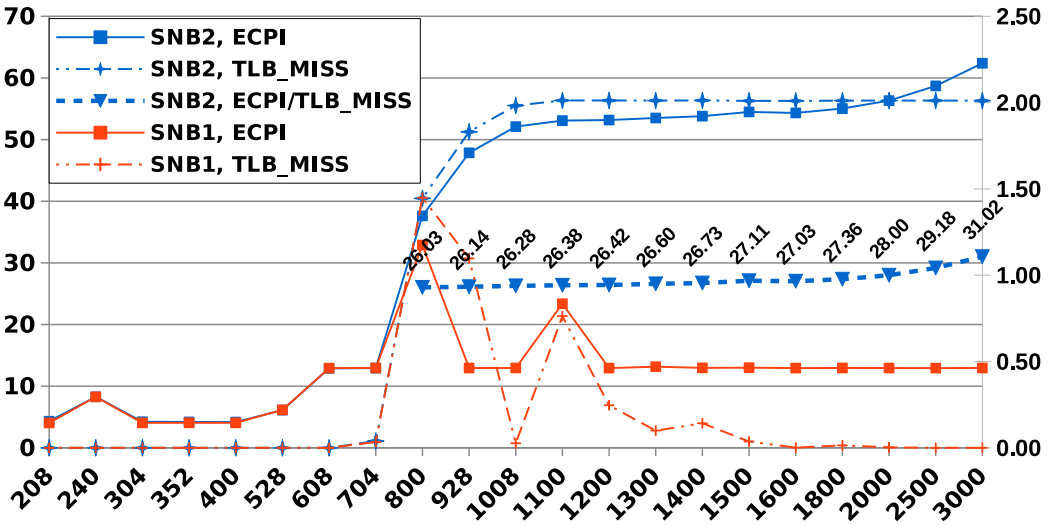


Figure 5.4: Impact of TLB Misses

Results show the LS variant of hqr_15_se and are normalized per iteration. ECPI represents the execution time in cycles. The TLB_MISS data is shown on the secondary y-axis. All data sets fit in L3 for this codelet.

Hqr_15_se triggers TLB misses starting from point 800, reaching 2 TLB misses per iteration at point 1008 for SNB2. SNB1 prevents most of the problem by using Transparent Huge Pages [113], and hence reveals page walks are indeed the bottleneck on this codelet.

We can see that each page walk takes around 26 cycles on SNB2. We hence set the matching page walk (or TLB miss) bandwidth to 1/26 per cycle, though this number may change depending on on the size of the pages used, as well as the type of virtual memory indexation.

We hence created a *TLB miss* node defined as following on SNB:

1. Workload: number of *L2 TLB* misses as measured with hardware counters (*DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK*).

2. Bandwidth: defined empirically (see Figure 5.4).

$$TLB\ miss\ bw\ =\ inverse\ (measured\ TLB\ miss\ cost)\ =\ 1\ /\ 26$$

Which counters to use is microarchitecture-dependent.

The effective page walk bandwidth may vary depending on the machine and the CPU's microarchitecture. For instance, Broadwell should be able to process 2 TLB misses in parallel [114].

The TLB miss node could be refined, as the exact penalty of a TLB miss depends on how far in the memory hierarchy the page translations have to be fetched. It could then be decomposed in different nodes, each accounting for misses of different

severity. A global node could then aggregate the results of these specialized nodes in an additive manner. However, we found this degree of details to be unnecessary so far.

5.5 Handling Unsaturation

Unsaturation happens when none of the model's nodes is fully saturated for a given workload and an actual measurement. In other words, the model fails to explain the entirety of the execution time, creating a mismatch between performance projections and reality.

We will present this phenomenon, some of its potential causes, and approaches that can be used to improve the model's precision.

5.5.1 Definition and Effect of Unsaturation

Unsaturation is the degree at which the modeled system is not fully saturated (in the sense of the *system saturation* metric defined earlier). We can hence define an unsaturation metric as follows:

$$unsaturation \ = \ 1 \ - \ system \ saturation$$

Unsaturation causes the Cape model to overestimate performance as it needs to fill this gap by scaling performance up until a node does reach a saturation point (and its system saturation assumption is fulfilled).

The following issues can cause unsaturation:

1. An existing node is mismodeled. For instance, not taking into account unlamination can cause the Front-End's workload to be underestimated, and hence create system level unsaturation if the Front-End is the actual bottleneck in the considered loop.

2. An important node is missing: some hardware features may have been overlooked when building the model. For instance, retirement is ignored in our current implementation as we assume it is working at an equivalent speed as the Front-End. This assumption could cause unsaturation if it broke for a codelet.

3. Limited buffer sizes: buffers (out-of-order resources in this context) allow the hardware to absorb some latency, e.g. from memory accesses or individual instructions. Cape nodes being bandwidth-centric makes them mostly oblivious to buffer restrictions, and not account for cases where the lack of buffer entries penalizes actual performance.

The root problem for the two first cases can (and should) be fixed by correcting the wrong assumptions in existing nodes, or creating new ones as the need arises.

The third case is harder to handle as it stems from non-bandwidth-related hardware characteristics, making the usual bandwidth-centric approach less applicable. We hence used the observation that system unsaturation is typically highest when the arithmetic and memory workloads are equivalent to introduce a post-Cape-calculation.

We will describe both these approaches in this section.

5.5.2 Overlooked or Mismodeled Nodes

The primary reasons why hardware components can be overlooked (and hence not have a node of their own) are:

1. Wrongful assumptions they cannot be (or rarely ever are) bottlenecks. For instance, SNB's uop cache can make one think the Front-End should not be a problem anymore (compared to earlier Core microarchitectures), whilst the restrictions explored earlier show it is counter-intuitively still an important component.

2. They did not get in the picture in previously considered loops. Cape is in large part an evolutive model, where nodes are added as they are needed. This is the case for the TLB miss node (which was unimportant for as long as we studied loops with hardware-friendly access strides) or inter-iteration dependencies.

Components can be mismodeled when their behavior was not as simple as anticipated, and:

1. The bandwidth is incorrect: this could e.g. happen for the divider nodes when we were not addressing the variable complexity of divisions in the considered codelets.

2. The workload is misevaluated: this would happen for the Front-End node if unsaturation were not accounted for.

5.5.3 Buffer-Induced Unsaturation

The difficulty in estimating how a code will be impacted by the necessarily limited size of out-of-order buffers makes buffer-induced unsaturation the hardest type of unsaturation to address. This is partly due to DECAN transformations changing the level of stress on out-of-order resources in the execution pipeline.

Our approach here rests on the observation that out-of-order buffers tend to have the biggest impact when FP and memory instructions take similar amounts of time to process 5.5. Intuitively, this situation fosters competition for acquiring shared resources such as Reservation Station or ROB entries.

We can then build statistics for the level of interference due to buffers depending on the time taken by FP and memory nodes, and heuristically adjust the model's output based on measured precedents.

From a user's perspective, it can be thought similarly to a Fourier transform for signal processing. Instead of switching to the frequency domain, using Cape allows to see hardware and software from a bandwidth-centric view, helping perform changes on the SW or HW sides with a trivial complexity (compared to other existing options). However, unlike Fourier transforms, switching to the "bandwidth" domain induces some performance distortion which we try to dampen with our heuristic adjustment.

5.6 Related Work

Other mechanistic models such as [115, 116] target HW mechanisms to perform performance projections, though they focus on miss events to compute performance and

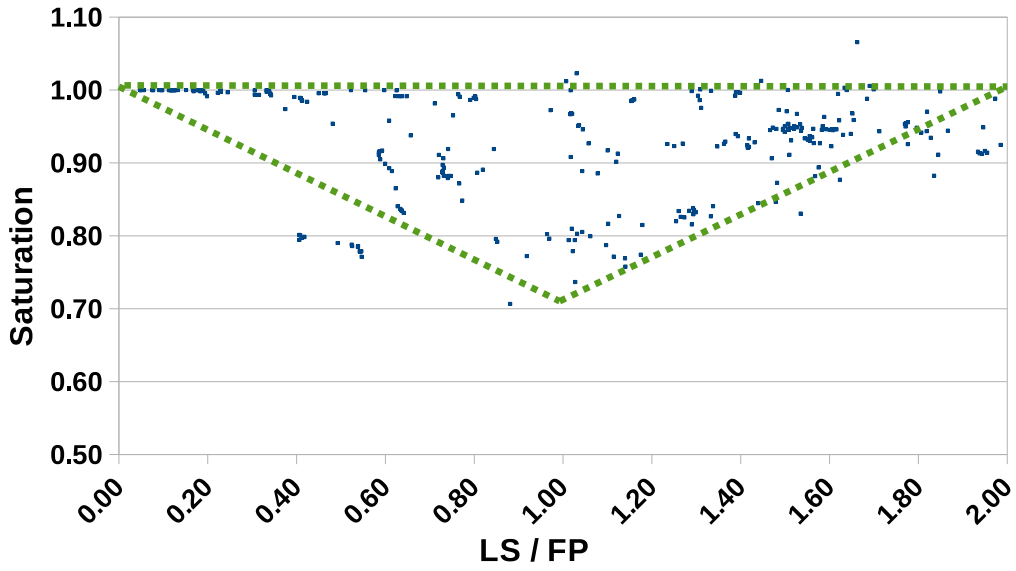


Figure 5.5: DECAN-level System Saturation

This figure shows DECAN-level saturation across all our classical NR codelets (on SNB2 and for all data sets) as a function of the ratio cycles per iteration (LS) / cycles per iteration (FP). *DECAN-level saturation is the ratio min* (FES, FP, LS) */ REF (where each variant name represents the time spent in said variant). It quantifies the degree of overlap for the Front-End, floating point and memory workloads and can be used as a base line to estimate Cape-level system saturation. Points with an LS / FP ratio over 2 do exist but were not shown to clarify the figure; their saturation values neared 1. 317 data points are shown on the graph.*

We can see saturation tends to be lower when the LS / FP ratio nears 1. The triangle with dashed lines represents the generally expected realm of saturation values for a codelet given an LS / FP ratio.

We can see outlier data points for realft_4_de in the bottom left side of the triangle, reaching a saturation of only 0.8 *despite having an LS / FP ratio of* 0.4 *(which is quite distant from the expected worst-case scenario of 1). Most points near the upper right point of the triangle represent toeplz_1_de, though their impact is lesser due to being above the* 0.9 *line. Another outlier is the point in the top right part of the graph with a saturation* exceeding *1, which should not happen. This is likely due to a measurement error, with the measured LS time exceeding that of REF.*

Other codelets with saturation typically below 0.90 *are four1_2_me, lop_13_de, mprove_8_me and rstrct_29_de. Causes behind this phenomenon will be explored in more details in Chapter 7.*

assume execution goes unimpeded outside them (in contrast to Cape's bandwidth-centric approach).

Treibig and Hager [117] propose a mechanistic performance model for memory workloads assigning constant costs for cache line transfers based on the estimated cache activity. This differs from the still codelet-specific effective bandwidths for our L2, L3 and RAM nodes and is actually more in line with what we would seek for Cape nodes. However, it is not precise enough yet for our purposes.

The Roofline model [118] evaluates the relationship between RAM bandwidth

and *operational intensity* (the number of operations performed per byte transferred from/to RAM) to offer insights on a program's maximum theoretical performance. While the model provides an admittedly limited accuracy, it is also very simple and inexpensive, aiming more at guiding the optimization process than at correctly predicting performance.

Linear regression models built with [119] provide a high speed and flexibility in terms of exploring potential designs. They also take more factors into consideration than Cape by modeling latency, buffer sizes and cache sizes explicitly. However, they are trained on a per-program basis using a cycle-accurate simulator to predetermine the relationship between key microarchitectural characteristics. Cape only needs cost-effective analyses or measurements, and is directly applied to existing microarchitectures.

Techniques such as [120] can also used to harness the precise design insights provided by simulators despite their extremely low speed. A smart selection of parameter combinations can help reduce the number of simulation runs needed to explore design spaces.

Cape also operates at a strict loop granularity, preventing compensation effects in a program's execution when validating the model.

5.7 Future Work

Explicit support for queue modeling would help not depend on ad hoc heuristic solutions to fix performance projections, be more accurate and be overall more flexible with the performance tuning knobs made available to users. Furthermore, queues have an important role in absorbing and improving the performance of the memory subsystem. Queue modeling could be very beneficial in improving memory modeling, possibly to the point where memory nodes' bandwidth may be used across codelets. This is an important challenge, but one that may significantly improve Cape modeling as a whole.

One of the strongest advantages of Cape is that users can modify both hardware features (node bandwidths), but also software ones (node capacities). Adjusting these parameters can lead to interesting performance tuning insights in terms of vectorization potential (as will be seen in Chapter 6), or other optimizations such as loop blocking: what would happen if the workload for L3 was increased, and that of RAM decreased by the same amount?

Cape could also help determine whether parallelization efforts could be worthwhile by modeling the RAM bandwidth achievable by the CPU and that by a single core separately, and exposing the leeway between the two.

Finally, Cape could be used to project performance from a given machine to another, if a list of bandwidth modifications can be supplied. For instance, what would happen on a machine with RAM that is twice faster? In a more complex fashion, projections between microarchitectures could also be done. For instance, projecting HSW performance from SNB measurements could be tried by distributing some of the dispatch workload on more ports, increasing the bandwidth of the divider to match its speedup, doubling the L1 and L2 bandwidths...

While validation data is hard to get when making arbitrary hardware modifications due to not having reference measurements to compare projections to, more validation data on software related changes (e.g. L3 blocking) could strengthen the credibility of the model in hard-to-verify areas.

5.8 Acknowledgments

All changes to the Sandy Bridge version of the Cape model were done in collaboration with David C. Wong, David J. Kuck (Intel Corporation) and William Jalby (UVSQ).

Vish Viswanathan's (Intel Corporation) help and expertise on hardware counters was key in finding the adequate hardware events for memory node capacities.

Zakaria Bendifallah, Emmanuel Oseret and Mathieu Tribalat's (UVSQ) collaboration was vital in keeping the CQA and DECAN tools in line with the project's advancement.

5.9 Conclusion

Cape is a powerful model combining simple linear micromodels called nodes to decompose loop performance and allow very fast performance projections to be done when modifying hardware or software characteristics.

We presented Cape improvements aiming at adapting it to recent Intel *Core* CPUs, and particularly Sandy Bridge. This work was a necessary step to give Cape a detailed awareness and control of the main bottlenecks in these CPUs, opening the door to works such as VP3 (see Chapter 6).

The Cape performance decomposition performed with modeling in mind can also be used for just what it is, i.e. a very detailed and hierarchized list of bottlenecks established at a relatively low cost. In this sense, Cape could be a direct competitor to PAMDA (presented in Chapter 4).

The work done to find missing nodes, improve existing ones and explain mismatches also helped refine other tools and approaches. For instance, a new basic transformation called FES was added to DECAN to estimate the impact of the Front-End, and unlamination is now compensated for in other DECAN variants aiming to preserve the Front-End workload. The effects of both unlamination and uop queue restrictions on uops from different iterations were also integrated in CQA.

Furthermore, the importance of out-of-order queues was exposed thanks to the mismatches we could not explain with regular bandwidth-centered nodes, leading us to explore projects such as Uop Flow Simulation (see Chapter 7).

Chapter 6

VP^3: A Vectorization Potential Performance Prototype

We present VP^3 [121], a new methodology and tool prototype for vectorization insight. It works on real applications and data, and offers accurate performance gain bounds. VP^3 guides expert developers toward the highest potential performance gain sections of the application and helps them avoid those with low potential. Choices of vectorization legality are left to the developer. Examples of its effectiveness are given. The ideas and insights provided may also be of use to compiler designers as well as system architects.

6.1 Introduction

High end microprocessors rely more and more on vector hardware units and instruction sets to increase their peak performance, including VIS[1] and MMX[2] (64 bits), AltiVec[3], SSE[4] and NEON[5] (128 bits), AVX[6] (256 bits) and soon AVX-512[7] (512 bits). On AVX-512, maximum DP scalar code performance is $1/8^{\text{th}}$ of the peak. Using full vector length versus scalar results in a modest increase in power with potentially important performance gains [122], leading to major energy/performance gains, so there is a strong incentive to exploit vector units as much as possible.

To that end, much effort has been spent on automatic vectorization technology. Vectorizers detect dependencies that prevent vectorization, and also try to remove harmful dependencies using loop transformations such as splitting, unroll and jam, array renaming, interchange, etc. Finally, they emit vector code, dealing with target instruction set limitations. For example, non-unit stride vector loads are not supported in SSE or AVX, so the compiler applies transformations to avoid non-unit stride accesses. As a result, vectorizers have reached a very high level of complexity. In practice, advanced analyses are limited to control compilation time, so many vectorization opportunities are not exploited [59]. Furthermore, some vectorization restructuring needs are very costly or beyond automatic tools (e.g. tracing for exact data dependence analysis), so developers must hand-modify codes. All of this makes vectorization expensive, so vectorization efforts must be spent wisely.

When automatic vectorization results are disappointing, developers need simple ways to understand their reasonable options, given the available time and skills.

[1] SPARC® Visual Instruction Set™

[2] Intel® MMX™ technology

[3] IBM® AltiVec™

[4] Intel® Streaming SIMD Extensions™

[5] ARM® NEON™

[6] Intel® Advanced Vector Extensions™

[7] Intel® Advanced Vector Extensions 512™

After profiling, their best current option is often adding SIMD directives to experiment with potential performance gains. This can force compiler vectorization by ignoring data dependencies. The generated code can reveal potential performance gains, but less robustly than VP^3, for several reasons:

1. Compilers may fail to vectorize some loops, even with directives.

2. Compiled code may crash because it is incorrect (due to directives forcing the compiler to ignore actually important dependencies).

3. The compiler does not produce the detailed quantitative information that VP^3 can.

As a developer tool, VP^3 would be used whenever a vectorizing compiler gave weak results. VP^3 would allow developers to understand their best restructuring options, together with the potential performance payoff of each.

Both SIMD directives and VP^3 suffer from the major aggravation that potential gains depend on dynamic code properties such as loop iteration count (short vectors benefit less from vectorization than long ones), operand locations (too many RAM accesses lead to minor payoff) or data alignment. Vectorization is only relevant when it can be done legally (i.e. preserving the final output) and with a satisfying speedup. VP^3 and SIMD directive insertion can both violate program semantics, and evaluating vectorization legality is very costly, so first getting an idea of potential performance gains can be an efficient way of trimming out low-incentive candidates from the list of loops to consider.

Vectorization is not an all-or-nothing activity; partial vectorization (e.g. vectorizing only the floating-point operations) can yield solid performance gains. With SIMD directives this is hard to control, whereas VP^3 fully exploits partial vectorization automatically. VP^3 has options allowing it to assume non-unit stride, gather, or scatter architectural features, and gives performance estimates even where a given compiler fails. Manual optimization choices can be delicate because they depend upon target architectures' vector lengths and instruction set characteristics. For example, the vector units of the recent Silvermont[8] and Haswell[9] processors are very different. VP^3 architecture options can be used to choose the best-performing system for a given code.

The major contribution of this chapter is VP^3, a methodology and tool prototype, which assesses potential vectorization performance gains, guides the user toward loops with potentially high benefits, and has the following properties:

1. Quantitative assessment of vectorization performance gains, in terms of the target architecture's details, taking into account dynamic constraints such as loop iteration count and operand location.

2. Single pass, no-fail operation, in contrast with the complexities of deciding where to insert SIMD directives.

3. Practicality relative to running speed (few times real time), *in vivo* analysis of real production applications and data, and at least 80% coverage of total real time.

[8] Intel® Silvermont microarchitecture

[9] 4$^{\text{th}}$ Generation Intel® Core™ processors

Section 6.2 describes VP^3 operation from a user's point of view, while Section 6.3 has examples of usage on real applications. Section 6.4 gives the general architecture/principles of the tool. Section 6.5 presents VP^3 methodology and validation of the performance gains predicted versus real measurements. Section 6.6 presents tool extensions under development.

6.2 Tool Operation

This section will present general objectives for prediction tools, as well as VP^3's output.

6.2.1 General Objectives for Prediction Tools

Performance tools usually work on the principle of informing users about the current situation inside a computation. For an existing program and data set, profilers show where the execution time is spent and performance problem summaries are developed to point to problems on the basis of compiler and run-time information. Seldom does a tool *predict* performance gains in specific areas of a source code. VP^3's capabilities are to distinguish good performance from bad according to a user-defined threshold, and to make accurate predictions about performance above the threshold. It can do this for the cache-neighborhood of any data set provided for measurement, and report that neighborhood to the user (e.g. L3 contained).

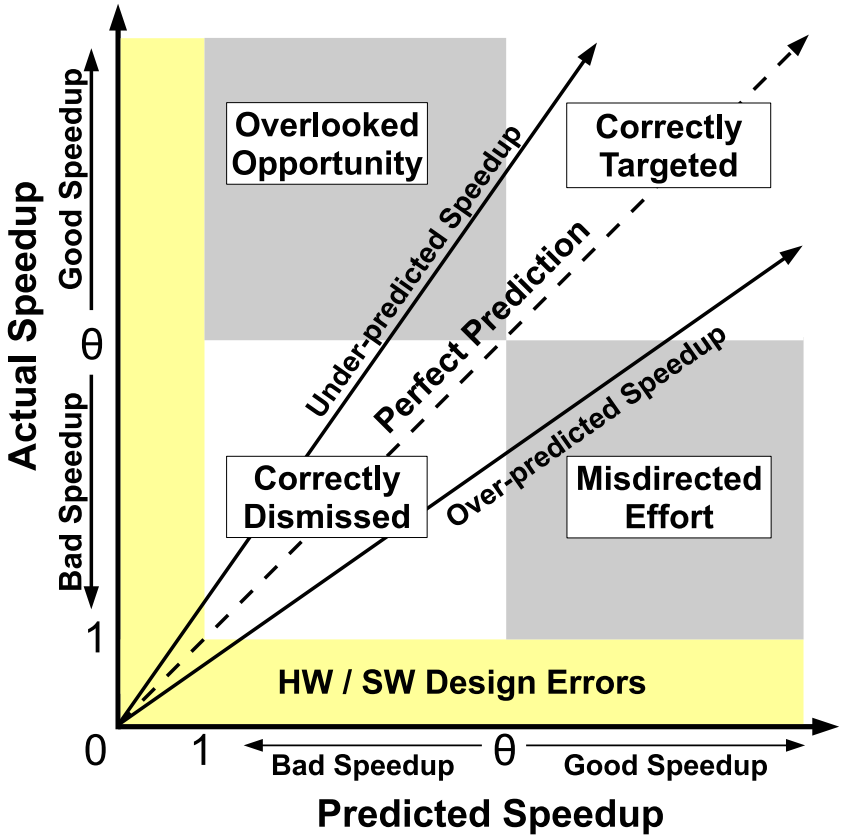


Figure 6.1: Operating Space of a Performance Prediction Tool

Fig. 6.1 shows the overall situation for any such tool, relative to performance and to the prediction quality. The axes are defined as speedup by vectorization

over scalar time and are marked to show good and bad performance. The x-axis represents VP^3 predicted speedup, and the y-axis shows actual speedup obtained by compiler or manual vectorization. The point θ marked on each axis marks a threshold performance that the tool user chooses as a worthwhile vectorization performance goal. It may depend on the maximum theoretical speedup possible, the time allowed for the vectorization work, and the skill of the SW development team. The band between 0 and 1 represents various types of performance that we call erroneous, where vectorization may lead to slowdowns. Such issues have various HW and SW sources beyond the scope of this paper.

Within the graph, labels show the tool quality regions. The dashed line with slope of 1 represents perfect predictions made by an ideal tool. The four quadrants show prediction quality. We can state 3 tool objectives, in rank order:

1. Reject candidates with performance potential below threshold θ ("correctly dismissed").

2. Quantify gains for the complementary region of potential performance improvement ("correctly targeted").

3. Carry out these two types of predictions with relatively low respective errors.

The lower left quadrant is most important because it saves developers time and effort by not having to even consider certain loops. The upper right quadrant forms the target loops, and the solid lines with slopes above and below 1 denote regions with large errors, which may also be set by user parameters. If a tool projects a sufficiently large gain, and the gain obtained is even greater, few users will be unhappy. Even if the gain is less than projected but above the threshold, the tool is useful.

The lower right quadrant of "misdirected efforts" shows the region where a tool gives a high projection but the actual gain is below the threshold. The upper left quadrant of "overlooked opportunities" shows rejected loops that are actually worth vectorizing. Both quadrants are to be avoided.

6.2.2 Tool Output

VP^3 output for each target loop is an estimated vectorization speedup gain. Such output is augmented by an analysis of the main source of performance losses, including:

- Whether the loop performance is dominated by data access (Load Store instructions) or arithmetic (Floating Point instructions) and by how much. This is useful to guide the user in optimization and in particular through partial vectorization.

- For data access bound code, non-unit stride access is reported, as is the most heavily used memory hierarchy level. More precise data on non-unit stride values can be provided at the cost of an additional run. Also, VP^3 will accept (by default) non-aligned vector loads and stores. An estimate of potential gain by alignment can also be provided.

- For Floating Point bound code, potential recurrence limitations, slow operations or code bloated by address computations or data reorganizations are reported.

This reporting can be even further refined by compiling the code with -g option enabling to establish a correlation between assembly load/store instructions and arrays in the source code.

6.3 Experimental Results

This section will present results for VP^3 on real-world industry applications.

6.3.1 Motivating Example

YALES2 [123, 124] is a numerical simulator of turbulent reactive flows using the Large Eddy Simulation method. It is a finite volume code for unstructured meshes, with an innovative 4$^{\text{th}}$ order spatial scheme for the discretization of convective and diffusive terms. It is based on the low-Mach number approximations of the Navier-Stokes equations, which solves an elliptic Poisson equation at each iteration and scales well to over 16K cores. The MPI version uses subdomain decomposition with adjustable domain size, allowing efficient cache usage.

```
do ip=1,el_grp%npair
S1  ino1 = pair2node1_v(ip)
S2  ino2 = pair2node2_v(ip)
S3  co = sym_op_v(ip)*(r1_p_v(ino2) - r1_p_v(ino1))
S4  prod_r1_v(ino1) = prod_r1_v(ino1) + co
S5  prod_r1_v(ino2) = prod_r1_v(ino2) - co
end do
```

Figure 6.2: YALES2 loop example

Consider vectorizing the YALES2 loop of Figure 6.2. Statements *S1* and *S2* obviously vectorize. *S3* is more challenging because the indirect addressing needs hardware support (vector gather instructions) which is available on the latest generation of Intel® Haswell processors. *S4* and *S5* are much more difficult because indexes `ino1` and `ino2` can take values throughout the loop execution that create dependencies within *S4* and *S5* themselves, and between them. So there are 3 levels of vectorization opportunities in Figure 6.2.

1. The easiest to vectorize are *S1*, *S2* and *S3* and the FP operations in *S3*, *S4* and *S5*. While within autovectorizer capabilities, the current vectorizers did not vectorize.

2. The LS dependencies of *S4* and *S5* due to indices `ino1` and `ino2`, can be removed by introducing coloring to ensure that `ino1` and `ino2` values are distinct. This is well beyond autovectorizer capabilities, but VP^3 can suggest, e.g. that scatter instructions would be usable if the developer introduced coloring. Forced vectorization using SIMD directives can cause the generated code to be wrong with the application producing errors or crashing.

3. To avoid all of the issues related to indirect addressing, VP^3 could advise the code developer to first use coloring to remove dependencies, and second to entirely restructure the arrays to generate stride 1 accesses, which will be then easy to vectorize. This second step is a major undertaking, requiring one to copy portions of irregular data structures into regular ones.

Solutions 2 and 3 put increasing burden on the code developer (several weeks of code rewriting). Before embarking on such efforts, developers want to know how much performance gain to expect. The main goal of VP³ is to provide such answers. However, succeeding in vectorizing is not enough, because vectorization is highly dependent upon data access location: if data is in L1, performance gains will be much larger than if operands have to be fetched from memory. In the latter case, it means that code developers will not only have to perform data restructuring but also blocking, further increasing the cost of vectorization.

6.3.2 VP³ on YALES2

Vectorizing YALES2 is a major challenge due to the indirect accesses induced by the irregular mesh structure: many of the loops have a structure very similar to the one presented in Figure 6.2. Of the 200 loops necessary to achieve 80% coverage of the total application time, about 2/3 are data access bound, the rest being FP bound. Some loop bodies are fairly complex; two contain more than 300 assembly instructions. On Sandy Bridge architectures, the lack of scatter or gather operations would be a performance killer for vectorization.

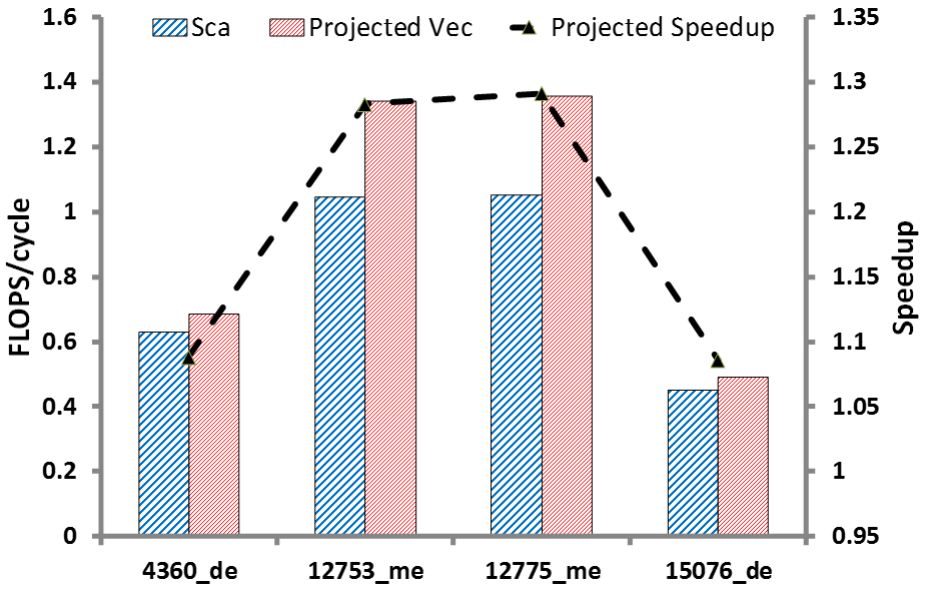


Figure 6.3: VP³ Projection Results for YALES2:
Low Prospects for Vectorization

For YALES2, the results shown in Fig. 6.3 corresponds to a Sandy Bridge target. Since Sandy Bridge does not support Scatter/Gather instructions, VP³ was set not to perform Load/Store vectorization. The performance gains remain modest (1.2 to 1.3X). Since the cost for vectorizing this code was extremely high, and VP³ showed limited performance gain potential, we did not push for this optimization for Sandy Bridge.

On Haswell, the loads could be vectorized but the overall gain would remain limited due to performance limitations of the gather instructions.

6.3.3 VP[3] on POLARIS(MD)

POLARIS(MD) (developed at CEA DSV) is a parallel code that simulates microscopic molecular systems using sophisticated interatomic potentials (including polarization effects and beyond). It includes an efficient multi-level polarizable coarse grained approach to modeling an extended chemical environment (from nanometer to micrometer scale). It is well-suited to investigate the properties of solvated protein systems and of heavy ions in complex chemical environments [125].

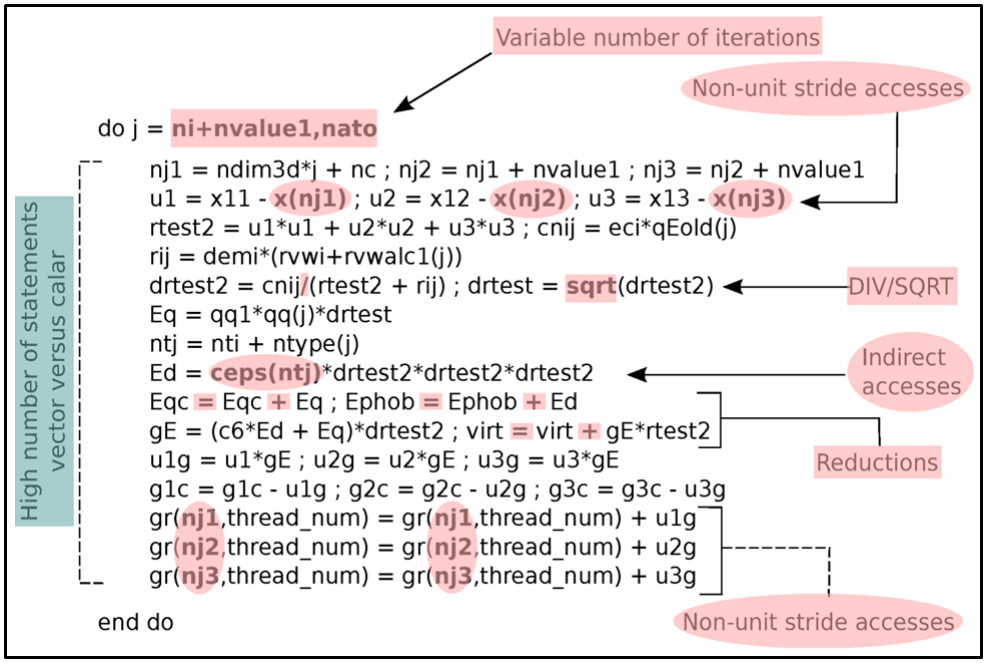


Figure 6.4: Source Code for POLARIS Loop Example

Out of all the potential performance issues, only the division and square root operations impede performance.

Disclaimer: we already evaluated this particular loop with PAMDA (in Section 4.2), but we will analyze it again from the VP[3] perspective here.

A typical loop computing interactions between pairs of atoms is shown in Fig. 6.4. This loop was originally not vectorized to preserve roundoff. The key issue was to determine what was worth vectorizing. A first VP[3] run showed that the cost of FP instructions was 4 times higher than the cost of Load Store instructions; the effort had to be focused on FP vectorization. VP[3] further revealed that partial vectorization of FP would provide a 2X performance improvement simply due to the SQRT and DIV vectorization. SIMD directives were added, full runs were made to check that round off errors did not change, and the loop got a 2X speedup. The code contained 40 similar loops (i.e. containing DIV/SQRT) for which VP[3] gave similar diagnostics. The SIMD directive was applied to all 40 loops resulting in a 1.5X speed up on the whole application.

VP[3] was run on 7 fairly large POLARIS codelets (up to 111 assembly instructions in the loop body), including the FP saturated one discussed above. Fig. 6.5 shows the scalar (Sca), projected vector (Projected Vec) and hand-vectorized (Vec) results, as well as their speedup ratio (Real Speedup).

One of them actually shows an unforeseen slowdown due to unexpectedly complex address computations, causing a 25% over-prediction. Furthermore, it is in the

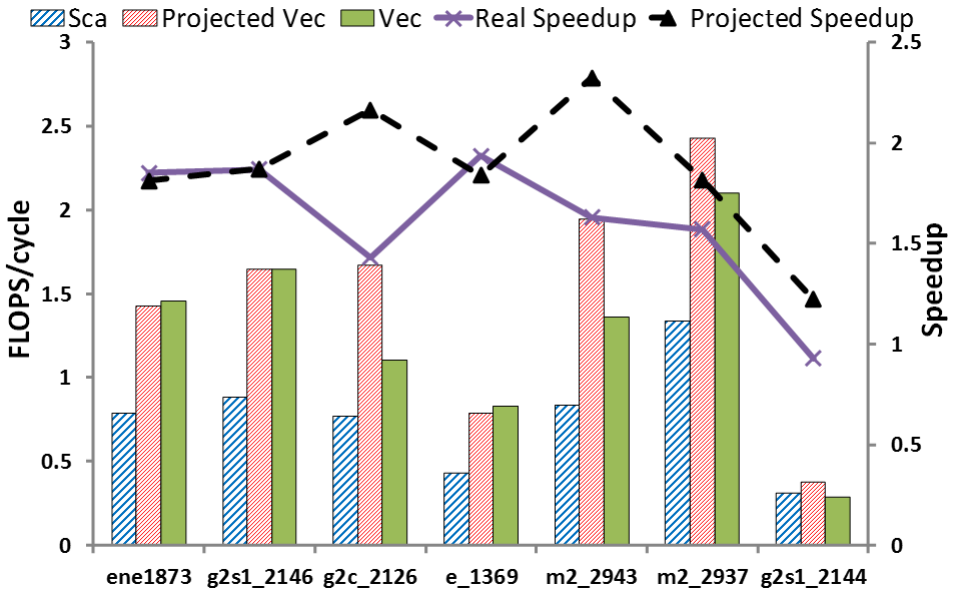


Figure 6.5: VP^3 Projection Results for POLARIS:
VP^3 vs. Measurements

lower right quadrant in Fig. 6.6: the user would be misled into vectorizing it. The 6 others are correctly estimated to highly profit from vectorization. Despite 2 being over-predicted by around 30%, the user's minimum expectation would be met.

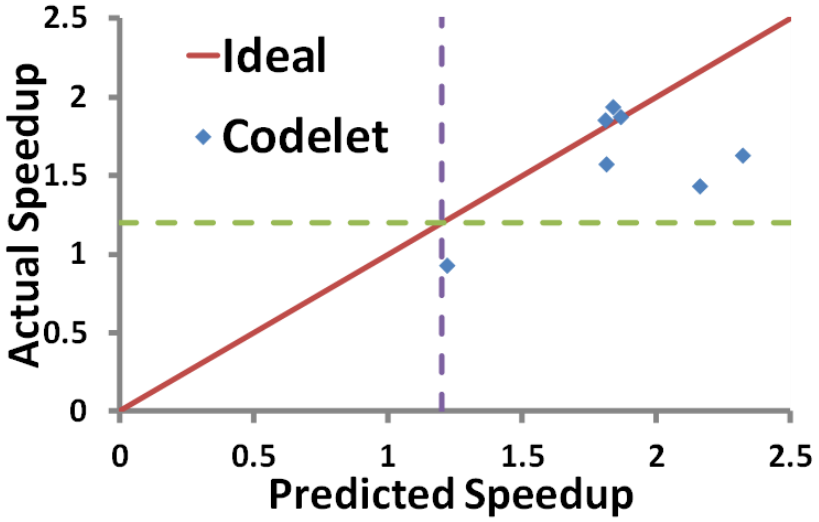


Figure 6.6: Operating Space of VP^3 on POLARIS ($\theta = 1.2$)

6.4 Tool Principles

The key principles of the method are to first isolate the performance contributions of FP (floating-point) arithmetic operations and LS (load/store) data access operations, then model the performance impact of vectorization on each of these components and finally recombine them to get the global performance.

Starting with a scalar binary loop, vector performance (in cycles/iteration) prediction proceeds in 4 phases (see Fig. 6.7):

1. FP/LS variants generation and measurement (Section 6.4.1):
   (a) Generation of FP and LS variants of the original scalar loop using DECAN [37].
   (b) Execute and measure the FP and LS variants, as well as the original binary.
2. Static projection of FP and LS vectorized performance using the generated FP/LS variants (Section 6.4.2).
3. Refinement of projected LS vectorized performance with memory traffic information gathered from scalar execution to account for memory hierarchy performance bottlenecks that limit vectorization benefits (Section 6.4.3).
4. Combine the projected FP and LS vectorized performance to get overall vectorized performance (Section 6.4.4).

The vector performance can be refined further by adding extra time for compiler-generated peel/tail loops[10] .

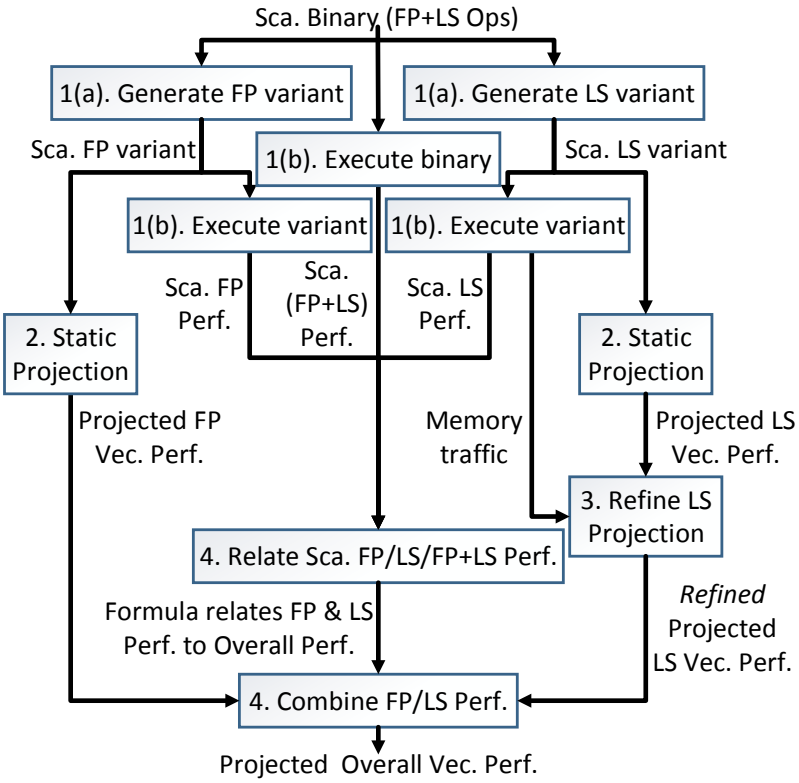


Figure 6.7: VP[3] Vec. Projection Steps

[10]These loops are generated to get proper data alignment for vectorized loop execution and deal with leftover iterations.

6.4.1 FP/LS Variants Generation and Measurement

Starting with a scalar binary loop, VP³ invokes DECAN [37] (a binary loop transformer) to generate two scalar binary variants: LS (in which all FP arithmetic instructions have been suppressed) and FP (in which all data access instructions have been replaced by `XORPS` instructions on the same register in order to avoid introducing extra dependencies).

These scalar variants, along with the original binary loop, are executed with different input data sizes to explore memory accesses from L1, L2, L3 and RAM. For each execution, in addition to measuring cycles per iteration, performance evaluation counters are used to collect memory traffic information (less than 10 counters are required to model data traffic accurately between all cache levels). These dynamic measurements are used to refine LS projection (Section 6.4.3).

6.4.2 Static Projection of FP/LS Operations

This phase is entirely built around CQA [21], a tool for analyzing binaries statically. CQA estimates a binary loop's performance (cycles/iteration) without executing it, by assuming all operands are in L1 and the loop trip count is infinite.

For each scalar FP/LS binary loop generated in Section 6.4.1, CQA can project the vectorized performance by generating a vectorized version of the corresponding scalar binary loop (called vector mockup code) and statically analyzing the vectorized loop.

To generate the vector mockup code for a loop, CQA unrolls and jams consecutive iterations to regroup the unrolled instances of scalar instructions into "packs", which become candidates for vectorization. The jamming process is made taking the dependence into consideration. CQA examines each pack of scalar instructions and tries to replace it by vector equivalents using the same register operands. There are three kinds of scalar instruction packs:

1. Simple scalar FP instructions (e.g. `ADDSS` instruction with only register operands) are simply replaced by their vector equivalent (e.g. `ADDPS`).

2. Simple scalar Load/Store instructions (e.g. `MOVSS` instruction with a memory operand) are replaced by a vector equivalent (e.g. `MOVAPS`), if CQA detects a stride-1 memory access pattern. Otherwise, for non-unit stride accesses, the instruction is not vectorized and special instructions (insert/merge) are added to pack scalar operands into a vector register.

3. Scalar FP-LS mixed instructions (e.g. `ADDSS` instruction with a memory operand) are first split into a simple FP and a simple Load/Store instruction. These instructions are then handled as described in 1 and 2 above.

Sometimes the user may want VP³ to project the vectorization performance when *only FP* operations are vectorized. This is done by generating the insert/merge instruction even when the Load/Store instruction performs stride-1 access.

6.4.3 Refinement of Static Projection of LS Operations

To refine the vector performance of LS Operations to account for performance degradation due to L2, L3, RAM and TLB activities, VP³ uses Cape [70] with inputs from CQA static analysis (Section 6.4.2) and dynamic measurements from Section 6.4.1.

Cape projects *cycles per iteration* of a loop by modeling a system consisting of N nodes. Each node abstracts a hardware subsystem using software-related metrics. A maximum operation rate called *bandwidth* is associated with each node (B_i for node i). Per loop iteration, node i will perform O_i operations. Cape uses Equation 6.1 to compute the *cycles per iteration* T of the loop. To get the vector performance

$$T = \max_{1 \le i \le N} \frac{O_i}{B_i} \tag{6.1}$$

of LS operations, the relevant nodes are:

1. L1 Load, L1 Store, Front-End, issue ports. The operation counts are obtained from CQA analysis results. The microarchitecture bandwidths are obtained from [106].

2. L2, L3, RAM. The operation/traffic counts are obtained from performance counters (Section 6.4.1). Here, we assume the data access pattern of a vectorized loop is the same as the scalar version of the loop, so we can use operations counts from scalar loop runs. The bandwidths are obtained by using scaling factors from Table 6.1 to scale up the bandwidths determined from scalar loop runs. The scalar bandwidths are empirically determined by observing the peak traffic rate when the scalar loop is executed with different input data sizes.

3. TLB. The operation count is obtained from a performance counter, multiplied by a TLB miss penalty. The bandwidth is assumed to be 1. Similar to L2/L3/RAM node, we assume the TLB behavior of a vectorized loop is the same as the scalar loop.

Table 6.1: BW Scaling Factors (BW Vector / BW Scalar)

Type of Scalar	L2	L3	RAM
Single Precision	2.04	1.63	1.25
Double Precision	1.79	1.40	1.10

In bullet 2 above, Table 6.1 was used for bandwidth scaling for SSE vector instructions. It is measured by running various kernels using different types of vector instructions (loads/stores, SP/DP). Numbers of streams were measured, and compared with equivalent kernels using scalar instructions. Table 6.1 shows that vector instructions nearly double the bandwidth of L2 but the increase drops when accessing the farthest levels (L3 and RAM).

6.4.4 Combining FP/LS Projection Results

With performance projection for FP (T_{FP}, Section 6.4.2) and LS (T_{LS}, Section 6.4.3) in cycles per iteration, VP^3 combines them to generate the overall performance estimate (T_{REF}). VP^3 assumes the relationship between T_{FP}, T_{LS} and T_{REF} can be related by a *fitting* function f such that $T_{\text{REF}} = f(T_{\text{FP}}, T_{\text{LS}})$. Therefore, to compute overall performance projection, VP^3 applies f with projected performance of FP and LS as inputs. The function is continually fitted by using *scalar* loop measurements described in Section 6.4.1 where scalar performance of FP, LS and overall are measured. VP^3 assumes the same function can be applied for both *scalar* and *vector* versions of the loop.

6.4.5 Tool Speed

High time-consuming tools can only be useful to find a few hot loops. VP^3 needs no trace and requires only several runs of the instrumented program, as follows. First, generation and execution of microbenchmarks for Table 6.1 is made once for all applications running on the same target architecture and is therefore negligible. Second, CQA passes (Section 6.4.2) and Cape (Section 6.4.3 and 6.4.4) computations are extremely fast (under a few seconds). So the cost of VP^3 is low enough to deal with a few hundred loops per application. This is often essential because many real-life applications have a very flat profile; reaching 80% coverage of total execution time may require 200 loops.

Most time consuming for VP^3 is application-specific measurements described in Section 6.4.1. A static analysis CQA pass allows the identification of unvectorized (or partially vectorized) target loops. Then running the application provides information about target loops, e.g. their weight in the program's total execution time and iteration counts. This can be done using Intel compilers, which provide max, min, and average, at modest overhead cost (20%), or using the MIL infrastructure [98] to provide more detailed information on the loop iteration count distribution at the cost of a higher overhead. Gathering traffic information via counters requires 3 further runs. Typically, the overhead associated with such collection remains under 20%. Next we execute the FP and LS DECAN variants. This can be achieved in a single run by using a rollback mechanism after each target loop; the overhead is 2X the execution time coverage. In total, 5 application runs are needed, with various slowdowns. Assuming a target coverage of 80%, the total time is around $1.2 + 3 \times 1.2 + (1 + 2 \times 0.8) = 7.4$ full application runs.

Once the data is available, it only takes a few seconds for a user to load it, choose whether to project the impact of fully or partially vectorizing and get the results.

6.5 Validation

This section will present the validation process we used and its results.

6.5.1 Methodology, Measurements and Experimental Settings

To assess the quality/limitations of VP^3, we picked a set of loops which were vectorized using a standard vectorizer. Then we forced the compiler to generate scalar versions using proper flags. These scalar versions were fed into VP^3 to generate estimates of potential vectorization gains. These estimates were compared with measurements of the vectorized loops.

The loops used for validation were extracted from Numerical Recipes (NR) [69] and previously used to validate Cape's features [70]. They were carefully selected to cover a wide range of different behavior: 1D/2D loops, 1D/2D arrays, unit and non-unit stride accesses. Both vector and scalar variants were compiled using Intel® Fortran Compiler (v12.1.3), using compiler flag "-O3" (as well as "-no-vec" for the scalar versions of the NRs, to prevent auto-vectorization). We made a few tests with compiler version v14 but (in most cases) differences were minimal.

All measurements were performed on a quad-socket Intel® Xeon E5-4640 (Sandy Bridge[11]) with Intel® Hyper-Threading Technology deactivated, on RHEL 6.3 x86-

[11] 2nd Generation Intel® Core™ processors

64 with Transparent Huge Pages . On every CPU, each of the 8 cores has dedicated: L1 instruction cache of 32KB, L1 data cache of 32KB and L2 cache of 256KB, plus access to a 20MB L3 cache shared by all cores of the chip.

6.5.2 Error Analysis

To capture the expected vectorization *benefits/effort*, VP^3 allows developers to set threshold θ (see Fig. 6.1), to focus only on loops for which potential vectorization performance gains are above threshold. As an example, we set default $\theta = 1.2$, but for a given type of app and developer skills, any value may be chosen. Fig. 6.8 summarizes, for the 16 validation codelets, the number of errors (mispredictions) and the breakdown between over predictions and under predictions. We allow a default tolerance of ± 0.02, i.e. when the user asks for $\theta = 1.2$, we guarantee a prediction for $1.18 \leq \theta \leq 1.22$. Note that the number of errors decreases linearly to 1 as the tolerance goes to 0.03. Developers could adjust the tolerance to study error sensitivity for a given set of apps, before doing their work.

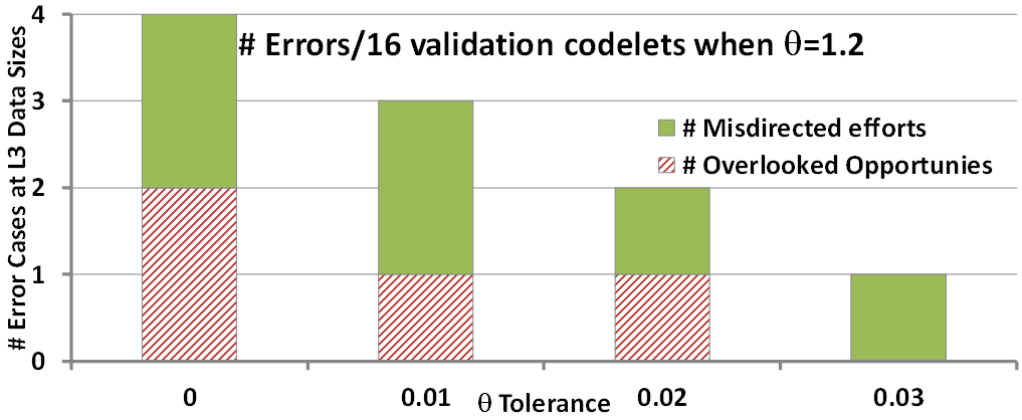


Figure 6.8: Error Cases/16 Validation Codelets vs. θ Tolerance

Our performance prediction methodology has the following sources of deviations from compiler vectorization results (cases 1. and 2.) and potential errors (3.):

1. **Dependence violation**: during the mockup vector generation in CQA, vectorized statements for which there are RAW dependencies between memory accesses could exist. Since we do not have such dependence information, our vectorization process could generate incorrect code and therefore an erroneous (conservative) prediction. Other types of dependence (WAR or WAW) are not an issue since a powerful vectorizer can easily work around them to generate vector code.

2. **Poor scalar code**: since the VP^3 vectorization process is a line by line replacement of scalar by vector instructions, it is very sensitive to the code quality of the initial scalar code. This is why we explore several unroll variants (via directives) to ensure that our process uses a high quality starting point. Note that VP^3 will not alter cache traffic. This is acceptable because apart from some corner cases, cache-optimization transformations are profitable to both scalar and vector versions. Therefore, if loop interchange is beneficial to scalar code, our starting point should be a loop interchanged scalar binary.

3. **Load-store variation**: to account for the increase in performance over scalar code for vector data access, we use a simple scaling table obtained by microbenchmarking (Tab. 6.1). For the moment, this table does not distinguish between loads and stores and the number of streams (separate array accesses). This can be improved by using more extensive benchmarks, and static analysis which will give us the number of load and store streams.

6.6 Extensions

In this paper, we have assumed that loops do not contain any branches. Recent vector instruction sets support masked operations which obtain acceptable vector efficiency for loop structures containing simple branch structures, i.e. (if... then... else). Our framework can be extended to cover such cases by using static analysis along the two potential paths. The mockup generation in CQA will assume that the two paths are generated using masked instructions.

The dynamic analysis then has the extra task of modeling the branch outcome time accounting for the branching probabilities. Assuming that traffic remains unchanged, the LS prediction can be performed in a similar manner to loops without branches. This is a valid assumption if the branch alternates targets at a fine level. Otherwise, a different method would have to be used.

Besides predicting the impact of moving from scalar to vector, VP^3 can also be used for moving from SSE to AVX, from AVX to AVX-512, or any pair of similar vector instruction sets. Some care has to be taken when moving among architectures because cache size changes may alter traffic.

6.7 Related Work

Due to the renewed importance of vector units, a lot of research effort has been recently devoted to vectorization. First, [59] analyzed in depth the major deficiencies of current vectorizers and proposed to (re)incorporate classical optimization techniques to improve automatic vectorization process. Then, to improve classical static dependence analysis, Vector Seeker [126] used on-line dynamic memory tracing analysis to obtain an accurate dependence analysis. The on-line tracing process is costly but it can detect candidate loops for vectorization which could not be analyzed satisfactorily with classic static techniques. Holewinski [127] took a similar angle but restricted their effort to regularly indexed loops. However, they provided useful information on loops for which unit-stride access could be obtained therefore improving the vectorization process. Rane [128] proposed combining static information (compiler optimization reports detailing the main reasons why the compiler failed to vectorize) with dynamic information (loop coverage, iteration count, stride, alignment) to help the user in selecting target loops for vectorization. Finally, [129] tried to vectorize directly at the assembly level, and therefore is subject to the same issues as SIMD directives.

None of the above provides estimates of vectorization performance gains. VP^3 is in fact complementary to the previous work. Running VP^3 first will allow the selection of loops of interest, and then tools like Vector Seeker can be used to discover whether or not vectorization is legal.

6.8 Conclusions

We have presented our fast and flexible prototype VP^3 for vectorization insight. VP^3 can be used by compiler and software developer to focus on loops with high vectorization gain, taking into account dynamic constraints like loop iteration count and operand location. To validate VP^3, we used 16 validation codelets extracted from [69] and [59]. We observed that VP^3 can project the vectorized performance from scalar performance within reasonable error thresholds. We also performed further investigations and have identified sources of errors, which will drive our future improvements of the tool. In addition to evaluating VP^3 with respect to difference between projected and measured vectorized performance, we also evaluated the results considering the impact to the user. To evaluate VP^3 on real world problems, we used VP^3 to analyze POLARIS and YALES2. For POLARIS, the results matched well the previous human optimization effort. We believe VP^3, as it stands, is a useful tool for those who want to improve software performance by vectorization. With the improvement we are performing, VP^3 will deliver higher quality insight to the users.

Chapter 7

Uop Flow Simulation

Different performance models can be built with very different purposes in mind. For instance, CPU architects may need very low-level cycle-accurate simulators to find and fix bugs in their design, in which case accuracy is so important that practical aspects like processing time and lightweightness become completely secondary. At the other end of the spectrum, models like CQA [21] and Cape [70] aim to provide good-enough predictions at minimum cost, both in terms of time and space. Many models exist as compromises between these extremes.

In this chapter, we will present *Uop Flow Simulation* (UFS), a technique simulating a CPU's out-of-order engine's behavior on a cycle-accurate basis and offering the following advantages:

1. Awareness of out-of-order engine limitations.
2. Very fast speed: several hundred thousand simulated cycles per second in typical cases.
3. Low memory consumption: only a few MB of RAM are required.
4. Small input files: only limited, statically extracted information needs be used for a given codelet.

We will explain key limitations of the out-of-order engine in Intel Core microarchitectures and use them to craft experiments exposing the effective sizes of out-of-order resources.

We will then describe our model in details and validate our UFS implementation both on in vitro codelets extracted from the Numerical Recipes [69] or Maleki's codelets [59], and on real-world loops from industry applications YALES2 [123] and AVBP [130], using CQA as comparison point to highlight our model's contribution.

7.1 Introduction

In a way, understanding performance is about understanding what prevents peak performance from being reached. For instance, Sandy Bridge's peak FP performance is only obtained when using the vector multiplication and addition units every cycle (respectively located behind ports 0 and 1), reaching 16 SP FP per cycle. Code that only uses additions would only be able to use one of these units, bringing the achievable peak down to 8 SP FP per cycles. If the code is furtherly not vectorized, then only 1 SP FP per cycle could be computed.

Furthermore, codelets will solicit different components more or less intensively, causing one (or some) of them to fall behind the others and drag the system's performance down by acting as figurative *bottlenecks*.

CQA and Cape shine at detecting and modeling the peak achievable performance for a given code and finding the matching bottlenecks *when their performance can*

be expressed in terms of bandwidth. However, they often do not consider *Execution Bubbles*, cycles in which such components are not used (or underused) due to microarchitectural constraints.

Hence, a key question is: how are such bubbles created, and how can they be detected, quantified and modeled?

7.1.1 On Model Accuracy

We will quantify the accuracy of our model using the *error* and *fidelity* metrics, whose respective definitions are presented in Equations 7.1 and 7.2.

$$error = |\frac{measured\ execution\ time - predicted\ execution\ time}{measured\ execution\ time}| \tag{7.1}$$

$$fidelity = 1 - error \tag{7.2}$$

7.1.2 On Buffer Sizes

Performance in superscalar out-of-order CPUs relies on queues a bit everywhere in the pipeline. Queues are a well-studied topic, partly due to how they can be applied to considerably more fields than just computer architecture.

Little's Law [131] is a central element of queuing theory, and states that

$$average\ number\ of\ waiting\ customers = arrival\ rate * average\ wait\ time.$$

In other words [132]:

$$used\ queue\ entries = bandwidth * latency$$

The consequence is that the higher the bandwidth or the latency, the bigger the queue (or *buffer*) should be not to lose performance.

A *balanced system* is a system in which buffer sizes are perfectly matched to both bandwidth and latency. In other words, resource scarcity and latency never prevent full bandwidth from being used.

While Little's formula is hard to apply directly to in-flight uops due to data dependencies, the varying nature of latency for different types of instructions and the particular relationship between the different out-of-order buffers, the point remains that buffer size and latency are two important performance factors that the largely bandwidth-centric Cape and CQA models mostly overlook. This would not be a problem if buffer sizes were always sufficiently large, but the exact number of resources needed to reach peak theoretical performance is program-dependent and has no upper bound (we will actually use this property in Section A).

Considering architectures have to work with limited resources, buffer sizes can never be large enough to cover all cases perfectly. A direct consequence is that systems can only be balanced *for a given workload.* Here, we will focus on codelets causing some *imbalance* on our target systems, and on how it is impacts CQA's fidelity.

7.1.3 On Uop Scheduling

The dispatcher (in our case, the Reservation Station) has many uops to manage, up to 6 uops to dispatch every cycle on Sandy Bridge (respectively 6 and 8 on Ivy Bridge and Haswell), and limited information to do so intelligently. The complexity of the task forces the selection process to be as simple as possible while remaining legal. Heuristics have to be used to gain as much in simplicity as needed while losing as little performance as possible. We will describe some of them, as well as their practical implications.

7.1.4 Out-of-Context Analysis

We will describe UFS as performing *out-of-context* (or *contextless*) analyses, as it does not use any contextual information to run. Such context could be static information like the presence outside the target loop of instructions that could impact its dependencies, which would not be expensive to extract. More importantly to our approach, though, UFS does not use dynamic context information such as register values, the number of loop iterations, branch traces or accessed memory addresses.

This makes UFS extremely fast as the studied loop (or the program it comes from) does not need to ever be run and typically expensive processes such as execution trace collection can be completely avoided. It also makes UFS very simple as far as cycle-accurate simulations go. However, it limits the number of phenomena that can be accurately modeled and makes issues such as memory dependencies invisible to the model.

CQA is a tool that also works contextlessly, making it a good comparison point for our model.

7.1.5 Motivating Example: Realft2_4_de

We will use codelet realft2_4_de as a motivating example to show the kind of problems raised by limited out-of-order resource sizes, as well as showcase how UFS can pinpoint the issue.

7.1.5.1 Presentation

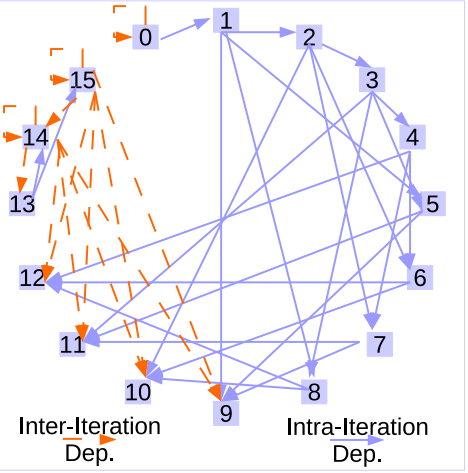
Realft2_4_de (Figure 7.1a) is a codelet from our Numerical Recipes suite, and is part of an inverse Fourier transform algorithm. It is a particularly interesting codelet as it suffers from an important CQA error even for L1 data sets. Indeed, CQA overestimates the loop's speed by 46% (see Table 7.1), making it a good case for precise study because:

1. It shows our target systems can be imbalanced for real world loops even with the very low latency provided by the L1 cache.
2. It proves said imbalance can result in significant performance drops.
3. The L1 latency and bandwidth are easy to determine statically (provided accesses do not conflict with each other, as is the case here), unlike with other cache levels or RAM. This allows us to take out the memory access variable of our analysis.
4. It reduces the list of potential culprit components involved in the phenomenon to those found in the execution core itself.

Figure 7.1: Realft2_4_de Codelet

```
0   do i=3, ishft (n, -2) + 1
1       i1 = i + i - 1
2       i2 = 1 + i1
3       i3 = np3 - i2
4       i4 = 1 + i3
5       h1r = c1 * (dat (i1) + dat (i3))
6       h1i = c1 * (dat (i2) - dat (i4))
7       h2r = -c2 * (dat (i2) + dat (i4))
8       h2i = c2 * (dat (i1) - dat (i3))
9       dat (i1) = h1r + wr * h2r - wi * h2i
10      dat (i2) = h1i + wr * h2i + wi * h2r
11      dat (i3) = h1r - wr * h2r + wi * h2i
12      dat (i4) = -h1i + wr * h2i + wi * h2r
13      wtemp = wr
14      wr = wtemp * wpr - wi * wpi + wr
15      wi = wi * wpr + wtemp * wpi + wi
16  end do
```

(a) Source Code



(b) Source Level DDG

Realft2_4_de is a codelet from our Numerical Recipes suite, and is part of an inverse Fourier transform algorithm.

Here, the DDG was made at the source level for clarity purposes. However, an assembly level version, more consistent with low-level analyses, is presented in Figure 7.2 (in conjunction with Table 7.2).

Table 7.1: Realft2_4_de: Measurements and CQA Error

Metric	REF	LS	FP	FES
Measured Duration	23.36	5.21	20.21	16.26
CQA Duration Projection	16.00	5.00	16.00	14.5
Error	31.51%	4.03%	20.83%	10.82%
Measured Resource Stalls	7.85 [RS]	0.01	7.31 [RS]	0.01

Measurements are in cycle per assembly loop iteration.

Here, we can see the error for REF is noticeably higher than for its LS, FP and REF components. Hardware counters suggest the Reservation Station (RS) may have a role in this unmodeled performance degradation.

7.1.5.2 Low-Level View

Detailed investigation using hardware counters (Table 7.1) revealed that stalls due to out-of-order resource scarcity may have had an impact on the actual performance. While not all stalls necessarily impede the execution, this was still an interesting lead directly incriminating the RS. As it is the buffer holding uops until their operands are ready, it gets particularly stressed when there are dependencies between instructions: we gave them a particular scrutiny in Figure 7.1b.

7.1.6 Alternative Motivating Example: Realft_4_de

Realft_4_de is an alternative version of the realft2_4_de codelet in which inter-iteration dependencies were removed. In practical terms, statements 13, 14 and 15 were removed from the source code presented in Figure 7.1a. Whilst it is no longer a valid Numerical Recipes codelet, the generated code is interesting and shifts the

Table 7.2: Realft2_4_de Assembly Instructions

#Id	Instruction	Type	Dispatch Ports
1	MOVAPS %XMM1, %XMM10	FP Move	5
2	LEA -0x1(, %RAX, 2), %RDI	Address Comp.	0, 5
3	MOVSXD %EDI, %R8	Address Comp.	0, 1, 5
4	MOVAPS %XMM4, %XMM11	FP Move	5
5	NEG %R8	Address Comp.	0, 1, 5
6	INC %RAX	Control	0, 1, 5
7	ADD %RCX, %R8	Address Comp.	0, 1, 5
8	MOVSD (%RSI, %RDI, 8), %XMM15	Load	2, 3
9	MOVAPS %XMM15, %XMM13	FP Move	5
10	MOVSD -0x8(%RSI, %RDI, 8), %XMM14	Load	2, 3
11	MOVSD -0x8(%RSI, %R8, 8), %XMM6	Load	2, 3
12	MOVAPS %XMM14, %XMM12	FP Move	5
13	MOVSD -0x10(%RSI, %R8, 8), %XMM7	Load	2, 3
14	ADDSD %XMM6, %XMM15	FP Add	1
15	ADDSD %XMM7, %XMM12	FP Add	1
16	SUBSD %XMM7, %XMM14	FP Sub	1
17	SUBSD %XMM6, %XMM13	FP Sub	1
18	MULSD %XMM2, %XMM15	FP Mul	0
19	MULSD %XMM3, %XMM12	FP Mul	0
20	MULSD %XMM3, %XMM14	FP Mul	0
21	MULSD %XMM15, %XMM10	FP Mul	0
22	MULSD %XMM3, %XMM13	FP Mul	0
23	MULSD %XMM14, %XMM11	FP Mul	0
24	MULSD %XMM4, %XMM15	FP Mul	0
25	MULSD %XMM1, %XMM14	FP Mul	0
26	MOVAPS %XMM12, %XMM8	FP Move	5
27	MOVAPS %XMM13, %XMM9	FP Move	5
28	MOVAPS %XMM1, %XMM6	FP Move	5
29	MOVAPS %XMM4, %XMM7	FP Move	5
30	MULSD %XMM5, %XMM6	FP Mul	0
31	ADDSD %XMM10, %XMM8	FP Add	1
32	ADDSD %XMM15, %XMM9	FP Add	1
33	MULSD %XMM0, %XMM7	FP Mul	0
34	SUBSD %XMM10, %XMM12	FP Sub	1
35	SUBSD %XMM13, %XMM15	FP Sub	1
36	SUBSD %XMM11, %XMM8	FP Sub	1
37	ADDSD %XMM14, %XMM9	FP Add	1
38	ADDSD %XMM1, %XMM6	FP Add	1
39	ADDSD %XMM11, %XMM12	FP Add	1
40	ADDSD %XMM14, %XMM15	FP Add	1
41	MOVSD %XMM8, -0x8(%RSI, %RDI, 8)	Store	(2, 3) + 4
42	MOVAPS %XMM4, %XMM8	FP Move	5
43	MULSD %XMM5, %XMM8	FP Mul	0
44	MOVSD %XMM9, (%RSI, %RDI, 8)	Store	(2, 3) + 4
45	MOVAPS %XMM1, %XMM9	FP Move	5
46	MULSD %XMM0, %XMM9	FP Mul	0
47	ADDSD %XMM4, %XMM8	FP Add	1
48	MOVAPS %XMM6, %XMM1	FP Move	5
49	MOVAPS %XMM8, %XMM4	FP Move	5
50	MOVSD %XMM12, -0x10(%RSI, %R8, 8)	Store	(2, 3) + 4
51	SUBSD %XMM7, %XMM1	FP Sub	1
52	ADDSD %XMM9, %XMM4	FP Add	1
53	MOVSD %XMM15, -0x8(%RSI, %R8, 8)	Store	(2, 3) + 4
54	CMP + JLE %RDX, %RAX	Control	5

The two instructions in line 54 will be decoded into a single macrofused *uop by the FE.*

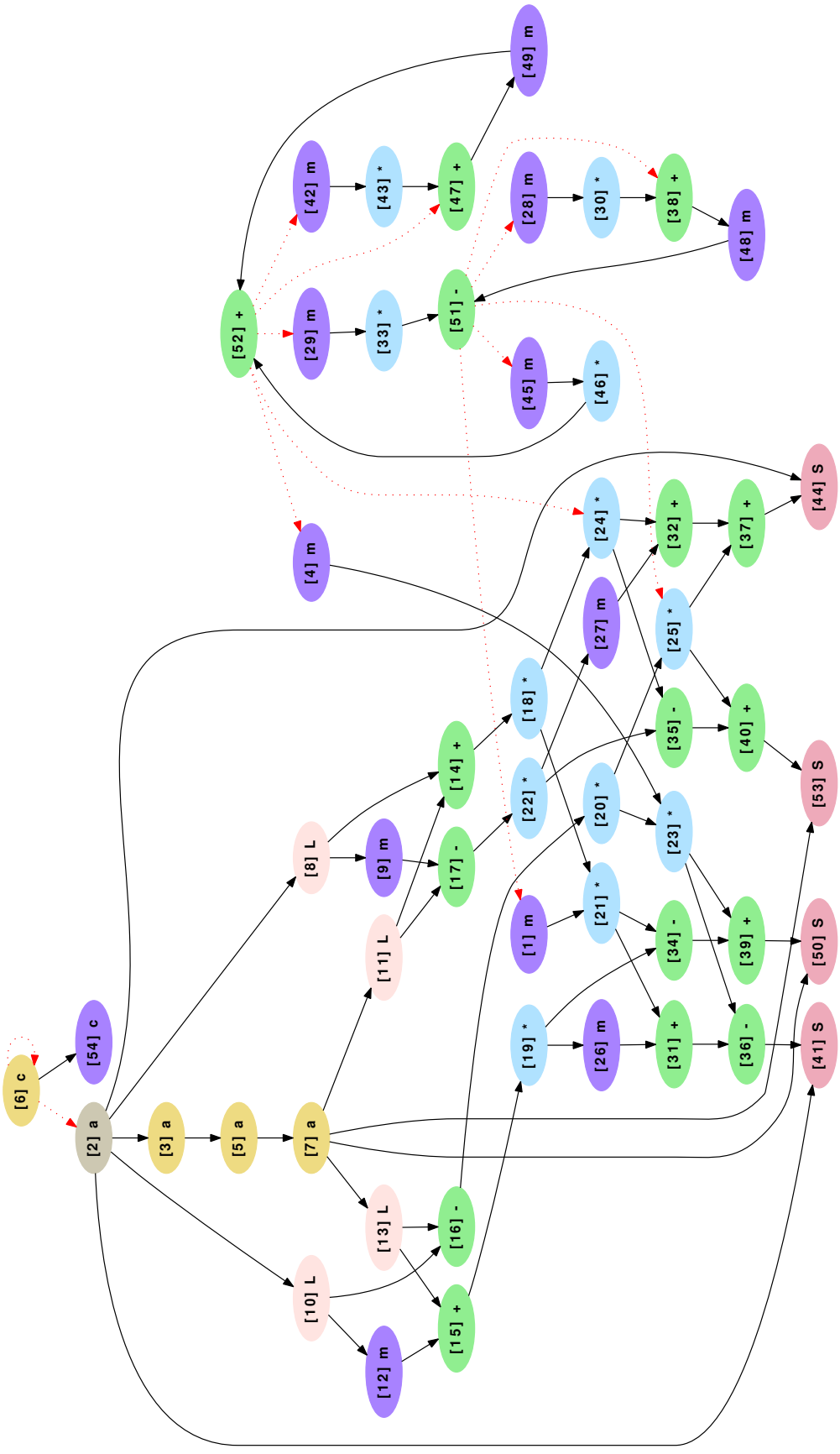


Figure 7.2: Realft2_4_de DDG

focus from inter-iterations dependencies to intra-iteration ones.

Measurement results for this codelet are presented in Table 7.3. They show that the stalls and CQA error for FP and LS are completely disconnected from those for REF.

Table 7.3: Realft_4_de: Measurements and CQA Error

Metric	REF	LS	FP	FES
Measured Duration	15.66	5.16	12.54	12.18
CQA Duration Projection	12.00	5.00	12.00	11.00
Error	23.37%	3.10%	4.31%	9.69%
Measured Resource Stalls	4.36 [RS]	0.01	3.17 [RS]	0.01

Measurements are in cycle per assembly loop iteration.

As with the original version of the code, we can see the error for REF is considerably higher than for its subcomponents. Hardware counters still flag the Reservation Station (RS) as being potentially problematic.

7.2 Understanding Out-of-Order Engine Limitations

While the out-of-order engine implemented in modern CPUs is a powerful mechanism to extract Instruction Level Parallelism, it comes with significant limitations.

7.2.1 In-Order Issue and Retirement

An important thing to keep in mind is that out-of-order execution only applies to a limited part of the execution core. The issuing and retiring of uops is still done in-order, which provides very interesting properties. For instance:

1. Transparency: speculative states are completely hidden from programs, for which the execution appears to be completely in-order.

2. A legal architectural state can always be retrieved in case of interruption or branch misprediction (e.g. mispredicted uops never impact the architectural state).

3. Resource allocation conflict avoidance: older uops never have to wait for younger uops to release resources to get issued. This prevents deadlock situations and spares the hardware from implementing costly detection and remediation mechanisms.

This hard ordering constraint can sometimes get in the way of performance, as it can prevent both the Front-End and the retirement unit from operating at full speed. In the case of retirement, it can also delay the time when certain out-of-order resources are freed and made available to newer uops.

Proposals to get around this limitation exist, but are not implemented in the studied hardware as far as we know. For instance, [133] and [134] show that performance could be indirectly gained by extremely speculative execution, where the execution of certain uops can be pursued not to advance the program's state, but to prefetch data to cache intelligently.

7.2.2 Finite Out-of-Order Resources

The execution window represents the uops being currently processed by the out-of-order engine. As uops may need to keep resources from the time they are issued until they are retired, its maximum size is constrained by the number of entries available in each buffer. The Reservation Station is a special case, as its resources are released at dispatch time and not retirement, but it can be a window size limiter too.

Having a large execution window is important, as it gives the hardware more opportunities to find instruction level parallelism (or more precisely, uop level parallelism) and improve performance. However, not only is increasing buffer sizes increasingly complex, the performance return per entry also decreases as sizes grow [135].

CPU architects consequently have to find a compromise between performance gain and resource sizes, which results in performance drops in certain codes.

7.2.3 Dispatching Heuristics

The out-of-order dispatching of uops is a complex task, for which flawless implementations would be extremely costly:

1. Only uops whose operands are ready can be handled by functional units: only those should be considered for dispatching.

2. Some uops require functional units only present behind a single execution port, making it easy to determine which one to send them to. However, this is not always the case and many uops can be processed by any of several ports, meaning extra intelligence is required to maximize performance.

3. The critical path for uop execution is hard to estimate.

Heuristics are consequently used to reduce dispatch complexity.

7.2.3.1 Port Binding

Uops being able to go to different ports gives the dispatch mechanism some leeway about which port to use, potentially reducing the number of port conflicts for uops in the RS. However, not all algorithms are adequate for this task. The ideal one would evaluate dispatch opportunities every cycle, maximizing the number of dispatched uops without actively penalizing any. For instance, if there are three uops in the RS for which respectively going to the following ports (1 | 2 | 3), (1) and (1 | 2) would be legal, an ideal dispatcher would make sure to dispatch the first uop on port 3, the second on port 1, and the third on port 2. However, this is extremely complex, as the dispatch port for the first uop has to be determined using information from uops that may be dispatched in the very same cycle.

In practice, simplifications are used. [136] strongly suggests the dispatch port is chosen depending on the number of uops assigned to each port in the RS in some Intel microarchitectures. There are different ways it could be done, but a possibility is that uops are bound to a single port when entering the RS, favoring the legal port for which the least number of uops are already assigned. With our earlier example, the first uop would be assigned to e.g. port 1 when entering the RS, the second would be assigned to port 1 (as it is the only port it can be legally dispatched to),

and the third to port 2 (due to port 1 already having 2 uops assigned to it, vs. 0 for port 2). This version is particularly easy to implement for the hardware, as once a uop is in the RS it can only ever go to a single port, and the dispatch algorithm is basic (*for each port, dispatch the oldest uop whose operands are ready and which is assigned to this port*).

Another possibility is that this is computed dynamically. In this case, the count of eligible uops at the beginning of the dispatch would be (P1: 2, P2: 2, P3: 1). The dispatcher would send the first uop on port 3 due to its being the least loaded legal port (leaving ports 1 and 2 to do work for which they cannot be avoided), the second uop to port 1 due to not having a choice, and the third to port 2 due to port 1 having been used already. However, this assumes that the dispatcher can use the information a port was already chosen to intelligently dispatch other uops in the same cycle. Also, heuristics would have to be used when all ports are equally loaded anyway, unless the count itself can be updated several times within a cycle. [137] describes a technique for dynamic port assignment inside the RS, but it uses certain heuristics too, and its description and recentness suggests it was probably not used before at least Haswell.

7.2.3.2 Pseudo FIFO Dispatching

An ideal dispatch strategy would ensure critical uops are always dispatched first. However, this would require extensive knowledge of instructions' dependencies, and hence be complex to implement.

Instead, the hardware can (and does, as far as we know) use the following heuristic: *for each port, always dispatch the oldest compatible uop whose operands are ready.* This presents several advantages:

1. Relative simplicity: no dependence graph or history of instruction dependencies needs be computed by the hardware, and only basic information is used: operand readiness, assigned port, and age of the RS entry.

2. Good resource management: as retirement is done in-order, older uops waiting for execution prevent all younger uops from retiring and releasing their resources. This strategy can hence reduce the amount of time for which each resource entry is allocated, and make better use of existing buffers.

While it often works well, there are cases in which it is not optimal. We wrote a pair of codelets ("rs_pb" and "rs_fix") exposing the issue: their assembly code can be found in Table 7.4, and results can be seen in Table 7.5.

7.2.4 Inter-iteration Dependencies

Inter-iteration dependencies are dependencies that are carried throughout the entire loop call. While uops from the same dependency chain have to be executed serially, out-of-order mechanisms can allow independent uops to be dispatched in parallel and absorb the induced latency.

However, if the inter-iteration dependency chain is long enough, it can cause latencies nearly impossible to absorb even with very large buffers and a perfect dispatcher: they impose a *de facto* upper bound on loop performance.

Figure 7.3 shows 3 examples of problematic inter-iteration dependencies with varying degrees of complexity.

Table 7.4: Exposing Pseudo FIFO limitations: Assembly Code for "rs_pb"

#Line	Instruction	Purpose
1	XORPS %xmm0, %xmm0	Resets dependency chain
2	MULPS %xmm0, %xmm0	Starts dependency chain
3	ADDPS %xmm0, %xmm0	Extends dependency chain
...	ADDPS %xmm0, %xmm0	Extends dependency chain
14	ADDPS %xmm0, %xmm0	Extends dependency chain
15	MULPS %xmm0, %xmm1	Dependent operation
16	MULPS %xmm0, %xmm2	Dependent operation
17	MULPS %xmm0, %xmm3	Dependent operation
...	MULPS %xmm0, %xmm[1+(...-15)%15]	Dependent operation
68	MULPS %xmm0, %xmm9	Dependent operation
69	ADD $1, %eax	Iteration count
70	SUB $2, %rdi	Loop control
71	JGE .LOOP	Loop control

The only difference in "rs_fix" is that the instruction on line 2 is an ADDPS *instead of a* MULPS*. As FP additions and multiplications go on different ports, it prevents uops from instruction #2 from competing with instructions labeled as* dependent operations *to get dispatched.*

Table 7.5: Exposing Pseudo FIFO limitations: Experimental Results

Metric	rs_pb	rs_fix
Measured Cycles	95.11	54.82
CQA Cycles	55	54
Measured RS Stalls	77.53	37.29

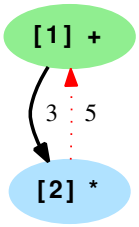
Results are normalized by the number of iterations.

In both versions of the codelet, the dependent operation *uops fill up the RS as the dependency chain gets executed, preventing the next iteration's uops from getting issued and considered for dispatching. As instruction #14 is executed, dependent instructions can finally get dispatched and make room for the next iteration's dependency chain.*

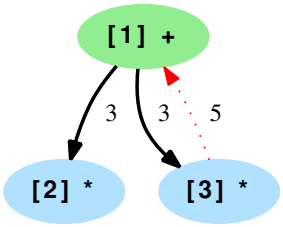
In rs_pb, instruction/uop #3 is only dispatched after all MULPS *from the previous iteration: as all their operands are ready, the older uops take priority even though they are not in the critical path (pseudo FIFO). This results in additions getting stuck in the RS, waiting for the instruction #3's execution to progress. Multiplications and additions are consequently never executed in parallel, causing a huge performance loss not foreseen by CQA. The very high number of RS stalls is a consequence of this phenomenon.*

In rs_fix, instruction #3 is now an addition: it can be dispatched as soon as it gets in the RS as older uops do not compete for the same execution port. It unlocks the execution of the rest of the ADDPS *uops in the iteration's dependency chain, allowing full parallelism of the multiplications from iteration N with the additions of iteration $N+1$, and reaching the performance peak predicted by CQA.*

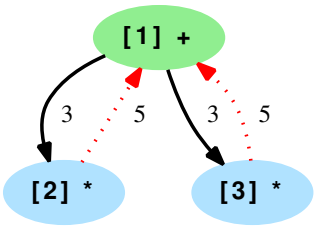
Figure 7.3: Inter-Iteration Dependency Cases



(a) Simple Case



(b) Recoverable Dispatch Conflict



(c) Irrecoverable Conflict

Nodes represent instructions, and arrows dependencies between them. Arrow labels represent the weight of the dependency in cycles (and matches the duration of the instruction from which it originates). Red arrows represent dependencies crossing the boundary of an iteration. Cycles in such graphs are what we call inter-iteration dependencies. For instance, in 7.3a, instruction 2 depends on the instance of instruction 1 from the same iteration, while instruction 1 depends on the instance of instruction 2 from the previous one. There is a cycle between 1 and 2 with a summed weight of $5 + 3 = 8$ *cycles: each loop iteration will last at least 8 cycles, regardless of any other hardware specification (out-of-order buffer sizes, number of execution ports, etc.).*

Dependency chains like the one in 7.3a can be detected and accounted statically (which CQA does). However, the one in 7.3b is more tricky: dependency-aware hardware would be able to maintain an upper performance bound of 8 cycles per iteration by prioritizing instructions from the inter-iteration chain (i.e. instructions 1 and 3 in this case), but real-world dispatch algorithms do not take this into account and instances of instruction 2 will be prioritized over instances of instruction 3 from the same iteration due to being older. Assuming only one multiplication can be dispatched in a given cycle, a 1-cycle delay will appear on top of the chain's original 8 cycles due to this dispatch imperfection, bringing the upper bound to 9 cycles per iteration instead.

In 7.3c, there is nothing the dispatcher can do to prevent this extra delay: the order in which instructions 2 and 3 are dispatched is now irrelevant. Only having at least 2 multipliers could eliminate the extra delay.

7.3 Input Resource Sizes

A key aspect to building our UFS model is getting inputs faithful to how the target microarchitectures behave in the real-world. This applies to e.g. instruction latencies (for which we rely on [107]), but also to other characteristics such as their effective out-of-order resource sizes.

While many of the theoretical sizes for the resources in question are publicly

available, they may differ from the sizes experienced in practice [61].

We wrote experiments to measure practical resource sizes and use the results as default input for UFS. A summary of our results can be found in Table 7.6 while the details and methodology were placed in Appendix A.

Gaps between on-paper figures and our results are likely due to microarchitectural constraints on resource allocation or deallocation we are not aware of, the impact of which we aim to alleviate by using these adjusted values.

Table 7.6: Experimental Resource Quantifying Summary

Uarch	SNB	IVB	HSW
BB	48	48	48
LB	64	64	72
FP PRF	112	113	138
Integer PRF	128	130	144
Overall PRF	141	165	177
ROB	165	168	192
RS	48	51	51
SB	36	36	42
ROB Microfusion	No	Yes	Yes
RS Microfusion	No	No	No

We found the practical sizes of the PRFs, the ROB and the RS to be different from the official ones in all three microarchitectures. Such differences can be explained by e.g. the mode the processor is working in (64-bit mode exposes more named registers than the 32-bit one), the number of pipeline stages involved in allocating or releasing resources, or by technical limitations unknown to us.

7.4 Pipeline Model

We will present the UFS models we built aiming at Intel Big Core microarchitectures in this section.

7.4.1 Principles

The purpose of the model is to account for limits of the out-of-order engine not taken into account in CQA using a limited cycle-accurate simulation, while still operating with very limited amounts of information on target loops. The semantics of instructions is completely disregarded; only the flow of uops is being computed, estimating the speed at which uops may travel through the pipeline.

7.4.1.1 Cycle-Accurate Simulation

Detailed interactions between uops and the pipeline are taken into account are simulated. For instance, the simulation keeps track of in-flight uops and the number of available resources, constraining the flow of simulated uops as a real system would. Dispatching constraints and heuristics are also implemented, allowing for a realistic estimation of port load in complex loops.

7.4.1.2 Limited Input

The simulator uses two types of inputs:

1. Loop information: as the model is only tracking the flow of uops and not their semantic purpose, register and memory values are not needed. It uses only basic information obtainable from static analysis, such as the type, register operands and outputs for each instruction in the studied loop. It also uses Agner's instruction tables [107] as reference for instruction dispatch port(s) and latency. Just as with CQA, loop inputs are generated using the MAQAO [138] framework. An example input is provided in Table 7.7.

2. Microarchitecture information: simple parameters such as the size of each out-of-order resource or the Front-End and retirement uop bandwidth are needed. Default values (see 7.3) can be provided for each target microarchitecture. All the studied microarchitectures have an issue and retire bandwidth of 4 uops per cycle. A few behaviors are also microarchitecture specific, such as the status of microfused uops in the ROB.

Table 7.7: Partial UFS Loop Input Example (Realft2_4_de)

#Insn	Nb_FE	Type	Input	Output	Latency	Ports
1	1	compute	XMM1	XMM10	1	P5
2	1	compute	RAX	RDI	1	P1, P5
...						
8	1	load	RSI, RDI	XMM15	4	P2, P3
...						
53	1	store_addr	RSI, R8		1	P2, P3
		store	XMM15		3	P4
54	1	branch	RDX, RAX	test	1	P5

*The specifics of what an instruction exactly does is irrelevant for UFS. Instead, only characteristics such as the number of uops it takes in the Front-End (*Nb_FE*), input/output registers and latency are important.*

The type *category allows us to determine what type of resource the uop will need to get issued (e.g. branch buffer entry for* branch *uops), coupled with the* output *field: for instance, instruction 1 is going to need a vector register as it produces an XMM value, while instruction 2 will need an integer one.*

The ports *column provides a list of ports compatible with the uop.*

In some cases, a single instruction gets split in several uops. As each of them can potentially have different attributes, differences between them need to be described explicitly. This is the case for instruction 53.

Complementary attributes are sometimes needed, e.g. in case of division uops, as they are going to use special resources exclusively for variable amounts of time.

The number of loop iterations to simulate can also be specified, with a default value of 1000.

The limited information needed to run UFS makes it possible to analyze loops outside their execution context, and hence gain very important amounts of time by not needing to simulate or execute uninteresting parts of the code to obtain legal register or/and memory states.

7.4.2 Engine

UFS simulates as many cycles as needed for *last uop of the Nth iteration to retire*, with N being the number of iterations to simulate. The different simulation steps for a cycle are as follows (in order of simulation):

1. *Update* [no order]: flags uops as being *executed* L cycles after they were dispatched, with L being their attributed latency. It is the equivalent of uop outputs being written back to the RS, allowing dependent uops to get dispatched.

2. *Retire* [in-order]: removes *executed* uops from the pipeline and releases their attributed resources. In a real microarchitecture, this is the step at which *executed* uops' outputs would be committed to the architectural state.

3. *Dispatch* [out-of-order]: removes uops from the RS when all the uops they depend on were properly *executed* and a compatible execution port is available.

4. *Issue* [in-order]: inserts new uops in the ROB (and RS if need be) and allocates all needed resources, but only if the latter are available.

The *current cycle count* (i.e. number of cycles simulated so far) is maintained and updated every cycle.

Figure 7.4 presents an overview of the model. We will describe details for each simulated component in the coming sections.

7.4.3 Simplified Front-End

As UFS targets loops, we can safely assume that all the uops sent to the *uop queue* came from the *uop cache*, hence ignoring the legacy decode pipeline and its limitations and providing a constant uop bandwidth of 4 per cycle. While the uop cache has limits of its own (e.g. it cannot generate more than 32B worth of uops in a cycle), we decided to ignore them as we could not find real world cases where they got in the picture. This is partly due to compilers being able to avoid dangerous situations by padding the code.

We also assume the branch predictor is perfect and never makes mistakes, meaning we do not need to simulate any rollback mechanisms. This is a decently safe assumption for the loops we study due to their high numbers of iterations, but limits the applicability of UFS for loops whose iteration counts are both small and unpredictable.

We consequently model Front-End performance in a simplified way:

1. For SNB / IVB: 4 uops can be generated every cycle, except a limitation of the uop queue prevents uops from different iterations from being sent to the RAT in the same cycle. For instance, if a loop has 10 uops, the uop queue will send 4 uops in the first two cycles, but only 2 in the third.

2. For HSW: 4 uops can be generated every cycle: the limit experienced in SNB and IVB was apparently lifted.

In some cases, the uop queue has to unfuse microfused uops (or *unlaminated*) before being able to send them to the RAT [110], causing more issue bandwidth to be consumed (and sometimes, also more out-of-order resources). We take common cases into account using the following rules:

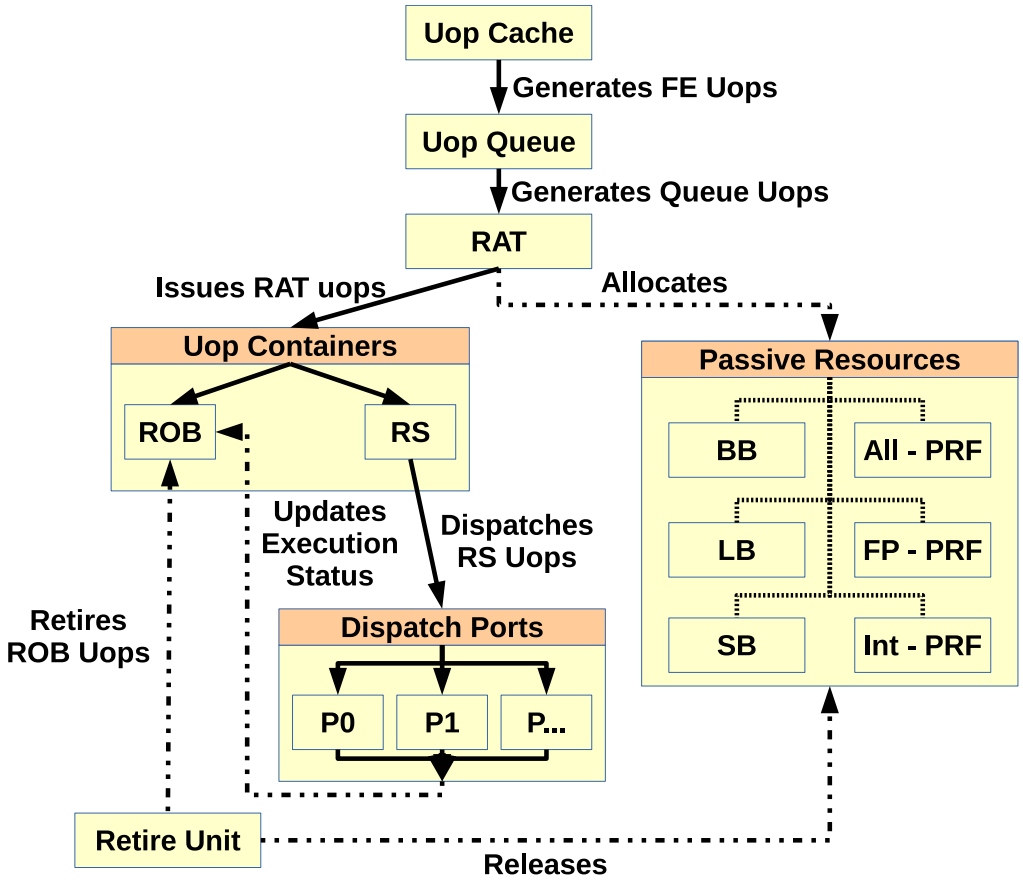


Figure 7.4: UFS Uop Flow Chart

Several types of uops are used in the pipeline. For instance, Front-End uops (FE uops) are different from Queue uops or ROB uops. Generally, the closer to the Front-End, the closer to the original instruction the uop is. As different components can split uops when processing them, Back-End uops may contain less of the original information and semantic than their earlier counterparts. In other words, more Back-End uops than FE uops may be needed to describe the same instruction. Such transformations will be detailed in the matching component's modeling description.

The ROB and the RS are both resources and uop containers. Other resources have a more passive role and do not describe uops, acting instead as mere dependencies.

1. For SNB / IVB, unlaminate when the number of register inputs for the whole instruction is greater than 2.

2. For HSW, unlaminate when the number of register inputs and outputs *for an AVX instruction* is greater than 4. This rule was obtained empirically.

The resulting model is described in Algorithm 1.

7.4.4 Resource Allocation Table (RAT)

The simulated RAT is in charge of *issuing* uops from the Uop Queue to the ROB and the RS, as well as allocating the resources necessary to their proper execution and binding them to specific ports.

It does not have any bandwidth limit other than that induced by the Uop Queue's output.

```
/* Prevents uop queue starvation                                    */
/* (assuming perfect uop cache)                                     */
fill_queue = FE_bw - nb_uops (uop queue);
while fill_queue > 0 do
    insert (new uop cache uop, uop queue);
    if has_to_be_unlaminated (youngest_queue_uop) then
        unlaminate (youngest_queue_uop);
    end
    fill_queue –;
end
/* Adapts uop queue behavior depending                              */
/* on the target microarchitecture                                  */
if target_uarch in {SNB, IVB} then
    issuable_uops = min (FE_bw, nb queue uops before next iteration);
else if target_uarch == HSW then
    issuable_uops = FE_bw;
/* Tries to get uops issued (by the RAT)                            */
while issuable_uops > 0 do
    if issue (oldest_queue_uop) == SUCCESS then
        issuable_uops –;
    else
        break;
    end
end
```

Algorithm 1: Simplified Front-End Algorithm

7.4.4.1 Resource Allocation

Typical resource allocation for *queue uops* (i.e. uops as sent by the uop queue) is described in Table 7.8.

In regular cases, a simple table look-up is enough to find resource attribution. However, it gets more complex when an instruction is decomposed into more than a single uop. In this case, all the resources needed at the instruction level will be allocated when the first uop reaches the Back-End. For instance, stores are decomposed into a *store address* and a *store data* uops: in this case, a Store Buffer entry will be reserved as soon as the *store address* uop is issued, and the second uop will be assumed to use the same entry. However, individual uop resources (ROB or RS entry) will still be allocated at the uop granularity.

It is important to note that if any resource is missing for the uop being currently considered, the RAT will stall and not issue any other uop until resources for the first one are first made available. This is commonly referred to as a *resource stall*.

Algorithm 2 summarizes how we model the *issue* mechanism.

7.4.4.2 Port Binding

Available information about dispatch algorithms in recent Intel microprocessors is rare and limited. We decided to bind uops to single ports in the RAT, sparing the RS from having to do a complex cycle-per-cycle evaluation of dispatch opportunities. Smarter strategies could be used, but we preferred to keep our simulation rules as

Table 7.8: Needed Resources for Queue Uop Types and Outputs (SNB)

Uop Characteristics		Needed Resources							
Type	**Output**	**BB**	**LB**	**All-P**	**FP-P**	**Int-P**	**ROB**	**RS**	**SB**
Branch		1					1	1	
Branch	GPR, test	1		1		1	1	1	
Compute	GPR			1		1	1	1	
Compute	Vector Reg.			1	1		1	1	
Fused compute	GPR		1	1		1	2*	2	
Fused compute	Vector Reg.		1	1	1		2*	2	
Unlam. 1st uop	GPR		1	1		1	1	1	
Unlam. 1st uop	Vector Reg.		1	1	1		1	1	
Unlam. 2nd uop	Any						1	1	
Secondary uop	Any						1	1	
Load	GPR		1	1		1	1	1	
Load	Vector Reg.		1	1	1		1	1	
Nop							1		
Nop	GPR			1*		1*	1		
Nop	Vector Reg.			1*	1*		1		
Store_addr							1	1	1
Store							1	1	
Fused store							2*	2	1

This table presents the resources needed to issue *uops with given characteristics (i.e. send them to the Back-End) in UFS modeling.* GPR *stands for "General Purpose Register", while* Vector Reg. *refers to vector registers, a.k.a. XMM, YMM or ZMM registers.* All-P, FP-P *and* Int-P *respectively refer to the All, FP and Integer PRFs.*

For instance, a load *uop with a vector register output will need entries in the LB, the All and FP PRFs, the ROB and the RS.*

Nop*-typed uops do not need to be dispatched as they are taken care of directly at issue time. This applies to regular* NOP *instructions, which do not have any input or output. However, other instructions are handled similarly, such as* XORPS *instructions applied to identical registers: the hardware can recognize such cases as always generating null values, making dispatching the uop unnecessary. A* XORPS *uop can hence fall in either the* compute *(if it operates on different registers), either the* nop *one (if it does not), changing the way the simulation will later treat it.*

Macrofused uops are a special case in which consecutive flag-modifying and conditional branch instructions are represented by a single uop. While other conditions apply in the hardware [139], we assume macrofusion to always be successful, and the resulting uops to be in the branch *category with a* GPR, test *output.*

In case of unlaminated uops (i.e. uops that were fused in the FE, but were split in the uop queue), we assume the output register to be allocated when issuing the first RAT uop, and that the second uop consequently only needs RS and ROB slots. The same rationale is applied to multi-uop instructions, for which only the first uop will be given physical registers.

Values with stars (*) are decremented for IVB and HSW, as the hardware seems to make better use of physical registers in zero-idiom cases [61].

simple as reasonably possible.

The simulated RAT keeps track of the number of in-flight uops assigned to each port, and assigns any queue uop with several port options to the least loaded one.

```
input  : target /*a uop from the uop queue*/
output: success status
if enough resources are available for target then
    allocate needed resources to target;
    port binding (target);
    insert target in ROB and RS;
    remove target from uop queue;
    return SUCCESS;
end
return FAILURE;
```

Algorithm 2: Issue Algorithm

In case of equality, the port with the lowest digit is assigned (which creates a slight bias towards low-digit ports).

This process is repeated on a per-uop basis, i.e. the simulated RAT uses knowledge generated by issuing younger uops in the same cycle, rather than using counts only updated once a cycle, which may in turn be optimistic. It is described in Algorithm 3.

Arbitrary numbers of ports can be activated as their use is regulated by the loop input file anyway (see 7.4.1.2). New microarchitectures with more (or fewer) ports could be simulated by tweaking input files' uop port attribution scheme to match the target's.

```
input  : target /*a uop being issued*/
Least_Loaded_Port = -1;
Least_Loaded_Port_Load = +∞;
foreach Eligible Port P for target in ascending digit order do
    if in_flight_uop_nb (P) < Least_Loaded_Port_Load then
        Least_Loaded_Port = P;
        Least_Loaded_Port_Load = in_flight_uop_nb (P);
    end
end
make Least_Loaded_Port the only eligible one for target;
```

Algorithm 3: Port Binding Algorithm

7.4.5 Out-of-Order Flow

Uops may progress in an out-of-order fashion from the moment they arrive in the Reservation Station until they are completed.

7.4.5.1 Reservation Station (Uop Scheduler)

The Reservation Station holds uops until their operands are ready, and the needed port and functional unit are available. When arriving in the RS, queue uops that are still microfused get split in two (see Section A.8.2), simplifying the dispatch mechanism.

The dispatch is done prioritizing older uops, following Algorithm 4.

```
for P = 0; P <= Nb Ports; P ++ do
    foreach RS uop U, in descending order of age do
        if U is bound to port P and
           U's operands are ready and
           port P's functional unit for U is available
         then
            dispatch uop U on port P;
            remove uop U from the RS;
            break;
        end
    end
end
```

Algorithm 4: Dispatch Algorithm

7.4.5.2 Port and Functional Unit Modeling

Ports act as gateways to the functional units they manage. They are modeled as all being completely identical, and being able to process any uop sent to them by the RS. Functional units are not modeled distinctly, and constraints over them are modeled inside their respective port instead. Several rules are applied to match realistic settings:

1. A port can only process a single uop per cycle (enforced by dispatch algorithm).

2. Uops can be flagged as needing exclusive use of certain functional units for several cycles. For instance, division uops will make exclusive use of the divider unit for (potentially) dozens of cycles. A port processing such a uop will flag itself as not being able to handle other uops needing this particular unit for the specified duration. The same mechanism is also used for 256-bit memory operations on SNB and IVB. A port with busy functional units can still service uops not needing them.

3. While the port itself does not check whether it should legally be able to process a given uop, the RS verifies this a priori, preventing such situations in the first place.

7.4.5.3 Uops' Execution Status Modeling

ROB uops have a time stamp field used to mark their status, and holding the cycle count at which they will be fully executed. By convention, the default value for newly issued uops is -1: a uop's output is available if *current cycle count* $\geq$ *the uop's execution time stamp* > -1.

Updating ROB uops' execution time stamp is typically done at dispatch time: as we deal with constant latencies, we can know in advance on what cycle the uop's output is going to be ready (*current cycle count* + *uop latency*).

In the case of typical *nop*-typed instructions (such as *NOP* and zero-idiom instructions like *XOR %some_reg, %same_reg*), their time stamp is directly populated with a correct value at issue time, reflecting the RAT being able to process them completely in the studied microarchitectures. As they also have 0 cycle of latency, their stamp is simply set to *current cycle count*.

However, we found an extra simulation step to be necessary to handle *zero-latency register moves* (implemented in IVB and HSW), which are *nop*-typed and are entirely handled at issue time too. Contrary to *NOP*s or zero-idioms, register moves have register inputs, the availability of which is not necessarily established yet when the move uop is issued. We tackle this issue by inserting such uops with a negative time stamp *if* their input operand's availability is not known yet, and letting a new "uop status update" simulation step update them when it is. This step is performed at the very beginning of every cycle, ensuring zero-latency moves do not stall retirement or dispatch.

7.4.6 Retirement

The retirement unit removes uops from the ROB, releases their resources and makes them available for futurely issued uops. Not all the following rules for modeling retirement are documented or strictly necessary, so we will explain our reasoning and degree of certainty as well:

1. (Needed) Retirement is done in-order: no uop can be retired if an older uop still exists in the ROB. This is necessary to be able to handle precise exceptions and rollback to a legal state.

2. (Certain) Retirement peak bandwidth is at least equal to that of the Front-End. Otherwise, retirement would become a bottleneck when non-unlaminated uops are issued, which we empirically did not find to be the case. We implemented this rule by setting the default retirement bandwidth to be the same as the FE's (4 uops per cycle), and by allowing ROB uops that were still microfused in the RAT to only count as a single uop in this regard.

3. (Assumption) Resources released in a given cycle cannot be reused in the same cycle. Our understanding is that it would be extremely complex to implement a solution allowing this, with very little performance to be gained (potentially increasing each resource's effective size by a maximum of 4). Note: we apply the same reasoning to the RS, even though its entries are freed at dispatch time instead of at retirement.

4. (Assumption) All resources allocated at the instruction level at issue time are released when retiring the last uop for the instruction in question. (This is not *certain* because the resource allocation scheme we use is an extrapolation to begin with.)

In the case of register-writing uops, real hardware retirement cannot free the very physical registers that were attributed at issue time, but rather the ones formerly used to hold the matching named registers' architectural state. Strictly speaking, this is contradicting the simplified modeling described above; however it is irrelevant for our purposes as a) we do not model individual registers, but only the number of used ones and b) we use the numbers of registers *available for speculation* as simulation input.

7.4.7 Things Not Modeled

Many aspects of the target microarchitectures' behavior are not simulated. Some of them are inherently so due to our approach, such as:

1. Cache and RAM behavior: accurate prediction of cache behavior would require simulating the program's semantics to identify the accessed cache lines or/and generating a dynamic trace of memory accesses, either of which would be extremely time consuming *and* barring the out-of-context analysis doable with UFS.

2. Read after write (RAW) memory accesses and conflicts: while some of them can be detected statically (which would be compatible with our limited input objective), finding them all would require using the same processes as for cache behavior. Furthermore, as compilers can typically preventively fix statically detectable occurrences anyway (especially in innermost loops), the value of partial coverage of RAW hazards would be questionable for the HPC loops we study.

3. Branch mispredictions: as with memory accesses, extensive dynamic information is required for the detailed simulation of branch mispredictions. Hence, we do not model branch predictors or the pipeline flushes needed when a mispredict (or exception) occurs. This restricts our ability to model short or branchy loops accurately.

Others are implementation choices, more subject to change:

1. The Load Matrix (LM) (see Appendix B): a little-documented out-of-order resource that keeps track of load uops from the moment they are issued until they are completed [140]. It was not implemented yet at the time of writing this chapter due to time constraints, but is likely rarely a limiting factor when operating in L1. Implementing it will be necessary to achieve a good accuracy when targeting higher latencies and cache levels.

2. The impact of pending stores on later load and/or store uops: stores whose address calculation was not dispatched or completed yet might prevent later load and store uops from getting dispatched due to memory ordering constraints. More experiments and tests should be done to ascertain this.

3. There are fewer simulated stages than existing for the target microarchitectures. While the number of stages mostly does not impact us as we do not model branch misprediction, it could have a slight impact on resource consumption and could be adjusted in the future.

4. Writeback bus conflicts and execution stack value transfers: while they are detailed well in [141, 142], we expect them to very rarely ever play a visible role in performance degradations, and implementing them to hence have a low *effort / reward* ratio.

5. Partial integer register stalls: some registers' subparts have explicit architectural names (e.g. AL represents the lowest 8 bits of 64-bit register RAX), allowing them to be accessed using different granularities. However, mixing such accesses sometimes incurs various penalties, as described in [143]. As with writeback bus conflicts, we expect the potential implementation gain not to be worth the extra complexity, and consequently opted to leave this issue aside for the time being.

6. Multi-output instructions: some instructions modify several registers. The current model does not support attributing several physical registers to the same instruction, but it could be easily implemented; the only reason why we have not done so yet is that they are infrequent in the HPC codes we target.

Furthermore, while a lot of information is available in terms of how Intel CPUs work, many hardware implementation details are not publicly available. We could fill some of the gaps using reasonable guesses, but they are probably flawed to some degree, restricting the precision attainable by our model.

7.4.8 Ad-Hoc L1 Modeling

While our model does not support detailed cache modeling, we can still change L1 performance in two manners:

1. Change the latency of load and store uops, which reflects L1 latency accurately.

2. Change the rate at which load uops' output is made available to other uops, impacting the effective L1 bandwidth.

We hence implemented the following (optional) features, focusing on loads:

1. Load latency change: the simulator ignores the latency specified for load uops in input files, and sets them to a user-specified value instead. This is an option of convenience as it allows users to quickly change L1's load latency but finer adjustments can be made by modifying loop input files.

2. Cycles per load restriction: the uop status update simulation phase described in 7.4.5.3 was tweaked to actively delay load uops by incrementing their execution time stamp when the defined quota is temporarily exceeded. This prevents them from retiring, and dependent uops from getting dispatched. This bandwidth constraint is expressed in *cycles per load*. Its default value matches the hardware's bandwidth (with each load keeping L1 busy for 0.5 cycles, i.e. up to 2 loads are executable by cycle), but it is a freely modifiable parameter.

This latter feature has no other purpose than to allow for a crude evaluation of how bandwidth constrained a codelet is.

7.5 Validation

The validation work for UFS is two-fold:

1. Accuracy: checking whether the model provides faithful time estimations for loops operating in L1.

2. Speed: making sure simulations are not prohibitively slow for their intended use.

We will focus on SNB validation, as it is the microarchitecture we most targeted during this thesis and identifying SNB performance drops was the primary motivation for developing UFS in the first place. Furthermore, its modeling is used as basis for IVB and HSW support, making SNB validation particularly important.

We will use the *fidelity* metric described in Section 7.1.1 to represent UFS accuracy for each studied loop, and systematically compare UFS results with CQA projections to highlight our model's contributions.

A short study of the time taken by our UFS prototype will be made, and results will be presented in terms of *simulated cycles per second*.

7.5.1 Another Look at our Motivating Examples

We will have a second look at the motivating examples presented earlier to see what light our UFS implementation could shed.

7.5.1.1 Realft2_4_de

Running UFS on the REF variant of *realft2_4_de* gives encouraging results (see Table 7.9), completely filling the gap between the CQA cycles estimation and the measurement.

Table 7.9: Realft2_4_de: UFS Validation (SNB)

Metric	Cycles per Iteration	Fidelity
Measured	23.36	N/A
CQA	16.00	68.49%
UFS (Normal buffers)	23.01	98.50%
UFS (Large buffers)	19.03	N/A

Fidelity for UFS (with regular SNB parameters) is 98.50%, an important improvement over CQA's 68.49%.

*Running UFS again with all out-of-order resources set to have 1000 entries (*UFS - Large Buffers *row) shows nearly 4 (*$23.01 - 19.03 = 3.98$*) cycles can be gained by merely increasing buffer sizes.*

Table 7.10 shows a cycle-per-cycle trace of the UFS simulation for this codelet. Its columns show the following for every cycle:

1. *C / RS*: respectively the cycle count and number of RS entries available at the beginning of the cycle. Other resource counters were left out for simplicity as they do not impact performance for this codelet.
2. *Issued*: issued uops.
3. *Cpltd*: uops reaching execution maturity; they can potentially be retired and dependent uops can now be dispatched.
4. *Dispatched*: where and when each uop is dispatched. Note that the color codes are the same as the ones used in Table 7.2 and Figure 7.2.
5. *Retired*: retired uops.

We chose to show cycles 189 to 211 as a) they show an entire assembly iteration being issued and b) a steady state is reached. Indeed, early iterations can behave in a significantly different manner as the pipeline fills up. Here, the trace for later iterations is strictly identical to the one presented except for shifted iteration numbers.

Several observations can be made:

1. Up to 3 different iterations cohabit in the pipeline: on cycle 189, uops from iteration 10 get issued as uops from iteration 8 are still being retired. However, the distance between the oldest and the youngest uops is of only *uops remaining from iteration 8 + uops from iteration 9 + uops from iteration 10* $= (54 - 46 + 1) + 54 + 4 = 67$, which gives an idea of the window size for this codelet.

2. The port binding algorithm was particularly efficient here, and the workload is as balanced as can legally be. As a consequence, P1 (the bottleneck according to CQA) only handles uops for which it is the only legal option.

3. The number of instructions dispatched per cycle varies from 1 (cycle 207) to 4 (cycle 192), with an average of $54/23 = 2.35$.

4. The RAT often gets stalled by the lack of RS entries in most cycles (e.g. only issuing 2 uops on cycle 190). It only issues uops at full speed on cycles 189 and 193.

5. The reason only 1 uop is issued on cycle 211 is the SNB RAT limitation on issuing uops from different iterations in the same cycle.

6. P1 execution bubbles are revealed, explaining the mismatch between CQA and measurements.

This accuracy gain is due to two main factors:

1. RS size awareness: running UFS again with a virtually infinite RS size shows that 4 cycles could be gained from having larger out-of-order buffers (Table 7.9).

2. Realistic dispatch: heuristics prioritized uops not on the critical path.

The CQA evaluation represents the performance the loop would attain if not for these issues.

7.5.1.2 Realft_4_de

UFS results for the REF variant of *realft_4_de* (see Table 7.11) also explain a sizeable chunk of the gap between the CQA-predicted and the measured performances. However, unlike *realft2_4_de*, only the size of the RS gets in the picture: running UFS again with larger buffers produces the same time estimate as CQA.

7.5.2 In Vitro Validation

We generalized our validation procedure on codelets we had already studied previously, and for which experimental data was readily available (see Chapter 3). We narrowed down our selection to those for which the smallest used data sets fit entirely in L1, in their SSE version. We hence used a) 14 codelets from the Numerical Recipes (NRs) [69] and b) 2 from Maleki's vectorization test codelets [59], present in both their vectorized and scalar versions (see Section 6.5).

Table 7.10: Realft2_4_de UFS Trace

C RS	Issued	Cpltd	Dispatched						Retired
			P0	P1	P2	P3	P4	P5	
189 4	[10,1,1/1] [10,2,1/1] [10,3,1/1] [10,4,1/1]	[8,44,2/2] [8,47,1/1] [9,53,1/2]		[9,15,1/1]				[8,49,1/1]	[8,43,1/1] [8,44,1/2] [8,44,2/2] [8,45,1/1]
190 2	[10,5,1/1] [10,6,1/1]	[8,49,1/1] [8,50,2/2] [8,51,1/1]	[10,2,1/1]	[8,52,1/1]				[9,1,1/1]	[8,46,1/1] [8,47,1/1] [8,48,1/1] [8,49,1/1]
191 3	[10,7,1/1] [10,8,1/1] [10,9,1/1]	[8,53,2/2] [9,1,1/1] [9,14,1/1] [10,2,1/1]	[9,18,1/1]	[9,16,1/1]				[9,28,1/1]	[8,50,1/2] [8,50,2/2] [8,51,1/1]
192 3	[10,10,1/1] [10,11,1/1] [10,12,1/1]	[9,15,1/1] [9,28,1/1]	[9,19,1/1]	[9,17,1/1]	[10,8,1/1]			[9,45,1/1]	
193 4	[10,13,1/1] [10,14,1/1] [10,15,1/1] [10,16,1/1]	[8,52,1/1] [9,45,1/1]	[9,30,1/1]		[10,10,1/1]			[9,4,1/1]	[8,52,1/1] [8,53,1/2] [8,53,2/2] [8,54,1/1]
194 3	[10,17,1/1] [10,18,1/1] [10,19,1/1]	[9,4,1/1] [9,16,1/1]	[9,20,1/1]					[9,29,1/1]	[9,1,1/1] [9,2,1/1] [9,3,1/1] [9,4,1/1]
195 2	[10,20,1/1] [10,21,1/1]	[9,17,1/1] [9,29,1/1]	[9,22,1/1]					[9,42,1/1]	[9,5,1/1] [9,6,1/1] [9,7,1/1] [9,8,1/1]
196 2	[10,22,1/1] [10,23,1/1]	[9,18,1/1] [9,42,1/1] [10,8,1/1]	[9,21,1/1]					[9,54,1/1]	[9,9,1/1] [9,10,1/1] [9,11,1/1] [9,12,1/1]
197 2	[10,24,1/1] [10,25,1/1]	[9,19,1/1] [9,54,1/1] [10,10,1/1]	[9,24,1/1]					[9,26,1/1]	[9,13,1/1] [9,14,1/1] [9,15,1/1] [9,16,1/1]
198 2	[10,26,1/1] [10,27,1/1]	[9,26,1/1] [9,30,1/1]	[9,33,1/1]	[9,38,1/1]				[10,3,1/1]	[9,17,1/1] [9,18,1/1] [9,19,1/1]
199 3	[10,28,1/1] [10,29,1/1] [10,30,1/1]	[9,20,1/1] [10,3,1/1]	[9,23,1/1]					[10,9,1/1]	[9,20,1/1]
200 2	[10,31,1/1] [10,32,1/1]	[9,22,1/1] [10,9,1/1]	[9,25,1/1]					[9,27,1/1]	
201 2	[10,33,1/1] [10,34,1/1]	[9,21,1/1] [9,27,1/1] [9,38,1/1]	[9,43,1/1]	[9,31,1/1]				[9,48,1/1]	[9,21,1/1] [9,22,1/1]
202 3	[10,35,1/1] [10,36,1/1] [10,37,1/1]	[9,24,1/1] [9,48,1/1]	[9,46,1/1]	[9,32,1/1]				[10,12,1/1]	
203 3	[10,38,1/1] [10,39,1/1] [10,40,1/1]	[9,33,1/1] [10,12,1/1]	[10,5,1/1]	[9,34,1/1]					
204 2	[10,41,1/2] [10,41,2/2]	[9,23,1/1] [9,31,1/1] [10,5,1/1]	[10,6,1/1]	[9,35,1/1]				[10,7,1/1]	[9,23,1/1] [9,24,1/1]
205 3	[10,42,1/1] [10,43,1/1] [10,44,1/2]	[9,25,1/1] [9,32,1/1] [10,6,1/1] [10,7,1/1]		[9,36,1/1]	[10,13,1/1]	[10,11,1/1]			[9,25,1/1] [9,26,1/1] [9,27,1/1] [9,28,1/1]
206 3	[10,44,2/2] [10,45,1/1] [10,46,1/1]	[9,34,1/1] [9,43,1/1]		[9,37,1/1]	[10,41,1/2]	[10,44,1/2]			[9,29,1/1] [9,30,1/1] [9,31,1/1] [9,32,1/1]
207 3	[10,47,1/1] [10,48,1/1] [10,49,1/1]	[9,35,1/1] [9,46,1/1] [10,41,1/2] [10,44,1/2]		[9,39,1/1]					[9,33,1/1] [9,34,1/1] [9,35,1/1]
208 1	[10,50,1/2]	[9,36,1/1]		[9,40,1/1]			[9,41,2/2]		[9,36,1/1]
209 2	[10,50,2/2] [10,51,1/1]	[9,37,1/1] [10,11,1/1] [10,13,1/1]		[9,47,1/1]	[10,50,1/2]		[9,44,2/2]		[9,37,1/1] [9,38,1/1]
210 3	[10,52,1/1] [10,53,1/2] [10,53,2/2]	[9,39,1/1] [10,50,1/2]		[9,51,1/1]			[9,50,2/2]		[9,39,1/1]
211 2	[10,54,1/1]	[9,40,1/1] [9,41,2/2]		[10,14,1/1]	[10,53,1/2]		[9,53,2/2]		[9,40,1/1] [9,41,1/2] [9,41,2/2] [9,42,1/1]

Uops are represented by triplets of the following format: [iteration, instruction number, uop rank / number of uops for the instruction]. A fourth element is needed for codelets where some uops are still microfused in the Back-End, but it is not the case here.

Uop ids are consistent with the ones detailed in Table 7.2 and used in Figure 7.2.

Table 7.11: Realft_4_de: UFS Validation (SNB)

Metric	Cycles per Iteration	Fidelity
Measured	15.66	N/A
CQA	12.00	76.63%
UFS (Normal buffers)	14.53	92.78%
UFS (Large buffers)	12.04	N/A

Fidelity for UFS (with regular SNB parameters) is 92.78%, improving over CQA (76.63%), but not as drastically as with realft2_4_de.

*Running UFS again with all out-of-order resources set to have 1000 entries (*UFS - Large Buffers *row) shows 2.5 (*$14.53 - 12.04 = 2.49$*) cycles can be gained with bigger buffers, matching CQA's projection.*

7.5.2.1 Experimental Setup

NR codelets were compiled with icc v.12.1.3, with compilation flags *-O3 -xSSE4.2 -g*; Maleki codelets were compiled with *-O3 -xSSE4.2 -std=c99 -g* for vector versions, and *-O3 -xSSE4.2 -std=c99 -no-vec -g* for their scalar counterpart.

Experimental measurements were made on SNB CPUs, with hyper-threading deactivated, and following the methodology described in Chapter 3. Data L1 size is 32 KB, as with all SNB CPUs. As we focus exclusively per-cycle single-core L1 performance, the exact model number is not relevant.

We perform our validation for the three main DECAN transformations: FP, LS and REF.

7.5.2.2 FP Variant

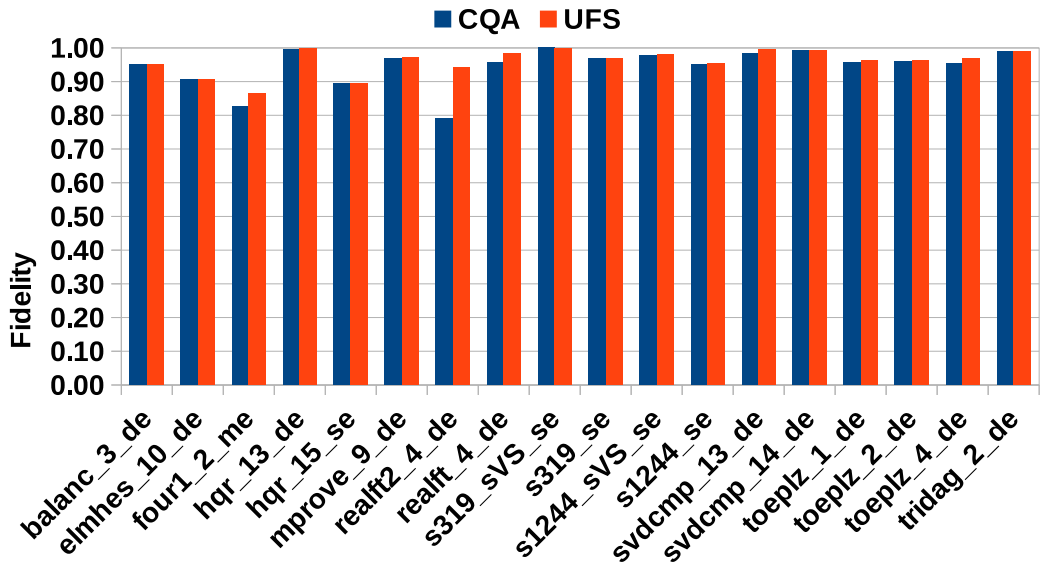


Figure 7.5: In Vitro Validation for FP [SNB]

UFS gives accurate results for FP variants. However, CQA already gives good results in most cases, and UFS only provides a significant improvement for realft2_4_de *(with a* .15 *point gain).*

More importantly, UFS's fidelity is always higher than (or equal to) CQA's.

Figure 7.5 shows the results for the FP versions of the selected codelets. UFS improves projections (compared to CQA) for certain codelets, but in most cases the gain is rather negligible.

The average fidelity for UFS is 96.02%, versus 94.55% for CQA. Hence, UFS has a high fidelity for FP, but does not establish a strong advantage over CQA there.

7.5.2.3 LS Variant

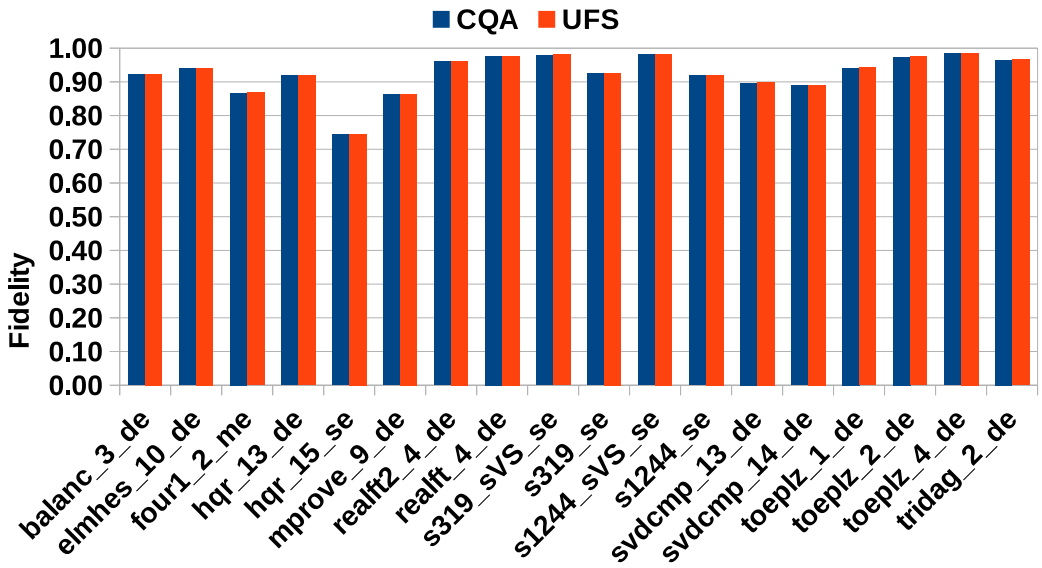


Figure 7.6: In Vitro Validation for LS [SNB]
UFS provides exactly the same time estimations as CQA for LS variants.

UFS results for LS (Figure 7.6) are even less marked than FP ones in regard to differentiating from CQA, and there is no corner case which UFS deals better with anymore.

This is however not surprising, as UFS's main added value is the handling of pipeline hazards not likely to occur with codes composed almost entirely of pure memory instructions: the number of dependencies is reduced, and with it the difficulty in keeping relevant dispatch ports busy.

Still, as with FP, UFS fails to show a clear advantage over regular CQA analyses.

7.5.2.4 REF Variant

REF is the most interesting variant for several reasons:

1. For the user, it is the one that actually matters in terms of experienced performance.

2. For our validation purposes, REF is the variant where the instruction flows from FP and LS are combined, increasing the overall complexity of the whole uop flow.

Results for REF can be found in Figure 7.7. UFS fills important gaps left by CQA, the most important case of which (*realft2_4_de*) we saw in details in

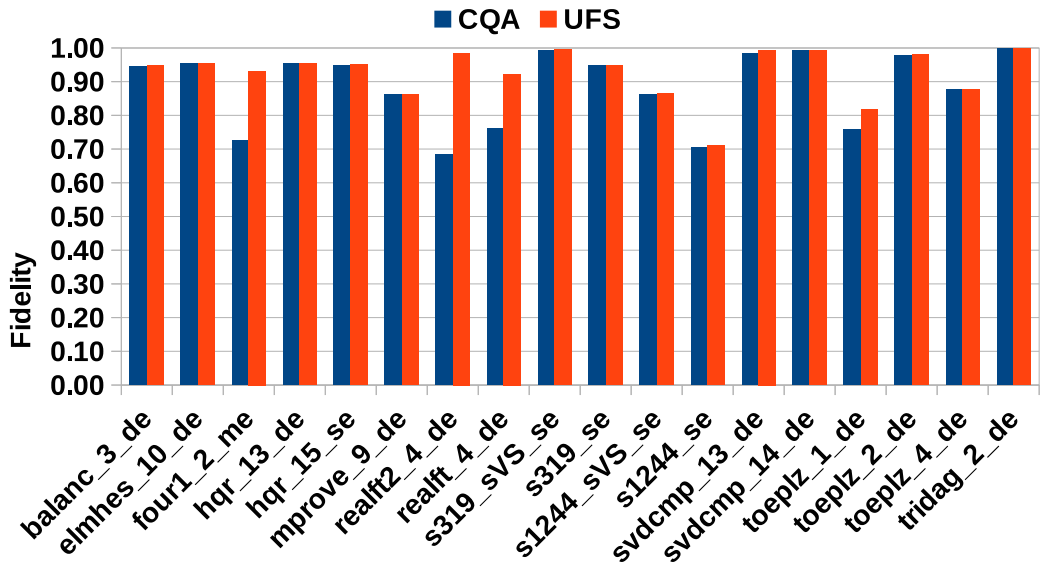


Figure 7.7: In Vitro Validation for REF [SNB]

*CQA modeling is sufficient in the majority of cases, but UFS provides a significant fidelity gain for several codelets (*four1_2_me, realft2_4_de, realft_4_de *and* toeplz_1_de*) without compromising accuracy in any of the others.*

There are still sizeable gaps for certain codelets, with a fidelity of only 71% *in the worst case (*s1244_se*).*

Section 7.5.1.1. The improvements for *four1_2_me, realft2_4_de, realft_4_de* and *toeplz_1_de* were of respectively .21, .30, .16 and .06 points.

However, not everything is explained yet, with projections for *s1244_se* still having an error of around 30% (29% for UFS, 30% for CQA).

The average fidelity for UFS is *92.73%*, against *88.49%* for CQA: the difference is more than twice higher for REF than it is for FP.

7.5.3 In Vivo Validation

We then shifted our validation focus from in vitro codelets to loops from actual industrial applications. They are typically more complex than NR or Maleki codes, and may give a legitimate idea of how UFS fares in real world cases.

7.5.3.1 Experimental Setup

Performance measurements for these applications were performed in vivo using DECAN.

The host machine had a two-socket *E5-2670* SNB CPU, with 32 KB of data L1 cache, 256 KB of L2, and 20 MB of L3. It also had 32 GB of DDR3 RAM.

For each tested application, we selected loops that:

1. Are hot spots: the studied loops are relevant to the application's performance.
2. Are innermost, have no conditional code and can therefore be analyzed out of context (see Section 7.1.4).
3. Have a measured time of more than 500 cycles per loop call for all the classical FP, LS and REF variants. This is needed to make sure measurements are

reliable (small ones can be inconsistent [144]). This may exclude small loops that are called numerous times.

We use DECAN variant *DL1* [37] to force all memory accesses to hit constant locations, and thereby getting a precise idea of what the original loop's performance would be if its working set fit in L1. This also allows us to make direct comparisons between measured cycles per iteration vs. UFS and CQA projections, as other components of the memory hierarchy are artificially withdrawn from the picture.

7.5.3.2 YALES2: 3D Cylinder

YALES2 [123, 124] is a numerical simulator of turbulent reactive flows using the Large Eddy Simulation method. Its performance scales almost linearly with the number of execution cores even with thousands of cores.

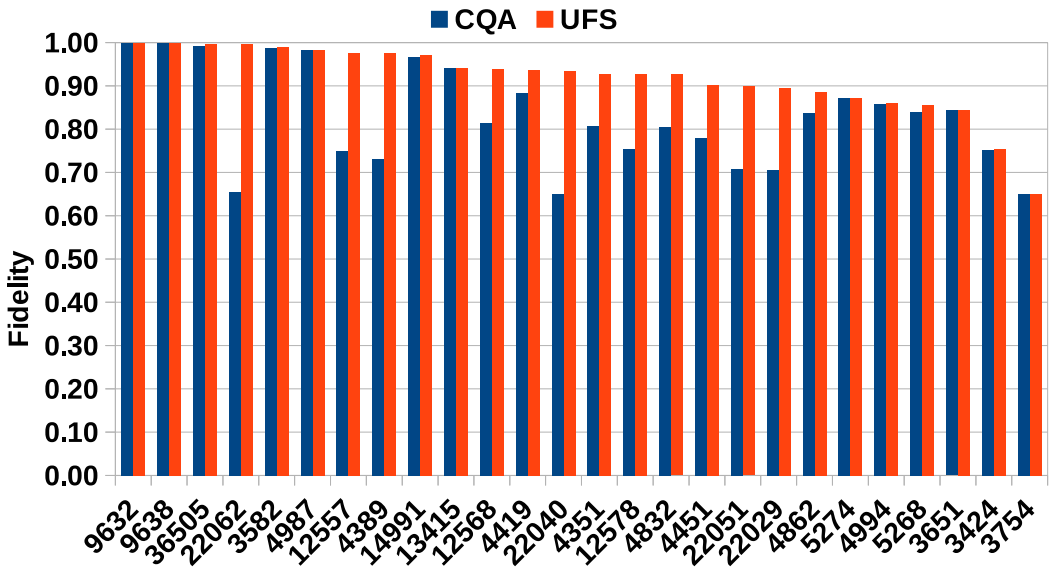


Figure 7.8: In Vivo Validation for DL1: Y2 / 3D Cylinder [SNB]

Gaps between CQA projections and measurements are frequent and often important, but UFS covers many of them at least partially. In the most favorable case (loop 22062), UFS makes a perfect projection (fidelity of 1.00), while CQA's fidelity is at only .65.

Certain loops' performance is still unexplainedly poor, though. For instance, both UFS and CQA have a fidelity of .65 for loop 3754.

Figure 7.8 shows UFS and CQA results for the *3D cylinder* part of this application. UFS shows fidelity gains of more than 0.05 points for 12 loops out of 26, with a maximum gain of .35 for loop 22062. Other particularly important gains include .28 and .24 for respectively loops 22040 and 4389.

Some loops' performance are degraded by factors apparently not modeled by UFS, with disappointing fidelities of respectively .65 and .75 for loops 3754 and 3424.

The average fidelity is of 91.67% for UFS, versus 82.93% for CQA.

7.5.3.3 AVBP

AVBP [130] is a parallel CFD numerical simulator targeting reactive unsteady flows. Its performance scales nearly linearly for up to 4K nodes [145].

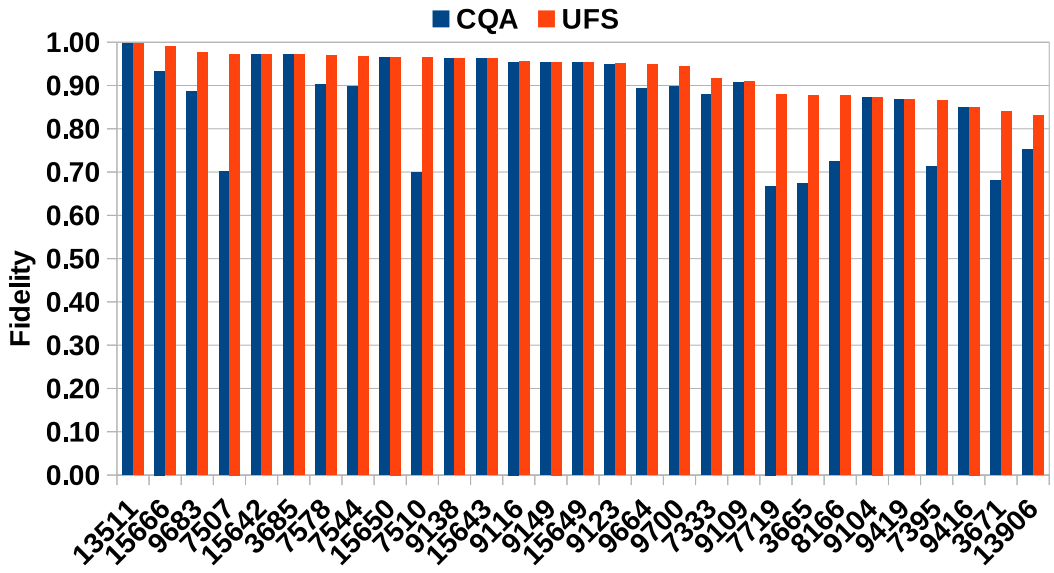


Figure 7.9: In Vivo Validation for DL1: AVBP [SNB]

UFS improves the static modeling of the studied loops significantly, with all projections having a fidelity higher than 75%. *Indeed, the worst case is* 78.18% *for loop 13906.*

There is however still room for improvement.

Figure 7.9 shows UFS and CQA results for 29 AVBP hot loops on Sandy Bridge. UFS shows fidelity gains of more than 0.05 points for 9 of them, with a maximum gain of .27 point for loops 7507 and 7510. Other important gains include .20 for loops 7719 and 3665.

The worst fidelity for UFS is 78.18% for loop 13906 (against 66.76% for CQA on loop 3665).

The average fidelity is of 91.73% for UFS, versus 86.34% for CQA.

7.5.4 Simulation Speed

Speed is very important for performance evaluation tools, especially in the context of optimization: various versions of a program can be tested, e.g. trying different compiler flags or hand optimizations.

A model's quality can be thought of in terms of return on investment: are the model's insights worth their cost?

We will hence study UFS's speed in this section, and evaluate the cost of UFS analyses.

7.5.4.1 Experimental Setup

Simulations were run serially on a desktop machine with an *i7-4770* HSW CPU, running at 3.4 GHz. They were run on a single core, with 32 KB of L1 data cache, 256 KB of L2 cache and 8 MB of L3. It also had 16 GB of DDR3 RAM.

The targeted microarchitecture was SNB, with its default microarchitectural parameters, but simulating different numbers of iterations: 1000 and 100 000. The former is the default one and the most relevant to our analysis, while the latter was run to give an idea of sustained simulation speeds past the initialization phase (slowed down by I/O).

Execution times were measured using the *time* Linux tool, with a resolution time of 10 ms. While other measurement methods would be more precise, we deemed this one to be enough for our purposes here. Furthermore, the time needed to generate the loop input files with MAQAO is not counted here. Measures were performed with 11 meta-repetitions to stabilize results.

7.5.4.2 NRs and Maleki Codelets

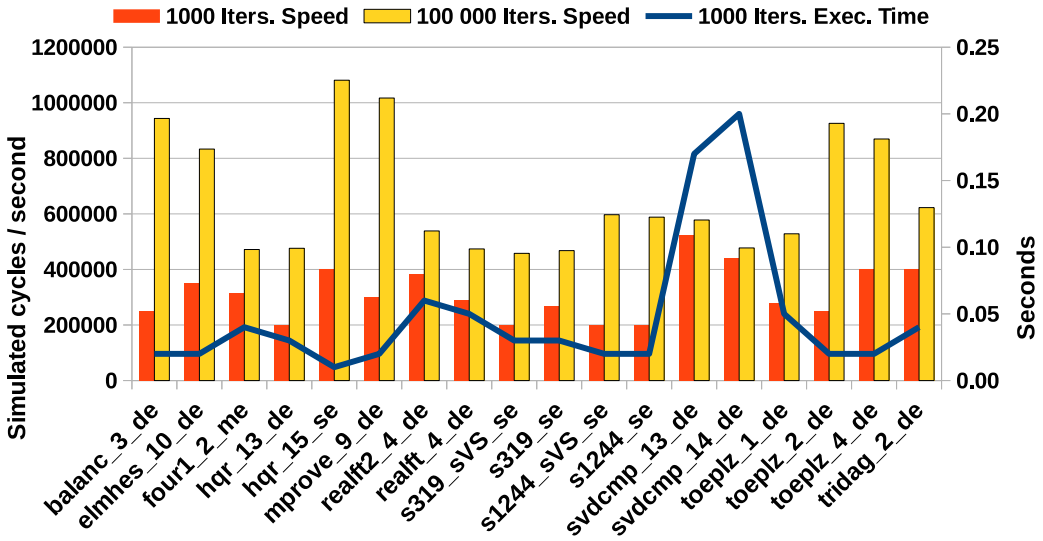


Figure 7.10: UFS Speed Validation for NRs and Maleki Codelets (REF Variant)

Our UFS prototype is very fast, and simulates hundreds of thousands of cycles per second. When simulating 100 000 iterations (which is excessively high for projecting a loop's performance, but may give a better idea of the simulation performance when ignoring initialization costs) this number can be above 1M cycles per second (e.g. hqr_15_se*).*

In practice, simulation times are around .05 seconds for each loop, though some of them reach .20 (e.g. svdcmp_14_de*).*

Figure 7.10 shows simulation speeds for the REF versions of the in vitro codelets studied earlier. We can see that simulating 1000 iterations usually takes around .05 seconds per loop, with peaks to around .20 for certain codelets.

UFS can also simulate many cycles per second, with peaks going as high as 500K cycles / second (for *svdcmp_13_de*) when simulating 1000 iterations, and 1M cycles / second (for *hqr_15_se*) when simulating 100 000. The discrepancy is due to initialization costs being absorbed in the latter case.

For NR and Maleki codelets, we achieve on average:

1. Simulation times (for 1000 iterations) of approximately .04 seconds per loop; and sequentially simulate up to 21.18 loops in a second.

2. The simulation of 314K cycles per second for 1000 iterations (and less relevantly, 664K for 100 000 iterations).

7.5.4.3 YALES2: 3D Cylinder

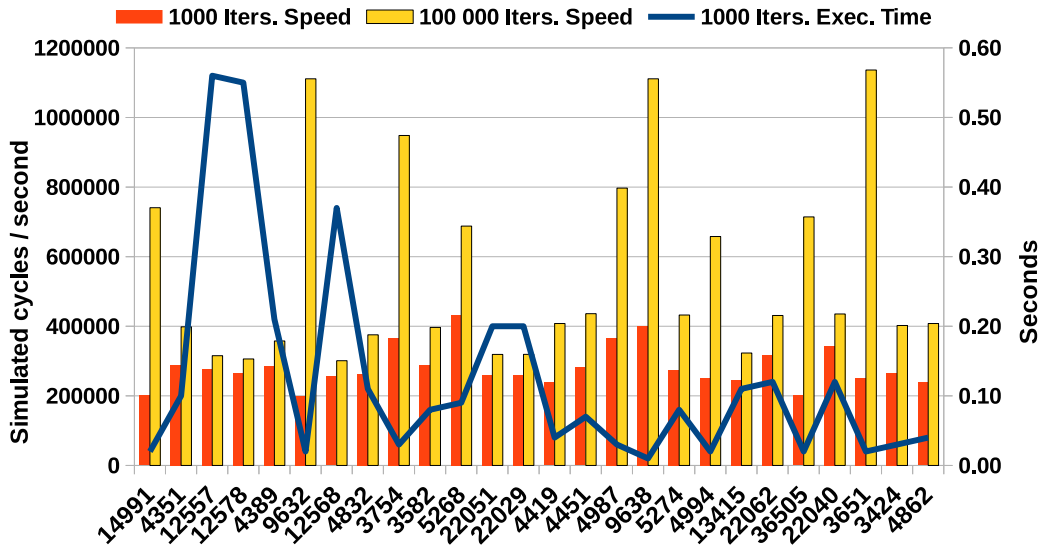


Figure 7.11: UFS Speed Validation for YALES2: 3D Cylinder

Our UFS prototype typically simulates around 200K cycles per second here. This number doubles when simulating 100 000 iterations.

In practice, simulation times are around .10 seconds for each loop. Some of them are particularly long to simulate, though, and can take around .60 seconds to complete (e.g. loop 12557*).*

Figure 7.11 shows simulation speeds for the YALES2 (3D Cylinder) loops we studied. Here, the time needed to simulate 1000 iterations has a high variability, and can go from as low as .02 seconds to as high as .56. This is due to how some of the loops are very complex and comprise hundreds of instructions. Hence, each iteration needs more simulated cycles to complete. Furthermore, the number of instructions can impact the locality of our UFS prototype's data structures, with large loops consequently being simulated less quickly.

However, the numbers of simulated cycles per second is still on par with the ones experienced with in vitro codelets, ranging from 200K (loops *9632* and *36505*) to 430K (loop *5268*) when simulating 1000 iterations. The range is greater when simulating 100 000 iterations, starting at 305K (loop *12578*) and ending at 1.136M (loop *3651*).

For YALES2: 3D cylinder, we achieve on average:

1. Simulation times (for 1000 iterations) of approximately .13 seconds per loop; and sequentially simulate up to 8 loops in a second.

2. The simulation of 281K cycles per second for 1000 iterations (and 549K for 100 000 iterations).

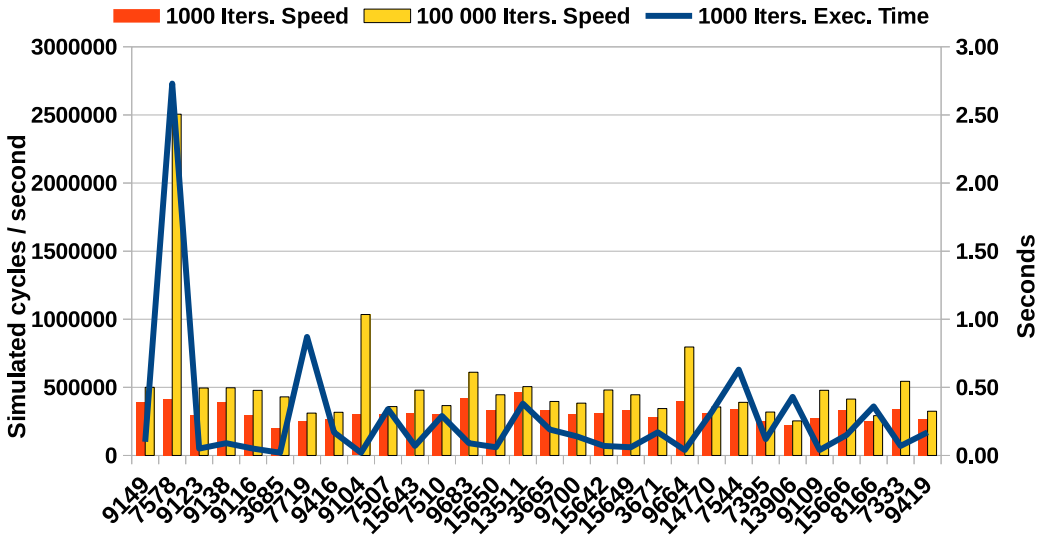


Figure 7.12: UFS Speed Validation for AVBP

Our UFS prototype simulates an average of 300K cycles per second for the studied loops. This average is 1.6x higher when simulating 100 000 iterations.

The average simulation time is of around .28 seconds for each loop. Loop 7578 takes particularly long to simulate (2.73 seconds) and presents an interesting case as it is also the loop for which simulating cycles is the fastest (with 416k cycles per second). The long duration is due to the loop containing 1337 instructions (including divisions), making each of the 1000 iterations require many simulated cycles to complete. The divisions are also what causes the high simulated cycles per second ratio: they slow down the flow of uops in the pipeline (causing stalls), and as a result the average number of changes to the pipeline's state between cycles is quite low.

The average simulation time would be .19 second without this outlier.

7.5.4.4 AVBP

Figure 7.12 shows simulation speeds for the AVBP loops we studied. As with YALES2 loops, the simulation time for a 1000 iteration is highly variable, going from as low as .02 for loop 3685 to as high as 2.73 seconds for loop 7578.

However, simulations take .28 seconds on average, which is longer than for YALES2 (.13 seconds). The difference is due to AVBP loops being more complex and consisting of 200 assembly statements on average, while this metric was at an already high value of 110 for YALES2.

We can hence sequentially simulate ~ 3.57 AVBP loops per second.

7.5.4.5 Comparison with CQA

We will quickly assess CQA's speed to give an idea of how UFS compares to it. As CQA can process hundreds of loops per second, measuring individual loops' processing time would prove tricky. Instead, we chose to run CQA on the YALES2 (3D cylinder) binary for:

1. All loops (including many that are never actually called at runtime).

2. The 26 loops studied earlier (in a single run).

When removing the overhead due to the MAQAO framework (mostly consisting in disassembling the binary) to make comparisons with UFS fair, we found that CQA could process a) 303.78 loops per second in the first case and b) 42.85 in the second.

If we assume the average complexity for all the loops present in the YALES2 binary to be equal to the average complexity of the studied NR and Maleki codelets, we can roughly estimate UFS to be $303/21 \simeq 15$x slower than CQA for low-complexity cases, and $43/8 \simeq 5$x for more complex loops.

Applied to AVBP, the same methodology shows that CQA can process 260.25 loops per second when handling all the loops in the binary, and 30.98 when targeting the hot loops we studied earlier. This brings the overhead for using UFS to $260.25/21 \simeq 12.39$ for simple cases (using the same assumption of average simplicity as above), and to $30.98/3.57 \simeq 8.68$ for complex cases.

7.6 Related Work

Code Quality Analyzer (CQA) [21], to which we compared UFS throughout this chapter, is the tool the closest to UFS that we know of: both analyze loops at a binary / assembly level, rely on purely static inputs and have a special emphasis on L1 performance. They actually both use the MAQAO framework to generate their inputs. CQA works in terms of bandwidth, which it assumes to be unimpeded by execution hazards. As its name suggests, it assesses the quality of targeted loops, for which it provides a detailed bottleneck decomposition as well as optimization suggestions and projections. UFS differs by focusing solely on time estimations, accounting for dispatch inefficiencies and limited buffer sizes. It does so by simulating the pipeline's behavior on a cycle-accurate basis, adding precision at the cost of speed. Finally, CQA supports more microarchitectures than UFS.

IACA [146] works similarly to CQA, and estimates the throughput of a target code based on uop port binding and latency in ideal conditions. It can target arbitrary code sections using delimiting markers, while both CQA and UFS only operate at the loop level. It does not account for the hazards UFS was tailored to detect, and we consequently expect it to be faster but less accurate. Like CQA, IACA also supports more microarchitectures than UFS.

Zesto [46, 147] is an x86 cycle-accurate simulator built on top of SimpleScalar [148] and implements a very detailed simulation of the out-of-order engine similar to that of UFS. However, as with other detailed simulators like [149], the approaches are very different: it works as a regular CPU simulator and handles the semantics of the simulated program. Its simulation scope is also much wider, with a detailed simulation of branch prediction, caches and RAM. UFS focuses solely on the execution pipeline, and particularly the out-of-order engine. It disregards the semantics, and targets loops directly with no need for contextual information (such as register values, memory state, etc.), making it considerably faster due to both not having to simulate regions of little interest and simulating significantly fewer things. Furthermore, UFS targets SNB, IVB and HSW, while to the best of our knowledge Zesto only supports older microarchitectures.

Very fast simulators exist, but typically focus on different problematics. For instance, Sniper [150, 151] uses both interval simulation (an approach focusing on miss events) and parallelism to simulate multicore CPUs efficiently. As said events (cache misses and branch mispredictions) are irrelevant in the cases targeted by

UFS (memory accesses always hit L1, loops have no if statements and have large numbers of iterations), the use cases are completely disjoint.

UFS is to our knowledge the only model targeting binary / assembly loops that both disregards the execution context and accounts for dispatch hazards and limited out-of-order resources.

7.7 Future Work

Evaluating the impact of unmodeled hardware constraints would be interesting to determine whether or not implementing them in UFS could be profitable. Such constraints include writeback bus conflicts and partial register stalls.

The impact of simulating fewer loop iterations should also be studied, as our current default value of 1000 may be unnecessarily high and time consuming.

As our base UFS model is aimed at Sandy Bridge, we could easily construct models for incremental improvements such as Ivy Bridge and Haswell on top of it. However, a validation work is necessary to evaluate their respective fidelities, and see if more microarchitecture-specific rules have to be implemented. Expanding the model to support further "Big Core" microarchitectures (e.g. Broadwell, Skylake...) would also be of interest.

The idea of Uop Flow Simulation can be applied to vastly different microarchitectures (such as the one used in Silvermont cores, or even ARM CPUs), and could have interesting applications beyond performance evaluation tools. For instance, its working out of context means it could easily be used by compilers to better evaluate and improve a generated code's quality.

In terms of codesign, UFS models could be used to quickly estimate the impact of a microarchitectural change on thousands of loops in a few minutes. Coupling this modeling technique with a bandwidth-centric fast-simulation model such as Cape [70] would allow for non-L1 cases to be handled efficiently as well.

7.8 Acknowledgements

We would like to thank Gabriel Staffelbach (CERFACS) for having provided our laboratory with the AVBP application, as well as Ghislain Lartigue and Vincent Moureau (CORIA) for providing us with YALES2.

We would also like to thank Mathieu Tribalat (UVSQ) and Emmanuel Oseret (Exascale Computing Research) for performing and providing the in vivo measurements we used to validate UFS on the aforementioned applications.

7.9 Conclusion

We demonstrated UFS, a cycle-accurate loop performance model allowing for the static, out-of-context analysis of assembly loops. It takes into account many of the low-level details used by tools like CQA or IACA, and goes further by estimating the impact of out-of-order resource sizes and various pipeline hazards.

Our UFS prototype shows that UFS is very accurate, and exposes formerly unexplained performance drops in loops from industrial applications and in vitro codelets alike. Furthermore, it offers very high simulation speeds and can serially process dozens of loops per second, making it very cost effective.

Chapter 8

Conclusion

This chapter will summarize the contributions of this thesis and present leads for future work.

8.1 Contributions

This thesis has explored the identification, quantification and modeling of HPC loops' bottlenecks, and provided the following major contributions:

1. PAMDA (in Chapter 4), a performance assessment methodology that combines static and dynamic analysis tools to help software developers find performance bottlenecks, quantify them and expose optimization opportunities.

2. An extension of the Cape linear model to Sandy Bridge (in Chapter 5), and an application thereof with VP3 (in Chapter 6) a tool evaluating potential vectorization gains for scalar loops.

3. UFS (in Chapter 7), an approach combining static analysis and cycle-accurate simulation to predict loop execution times a) with realistic out-of-order engine constraints and b) at a very fast speed. UFS identifies and quantifies performance problems not well captured by PAMDA or/and Cape modeling.

More minor contributions include:

1. A framework and methodology to study single-loop codelets from different angles, using static and dynamic tools and different execution parameters (in Chapter 3).

2. An experimental approach to quantify out-of-order buffers (in Appendix A).

8.2 Publications

Some of the work performed for this thesis was also presented in the following publications:

1. Simsys: A Performance Simulation Framework [70].

2. PAMDA: Performance Assessment Using MAQAO Toolset and Differential Analysis [85].

3. VP3: A Vectorization Potential Performance Prototype [121].

8.3 Future Work

We will present the main research leads and challenges related to our work here.

8.3.1 Differential Analysis

Differential analysis is the main objective of the DECAN tool, which is used as a component of most the tools and methodologies presented in this thesis.

The buffer-related interactions between the FP and LS instruction streams in the out-of-order engine represent a challenge to the differential analysis approach. Indeed, we could not find loop transformation rules that would directly isolate the performance impact of limited buffer sizes, so we can only estimate them by assuming leftovers not explained by other transformations.

Studying how safe this assumption is using detailed cycle-accurate simulators would be interesting, as well as actually trying to find an acceptable way to directly measure the impact of buffer sizes. A possible lead is to use dependency-breaking instructions (such as zero-idiom *XORPS*) to reset dependencies after each load and before each store. However, this potential transformation would have an impact on the Front-End which should be accounted for.

8.3.2 Cape Modeling

While the Cape bandwidth-centric modeling is very fast, flexible and usable for various purposes, it is currently being negatively affected by latency and buffer-size related phenomena. This needs to be addressed to improve the model's precision, especially when explored potential designs get too distant from the original machine's characteristics.

This could be done by involving simulator tools to handle part of the projection work (and e.g. refine final designs). While this would come at a cost in terms of projection speed, it would fit Cape's objective of exploring designs at a low cost while leaving the implementation details to more specialized tools.

Another possibility is to explicitly model latency and buffer widths within Cape. It may be possible to use specialized pseudo-nodes to adjust bandwidth-centric projections, but would require a more extensive knowledge of all the key queues and latencies within the modeled system.

8.3.3 Uop Flow Simulation

Testing and validating UFS on Ivy Bridge, Haswell and Broadwell would be a worthwhile endeavor, especially as Core CPUs evolve at Intel's fast-paced cycles of *ticks* and *tocks* [17]. However, the interest could be even greater on microarchitectures that are different altogether, such as Intel Silvermont, AMD's Bulldozer and Zen, ARM's ARMv8 and IBM's POWER8.

UFS could also be used to perform sensitivity analyses on loops regarding out-of-order buffers or access latencies, and evaluate their behavior beyond just L1.

Finally, as UFS is very fast and works completely out-of-context, it could reasonably be used by compilers instead of less detailed heuristics to better evaluate the quality of generated assembly codes.

Appendix A

Quantifying Effective Out-of-Order Resource Sizes

We can quantify them experimentally by crafting loops purposefully straining them, using the out-of-order weaknesses presented in Section 7.2. [61] achieved this for the ROB and the PRF using linked lists with poor spatial locality, also finding that FP and Integer registers are not entirely independent.

We extend on this work by:

1. Using square root operations instead of linked lists to generate stable high latencies, and to get constant execution times for as long as the number of targeted resources is not exceeded.

2. Measuring more resources (Branch, Load and Store buffers), as well as the Reservation Station (for which a slightly different technique has to be used).

3. Doing the measurements on HSW further to SNB and IVB.

A.1 Basic Experimental Blocks

We use different techniques to handle the two resource deallocation schemes existing in the target systems:

1. The "allocated at issue, released at retirement" scheme used for the BB, LB, PRFs, ROB and SB. Here, we use blocks of slow FP square roots to *jam* uop retirements, and independent instructions targeting different resources. This setup is described in Table A.1.

   The dispatcher is able to keep the divider unit permanently busy for as long as the next iteration's jam's first uop is issued before the current iteration's jam's last uop is fully executed. This is not the case anymore when the number of resources required by payload uops (and the control instructions) exceeds those available and causes issue stalls.

2. The "allocated at issue, released at dispatch" scheme, only used for the RS. The same idea is applied, but this time the jam prevents payload uops from getting dispatched. The latter are made to depend on the last square root operation for this purpose, making sure they stay in the RS until the last jam's uop is executed.

We can then try to determine the size of a given resource by varying the number of resources needed in the payload, and looking for the performance break point. We will refer to the number of instructions in the payload as being the *Payload Size*, or P.

The amount of square root instructions in the jam may have to be tweaked depending on the expected size of the target buffer. As square root operations being slow allows us to stack more uops behind them without their becoming a bottleneck, we use an approximation of Pi to increase their effective latency[1].

As a side note, we use the AVX instruction set to be able to use the 3-operand form, and be able to use input operands non-destructively. We do however only use 128-bit vectors to simplify uop counting and dispatching. For instance, *VSQRTPD* would need 3 uops in its 256-bit version in SNB.

Table A.1: Resource Quantifying Experiment Example

#Line	Instruction	Purpose
1	VSQRTPD %xmm0, %xmm1	Jamming
2	VSQRTPD %xmm0, %xmm1	Jamming
3	[instruction needing targeted resource]	Payload
4	SUB $1, %rdi	Loop Control
5	JG .LOOP	Loop Control

The jam *consists of slow FP square root operations, leaving plenty of time for the* payload *instructions to be issued and executed out-of-order.*

Retirement being done in-order prevents the payload's resources from being released until the jam itself is retired.

We can adjust the number of required resources by modifying the number of instructions in the payload.

A.2 Quantifying Branch Buffer Entries

The Branch Buffer keeps track of all branch uops, allowing the out-of-order engine to restore the Instruction Pointer to the correct address when the branch predictor is wrong.

We use *JE 0* branch instructions to fill up the branch buffer, ensuring they are never taken by setting the test flag with a always-false *CMP $-1, %rdi* comparison. The throughput for conditional branch instructions is 1 uop per cycle on SNB and IVB, and 2 on HSW. Furthermore, we used 5 *VSQRTPD* instructions to create a $5 * 21 = 105$-cycle jam on SNB ($5 * 14 = 70$ for IVB/HSW), leaving enough time for around $105 * 1 = 105$ branch uops to be dispatched in parallel (resp. $70 * 1 = 70$ for IVB, and $70 * 2 = 140$ for HSW).

An example assembly code for $P = 2$ is presented in Table A.2. As a side note, the target loops containing extra branches made it impossible to process them with the usual framework from Chapter 3, so measurements were made using source-level *RDTSC* probes instead.

Results for varying payload sizes are shown in Figure A.1. A performance degradation appears for point 48 in SNB, IVB and HSW: when taking into account the regular loop branch, it shows only 48 branches can be in-flight at the same time. This matches official figures perfectly (see Table A.3).

[1]The effective complexity of FP Square Root or Division operations depends on the complexity of the input numbers. Dividing by 1 can consequently be done faster than by 1.123.

Table A.2: Resource Quantifying Experiment Example for the BB (P = 2)

#Line	Instruction	Purpose
1	CMP $-1, %rdi	Changing test flag
2	VSQRTPD %xmm0, %xmm1	Jamming retirement
3	VSQRTPD %xmm0, %xmm1	Jamming retirement
4	VSQRTPD %xmm0, %xmm1	Jamming retirement
5	VSQRTPD %xmm0, %xmm1	Jamming retirement
6	VSQRTPD %xmm0, %xmm1	Jamming retirement
7	JE 0	Payload
8	JE 0	Payload
9	SUB $1, %rdi	Loop Control
10	JG .LOOP	Loop Control

This is the BB Quantifying Experience's assembly code for 2 payload instructions. Retirement is jammed with VSQRTPD *instructions, and a constant test flag is prepared so that payload branches are never taken.*

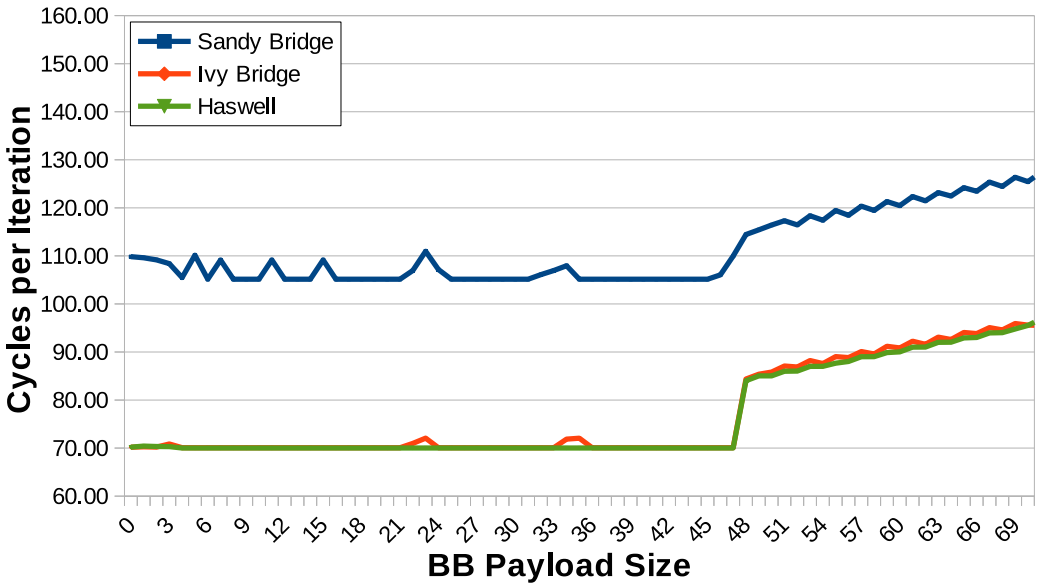


Figure A.1: Quantifying Branch Buffer Entries

The performance drops on point 48 for all studied architectures. When accounting for the experimental loop's base branch, it shows SNB, IVB and HSW all have 48 entries in their Branch Buffer.

Table A.3: BB Size: Measured vs. Official

Uarch	Measured	Official	Difference
SNB	[47-48]	48	0
IVB	48	48	0
HSW	48	48	0

Measurements match official numbers perfectly for the Branch Buffer.

A.3 Quantifying Load Buffer Entries

The LB keeps track of all load uops from the time they were issued until they are retired. It allows the out-of-order engine to prevent potential read-write conflicts between load and store uops.

The experiment used to quantify LB entries is a simple variation of the generic one presented in Table A.1, in which the payload is made of varying numbers of load instructions.

Instruction "*VMOVUPS mem, xmm*" was chosen due to having a high reciprocal throughput of 2 on all target microarchitectures (provided the memory location they target is properly aligned on a 16B boundary, as is the case here). Furthermore, we used 5 *VSQRTPD* instructions to create a $5*21 = 105$-cycle jam on SNB ($5*14 = 70$ for IVB/HSW), leaving enough time for around $105 * 2 = 210$ (resp. $70 * 2 = 140$) loads to be executed in parallel.

An example assembly code for $P = 2$ is presented in Table A.4.

Results for varying payload sizes are shown in Figure A.2. A performance degradation appears for point 65 in SNB and IVB, showing that the 65th instruction needing an LB entry was the one too many for these microarchitectures. This is strong evidence that both SNB and IVB have 64 LB entries. Haswell's performance break point occurs on point 73, suggesting it has 72 entries. These numbers match official figures perfectly (see Table A.5).

Table A.4: Resource Quantifying Experiment Example for the LB (P = 2)

#Line	Instruction	Purpose
1	VSQRTPD %xmm0, %xmm1	Jamming retirement
2	VSQRTPD %xmm0, %xmm1	Jamming retirement
3	VSQRTPD %xmm0, %xmm1	Jamming retirement
4	VSQRTPD %xmm0, %xmm1	Jamming retirement
5	VSQRTPD %xmm0, %xmm1	Jamming retirement
6	VMOVUPS 0(%rsi), %xmm1	Payload
7	VMOVUPS 0(%rsi), %xmm1	Payload
8	SUB $1, %rdi	Loop Control
9	JG .LOOP	Loop Control

This is the LB Quantifying Experience's assembly code for 2 payload instructions. In this case, the payload consists of register loads, each of which needing 1 LB entry to be issued. Its result can be seen in Figure A.2 on coordinate X = 2.

Table A.5: LB Size: Measured vs. Official

Uarch	Measured	Official	Difference
SNB	64	64	0
IVB	64	64	0
HSW	72	72	0

Measurements match official numbers perfectly for the LB.

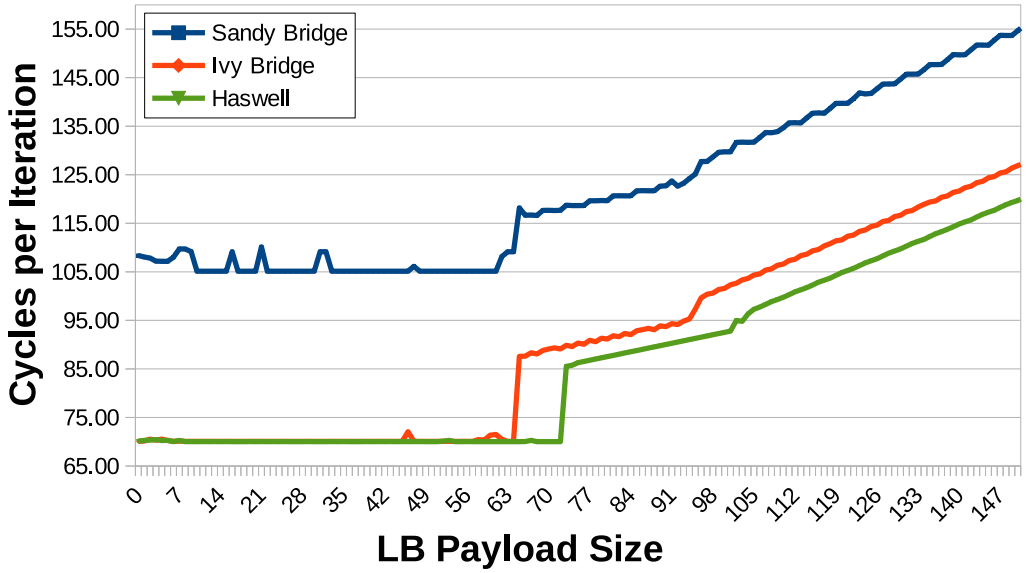


Figure A.2: Quantifying Load Buffer Entries

The performance drops significantly on point 65 for SNB and IVB, and 73 for HSW. This suggests SNB and IVB have 64 LB entries, and HSW 72.

We can also notice SNB experiences performance degradation spikes. This seems to be fixed in IVB and HSW.

A.4 Quantifying PRF Entries

The PRF keeps track of the register values attributed by each uop. It allows registers to hold speculative values while also keeping the architectural state, making sure there is always a legal state to get back to in case of e.g. branch misprediction.

A.4.1 Quantifying FP PRF Entries

The FP PRF keeps track of vector register values.

Table A.6: Resource Quantifying Experiment Example for the FP PRF (P = 4)

#Line	Instruction	Purpose
1	VSQRTPD %xmm0, %xmm1	Jamming retirement
...	VSQRTPD %xmm0, %xmm1	Jamming retirement
15	VSQRTPD %xmm0, %xmm1	Jamming retirement
16	VXORPD %xmm3, %xmm5, %xmm4	Payload
17	VADDPD %xmm3, %xmm3, %xmm4	Payload
18	VXORPD %xmm3, %xmm5, %xmm4	Payload
19	VADDPD %xmm3, %xmm3, %xmm4	Payload
20	SUB $1, %rdi	Loop Control
21	JG .LOOP	Loop Control

The jam comprises 15 VSQRTPD *instructions. The payload consists of intertwined xors and additions, each needing an FP PRF entry to be issued.*

As with the LB, the experiment used to quantify FP PRF entries is a variation

of the one presented in Table A.1. The payload comprises different instructions to increase the uop throughput per cycle:

1. "*VXORPD %xmm3, %xmm5, %xmm4*" only has a throughput of 1, but gets dispatched on a different port from *VSQRTPD*, avoiding potential dispatch conflicts.

2. "*VADDPD %xmm3, %xmm3, %xmm4*" also has a throughput of 1, but gets dispatched on ports different from either *VSQRTPD* or *VXORPD*.

Hence, alternating between *VXORPD* and *VADDPD* allows us to solicit different execution ports, and increase the payload's potential throughput to 2 uops per cycle.

We used 15 *VSQRTPD* instructions to create a $15 * 21 = 315$-cycle jam on SNB ($15 * 14 = 210$ for IVB/HSW), leaving enough time for around $315 * 2 = 360$ (resp. $210 * 2 = 420$) payload uops to be executed in parallel.

An example assembly code for $P = 4$ is presented in Table A.6.

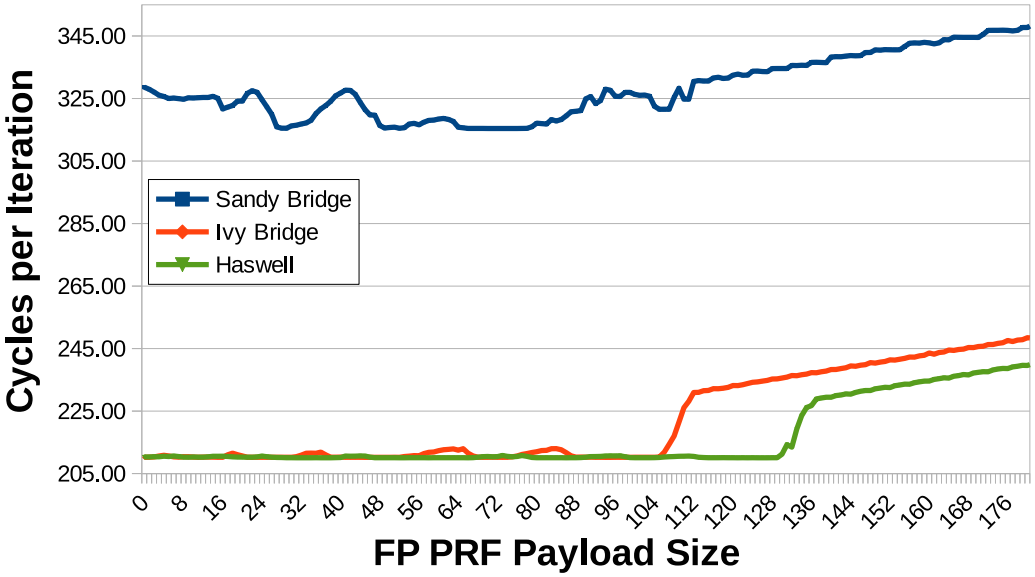


Figure A.3: Quantifying FP PRF Entries

The performance breaking points for IVB and HSW are respectively around [108-112] and [131-137]. The SNB curve is considerably less clear, but there is a clear trend upwards starting from point 80.

Results for varying payload sizes are shown in Figure A.3. Unfortunately, results are not as clear cut as for the LB, and require more interpretation:

1. The experiments actually count registers available for the speculative state, rather than all those in the FP PRF.

2. The jam's uops also require vector registers to be issued: this creates an offset of 1 between the size of the payload and the tested number of FP PRF entries. For instance, for $P = X$, $X + 1$ entries are needed to get visibility on the next iteration's jam.

SNB exhibits strange performance bumps while decidedly still within the FP PRF's size. It is also harder to pinpoint a particular break point for SNB than for IVB or HSW.

Table A.7: FP PRF Size: Measured vs. Official

Uarch	Measured	Official	Difference
SNB	[80-112]	144	32
IVB	[109-113]	144	31
HSW	[132-138]	168	30

A 16-register difference is expected, as there are 16 named vector registers, and as many physical registers have to be used to maintain their latest known-to-be-valid values (i.e. affected by already-retired uops). We do not know why 16 others apparently cannot be used for speculative states.

Table A.7 shows the measured sizes vs. the official ones. A constant difference of 32 registers seems to appear. While 16 can easily be explained due to having to maintain the architectural state for the 16 named vector registers (XMM[0-15]), it is not clear why twice this number is apparently unusable for speculative states.

A.4.2 Quantifying Integer PRF Entries

The Integer PRF is the equivalent of the FP PRF for general purpose registers.

Table A.8: Resource Quantifying Experiment Example for the Integer PRF (P = 4)

#Line	Instruction	Purpose
1	VSQRTPD %xmm0, %xmm1	Jamming retirement
...	VSQRTPD %xmm0, %xmm1	Jamming retirement
15	VSQRTPD %xmm0, %xmm1	Jamming retirement
16	ADD $0, %r11	Payload
17	ADD $0, %r12	Payload
18	ADD $0, %r13	Payload
19	ADD $0, %r10	Payload
20	SUB $1, %rdi	Loop Control
21	JG .LOOP	Loop Control

The jam comprises 15 VSQRTPD *instructions. The payload consists of "add %0, %reg" instructions, which have a throughput of 3 per cycle.*

The experiment used to quantify Int PRF entries is very similar to the one used for the FP PRF. The payload only uses *ADD* instructions, but they can be dispatched on three different ports, allowing for a throughput of 3 uops per cycle on SNB/IVB, and of 4 on HSW. However, unlike previously, we cannot ascertain there will be no dispatch conflict between uops from the jam and those from the payload.

We used 15 *VSQRTPD* instructions to create a $15 * 21 = 315$-cycle jam on SNB ($15 * 14 = 210$ for IVB/HSW), leaving enough time for around $315 * 3 = 945$ (resp. $210 * 3 = 630$) payload uops to be executed in parallel.

An example assembly code for $P = 4$ is presented in Table A.8.

Table A.9 shows the measured sizes vs. the official ones. The difference between them seems to improve with Haswell, but does not quite reach the ideal limit of 16.

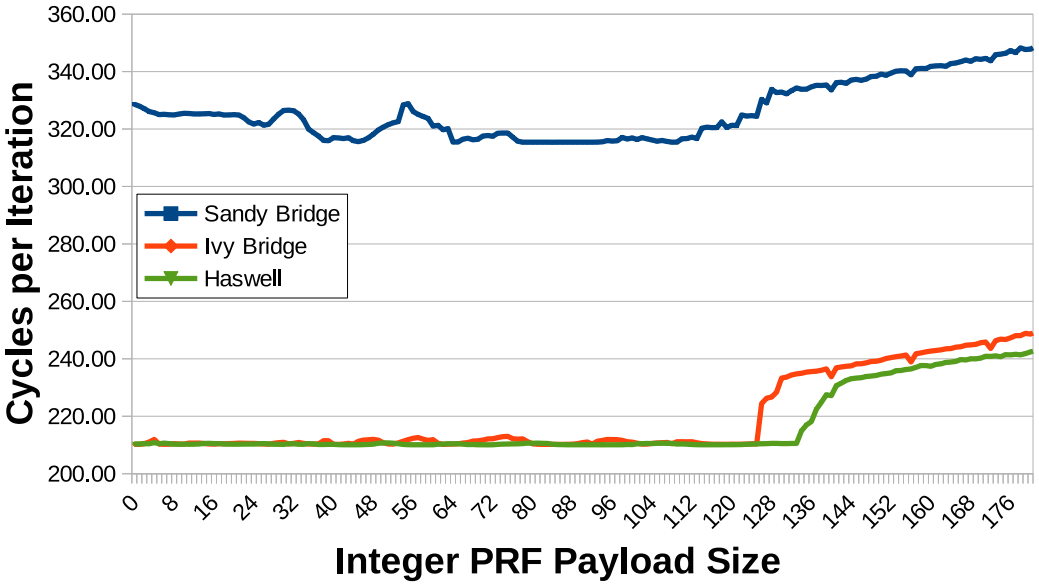


Figure A.4: Quantifying Integer PRF Entries

Table A.9: Integer PRF Size: Measured vs. Official

Uarch	Measured	Official	Difference
SNB	[114-128]	160	32
IVB	[126-130]	160	30
HSW	[136-144]	168	24

The gap between the official and the effective sizes of the Int PRF is roughly the same as the one for the FP PRF in both SNB and HSW. It makes sense, as there are as many named GP registers as vector ones in the studied modes. However, some progress is apparent in HSW.

A.4.3 Quantifying Overall PRF Entries

[61] shows something seems to limit the overall number of registers allocated at a given point. We hypothesize this limit as resulting from the existence of a resource shared by the Integer and the FP PRFs, and call it *Overall PRF*.

The experiment used to quantify Overall PRF entries is mixing the payloads used for the Int and the FP PRFs. It results in a payload with a throughput of 2 uops per cycle, and roughly allocating as many registers from both PRFs.

Table A.11 shows the measured sizes vs. the one we would expect. Here, considering we could find no official description of a limit on overall register allocations, we used the size of the ROB as the "official" limit. While IVB seems little affected by this problem, both SNB and HSW are shown to suffer from it. This confirms the observation made in [61].

A.5 Quantifying ROB Entries

The ReOrder Buffer keeps track of uops from the moment they are issued and until they are retired. It is the biggest out-of-order buffer as all Back-End uops need to be held in it during their life-time, including those that do not need to be dispatched

Table A.10: RQ Experiment Example for the Overall PRF (P = 4)

#Line	Instruction	Purpose
1	VSQRTPD %xmm0, %xmm1	Jamming retirement
...	VSQRTPD %xmm0, %xmm1	Jamming retirement
15	VSQRTPD %xmm0, %xmm1	Jamming retirement
16	ADD $0, %r10	Payload
17	VADDPD %xmm3, %xmm3, %xmm4	Payload
18	ADD $0, %r10	Payload
19	VADDPD %xmm3, %xmm3, %xmm4	Payload
20	SUB $1, %rdi	Loop Control
21	JG .LOOP	Loop Control

The jam comprises 15 VSQRTPD *instructions. The payload consists of intertwined GP and vector additions, for a cumulated throughput of 2 uops per cycle.*

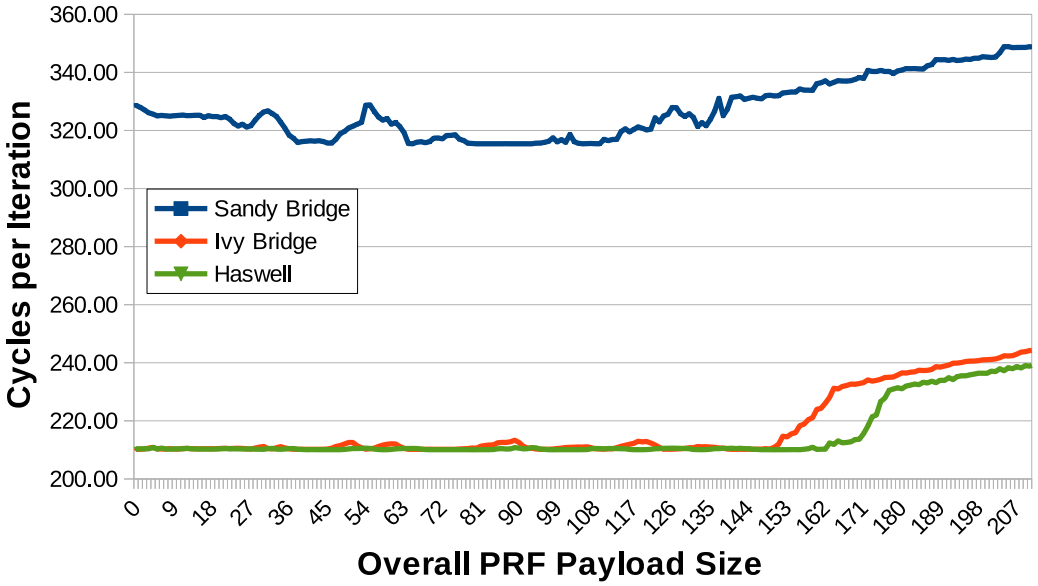


Figure A.5: Quantifying Overall PRF Entries

Table A.11: Overall PRF Size: Measured vs. Official

Uarch	Measured	Official	Difference
SNB	[115-141]	168?	27?
IVB	[152-165]	168?	3?
HSW	[164-177]	192?	15?

The number of entries attributed to the "Overall PRF" is increased in IVB, but did not follow the same progression as the FP and Int PRFs in HSW: more registers can be allocated simultaneously (177 vs. 165), but the gap increases (15 vs. 3).

(e.g. NOPs). The *Instruction Window* represents all the uops in the ROB, from which the out-of-order engine can try to extract instruction-level parallelism.

The experiment used to quantify ROB entries uses the same light jam as for

Table A.12: RQ Experiment Example for the ROB (P = 4)

#Line	Instruction	Purpose
1	VSQRTPD %xmm0, %xmm1	Jamming retirement
...	VSQRTPD %xmm0, %xmm1	Jamming retirement
5	VSQRTPD %xmm0, %xmm1	Jamming retirement
6	NOP	Payload
7	NOP	Payload
8	NOP	Payload
9	NOP	Payload
10	SUB $1, %rdi	Loop Control
11	JG .LOOP	Loop Control

The jam comprises 5 VSQRTPD *instructions. The payload consists of NOP instructions, which do not need to get dispatched, and have a throughput of 4 per cycle.*

LB entries: even though it is considerably bigger, the need to jam the execution is lessened by our use of *NOP* instructions in the payload and their high throughput of 4 uops per cycle. Indeed, *NOP* ("No Operation") does nothing more than change the Instruction Pointer, and is "executed" at the same time as it is issued. However, it does need to be retired in-order.

Our use of 5 *VSQRTPD* instructions create a $5 * 21 = 105$-cycle jam on SNB ($5 * 14 = 70$ for IVB/HSW), meaning as many as $105 * 4 = 420$ (resp. $70 * 4 = 280$) *NOP*s could be issued in parallel.

An example assembly code for $P = 4$ is presented in Table A.12.

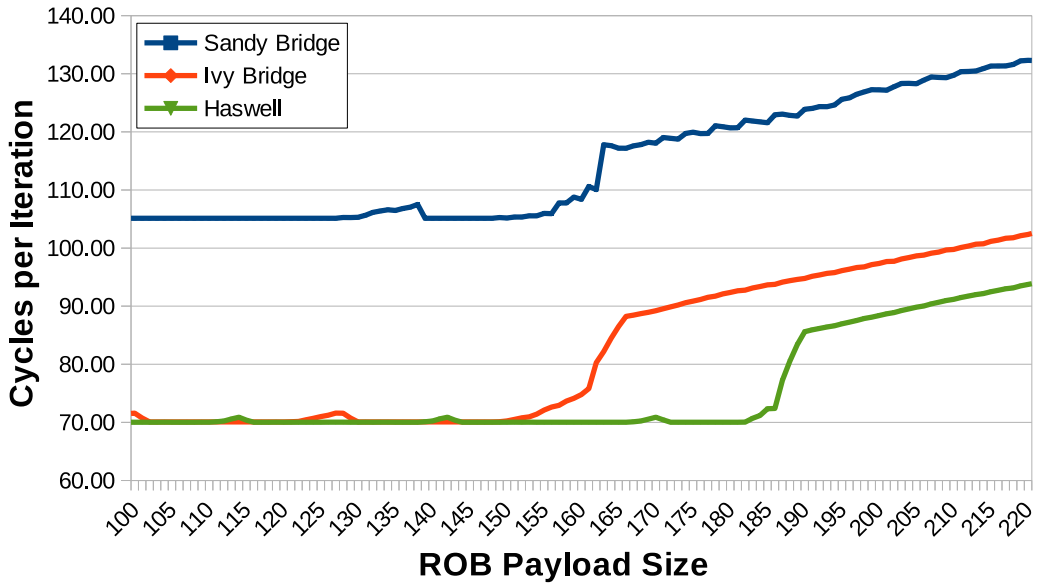


Figure A.6: Quantifying ROB Entries

Figure A.6 shows experimental results for the ROB measurement experiments. To interpret this figure, one needs to keep in mind instructions from the loop control and the jam also use up ROB slots, causing a slight shift between the size of the

payload size and the number of entries actually reserved.

Table A.13: ROB Size: Measured vs. Official

Uarch	Measured	Official	Difference
SNB	[159-165]	168	3
IVB	[155-168]	168	0
HSW	[185-192]	192	0

The difference between the effective (measured) and the expected (official) ROB sizes is very small. However, the measured degradation window for Ivy Bridge is not negligible, with a span of over 13 entries.

Table A.13 shows the measured sizes vs. the one we would expect. The difference is very small, through performance degradations occur when nearing the total size. Haswell improves on this behavior without completely fixing it.

A.6 Quantifying RS Entries

The Reservation Station holds all uops that need to be dispatched from the moment they are issued until they are dispatched. It keeps track of the dependencies between them, and dispatches uops once their operands are ready. This latter characteristic makes it unique among out-of-order resources, as its resources are released *before* retirement.

Table A.14: RQ Experiment Example for the RS (P = 4)

#Line	Instruction	Purpose
1	VSQRTPD %xmm0, %xmm1	Jamming dispatch
...	VSQRTPD %xmm0, %xmm1	Jamming dispatch
5	VSQRTPD %xmm0, %xmm1	Jamming dispatch
6	VADDPS %xmm1, %xmm1, %xmm2	Payload
7	VADDPS %xmm1, %xmm1, %xmm2	Payload
8	VADDPS %xmm1, %xmm1, %xmm2	Payload
9	VADDPS %xmm1, %xmm1, %xmm2	Payload
10	SUB $1, %rdi	Loop Control
11	JG .LOOP	Loop Control

The jam comprises 5 VSQRTPD *instructions. The payload consists of VADDPS instructions with a dependency on a register produced by the jam, forcing them to stay in the RS until XMM1 is ready. They have a throughput of 1 uop per cycle.*

The experiments to quantify the number of RS entries have to take this into account, as this RS slots getting freed out-of-order makes our classical set-up inadequate. We will hence adjust the payload to actually depend on the jam, preventing their getting untimely dispatched and ensuring they stay in the RS for as long as the jam is still being executed. Table A.14 represents an instance of our RS size experiment for $P = 4$.

The jams will interestingly only execute in-order (despite their uops never depending on one another) due to the RS always selecting the *oldest uop for which*

input operands are ready for each port: as *VSQRTPD* uops can only get dispatched on P0 and have the same unmodified input register, they will necessarily execute in-order. This makes it impossible for the last uop from a particular jam to be dispatched before its predecessors (preventing payload uops to get dispatched before *all* previous jam uops are completely executed), and also prevents jams from later iterations to interfere with the current one. The question is then: how many uops can fit in the RS before preventing the visibility of the next iteration's jam?

As before, we use 5 *VSQRTPD* instructions to create a $5 * 21 = 105$-cycle jam on SNB ($5 * 14 = 70$ for IVB/HSW). We use *VADDPS* instructions in the payload, which have a throughput of 1 per cycle, meaning we can dispatch $105 * 1 = 105$ (resp. $70 * 1 = 70$) payload instructions in parallel with the *the next* jam without impacting performance (provided the RS is large enough).

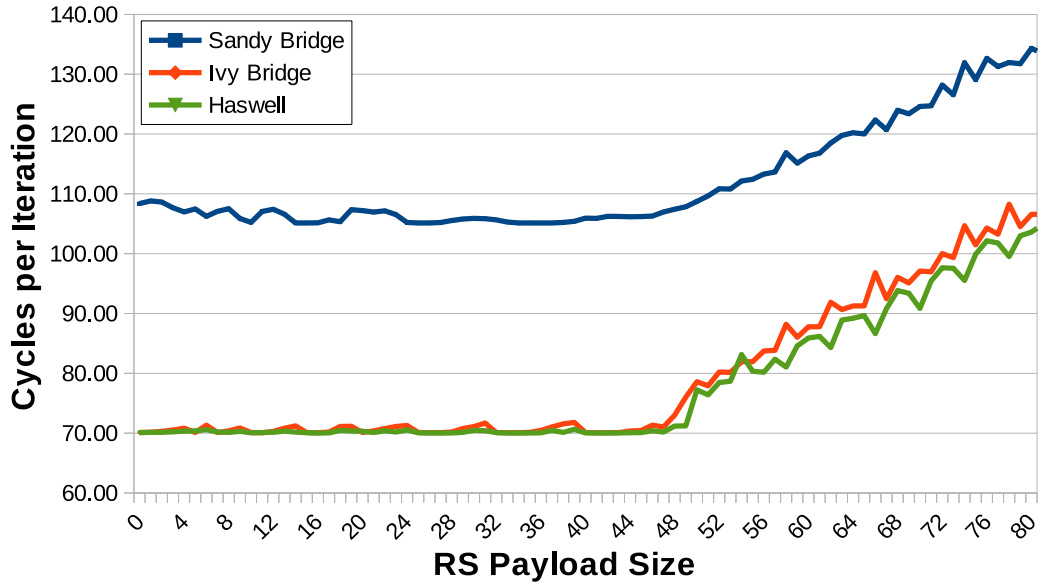


Figure A.7: Quantifying RS Entries

The transition from enough *to* not enough *RS entries is interestingly much smoother for SNB than for IVB and HSW.*

Figure A.7 shows results for our measurement experiments on the RS. Here, control instructions are not a problem as they do not depend on the jam and can thus be executed out-of-order. The number of entries occupying the RS between two jams is then exactly the number of instructions in the payload.

Table A.15: RS Size: Measured vs. Official

Uarch	Measured	Official	Difference
SNB	48	54	6
IVB	[49-51]	54	3
HSW	51	60	9

The mismatch is more important on SNB than on IVB, with a drop of 11% vs. 6%. It increases back on HSW with 15%.

The origin of these gaps is unknown.

Table A.15 shows the measured sizes vs. the one we would expect. The difference

is small on SNB and IVB (respectively 6 and 3), but it gets annoyingly high with HSW (9) as the RS is of limited size in the first place.

A.7 Quantifying Store Buffer Entries

The SB keeps track of all store uops from the time they were issued until they are retired. It is complementary to the LB in terms of preventing illegal access orders, but also holds the stored values as they are only committed to memory at retirement.

Table A.16: RQ Experiment Example for the SB (P = 4)

#Line	Instruction	Purpose
1	VSQRTPD %xmm0, %xmm1	Jamming retirement
...	VSQRTPD %xmm0, %xmm1	Jamming retirement
5	VSQRTPD %xmm0, %xmm1	Jamming retirement
6	VMOVUPS %xmm3, 64(%rsi)	Payload
7	VMOVUPS %xmm3, 64(%rsi)	Payload
8	VMOVUPS %xmm3, 64(%rsi)	Payload
9	VMOVUPS %xmm3, 64(%rsi)	Payload
10	SUB $1, %rdi	Loop Control
11	JG .LOOP	Loop Control

The jam comprises 5 VSQRTPD *instructions. The payload consists of vector store instructions with a throughput of 1 instruction per cycle.*

We use a light jam of 5 *VSQRTPD* instructions, as the expected SB size is rather small. The payload comprises *VMOVUPS* store instructions, with a throughput of 1 store uop per cycle. As with the RS, it means we can dispatch up to 105 store uops in SNB (70 for IVB and HSW) without stores become lengthier than square root operations.

Figure A.8 shows our experimental results for the SB size. They are very clear cut, as were the ones for the LB.

Table A.17: SB Size: Measured vs. Official

Uarch	Measured	Official	Difference
SNB	36	36	0
IVB	36	36	0
HSW	42	42	0

The measured SB sizes match the official ones perfectly.

Table A.17 shows the measured and the expected sizes match perfectly, revealing that the LB and the SB behave in a similar fashion here too.

A.8 Impact of Microfusion on Resource Consumption

Microfusion is an interesting case of hardware optimization, keeping uops needing to be dispatched to different functional units as one for a large part of the pipeline. This

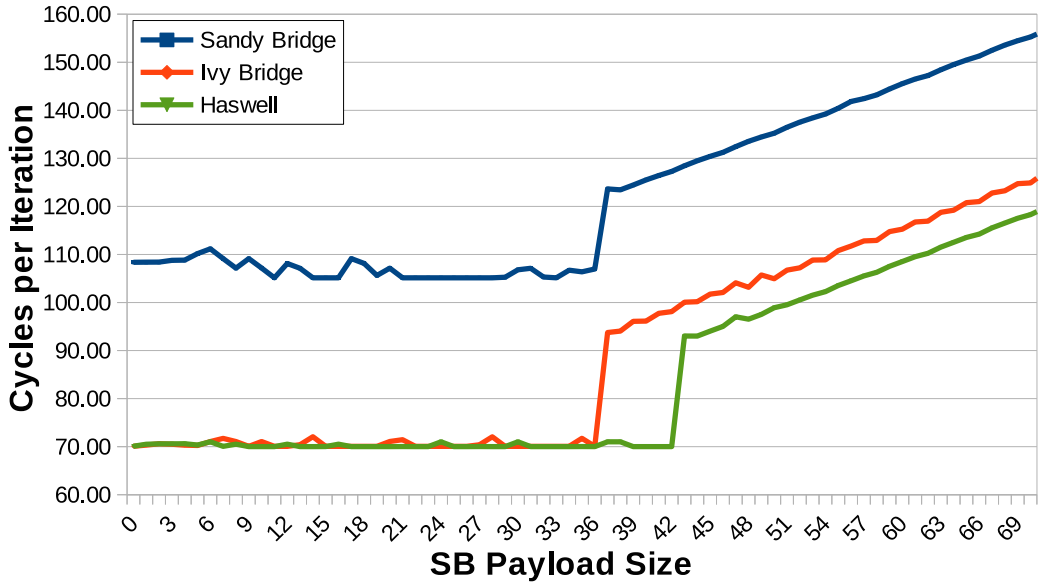


Figure A.8: Quantifying SB Entries

The transition points due to SB entries scarcity is very clear on all 3 studied microarchitectures: 37 for SNB and IVB, and 43 for HSW.

happens for stores, as well as for instructions having both data load and arithmetic components, such as *MULPD (%rax), %xmm1*.

We saw in Chapter 3 that in *unlamination* cases, the splitting of microfused uops was done in the uop queue, i.e. before getting in the RAT. Here, we will try to find out when it is done in other cases, so as to determine how many entries they should be expected to consume in each out-of-order resource.

A.8.1 ROB Microfusion

We try to single out the resource consumption of microfused uops in terms of ROB entries by using a special preamble. We use 15 *VSQRTPD*s for the usual purpose of jamming retirement, and add 20 *vector load [simple address] + vector add* instructions to put microfused uops in the ROB. The payload consists of a varying numbers of *NOP*s, as with the ROB resource-quantifying experiment. Table A.18 represents such an experiment for $P = 4$.

We can then evaluate whether not-unlaminated microfused uops remain fused in the ROB by comparing the results from these experiments (in Figure A.9) with those from the regular ROB experiment. We roughly expect a shift of 20 entries if they do remain fused, or 40 if they do not. Table A.19 shows that in SNB, microfused uops seem to consume 2 distinct entries in the ROB, but that they only take 1 in IVB and HSW. This improvement increases the effective ROB size when microfused operations are used.

A.8.2 RS Microfusion

We adopt the same strategy for determining the status of microfusion in the RS as the one used for the ROB. The experiments are based on the ones used for finding the size of the RS, except that an extra "base load" (consisting of 15 instructions whose uops are *still microfused in the uop queue*) is added, and the jam is adjusted

Table A.18: Microfusion Evaluation Experiment for the ROB (P = 4)

#Line	Instruction	Purpose
1	VSQRTPD %xmm0, %xmm1	Jamming retirement
...	VSQRTPD %xmm0, %xmm1	Jamming retirement
15	VSQRTPD %xmm0, %xmm1	Jamming retirement
16	VADDPS 64(%r12), %xmm0, %xmm2	Base Load
...	VADDPS 64(%r12), %xmm0, %xmm2	Base Load
35	VADDPS 64(%r12), %xmm0, %xmm2	Base Load
36	NOP	Payload
37	NOP	Payload
38	NOP	Payload
39	NOP	Payload
40	SUB $1, %rdi	Loop Control
41	JG .LOOP	Loop Control

The jam comprises 15 VSQRTPD *instructions. The base experiment consists consists of (vector load + addition) instructions with a throughput of 1 instruction per cycle. The payload then adjusts the ROB stress with simple NOPs.*

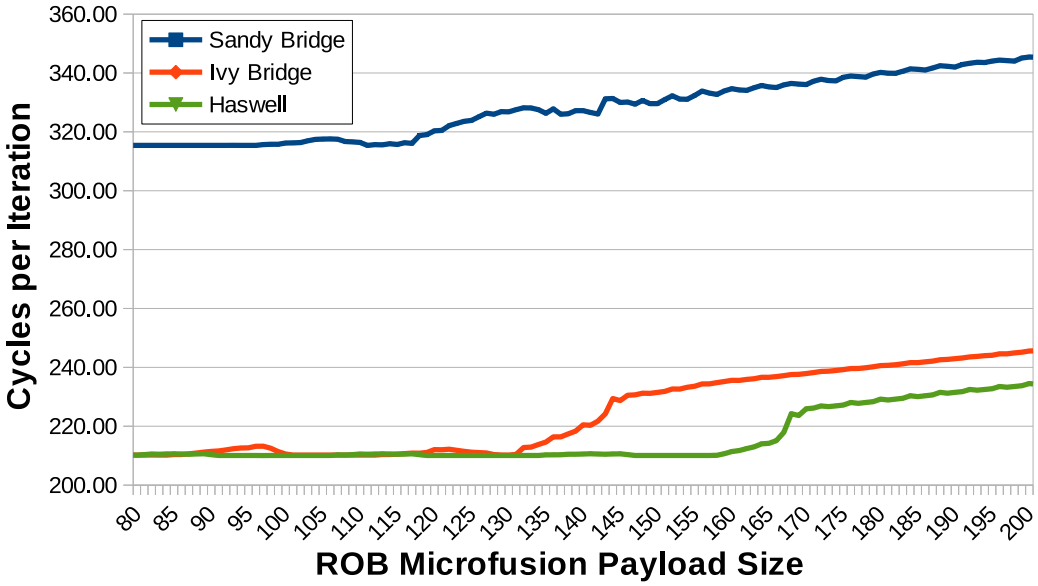


Figure A.9: ROB Microfusion Evaluation

The performance drop is shifted on the left (compared to Figure A.6) as expected, due to the extra instructions in the preamble.

so that load operations indirectly depend on it. An example for $P = 4$ is provided in Table A.20.

Figure A.10 shows the results of these adjusted experiments, with performance breakpoints occurring noticeably earlier than in the original curves (from Figure A.7). The following elements should be considered when evaluating this shift:

1. There are 2 extra entries taken in the RS due to the extra instructions in the jam.

Table A.19: ROB Microfusion Evaluation

Uarch	New Bump	Original Bump	Difference	Fusion
SNB	119	[158-164]	{39, 45}	No
IVB	[133-145]	[154-167]	{21, 22}	Yes
HSW	[161-169]	[184-191]	{23, 22}	Yes

The New Bump *column indicates the position of the performance break point in the ROB Microfusion experiments in terms of P.* Original Bump *does the same for the ROB Size experiments. The* Difference *is the subtraction of* New Bump *from* Original Bump *(or of the matching mins and maxes when relevant).*

The expected difference in case microfused operations takes 2 slots in the ROB is 40, and 20 if they only take 1. Here, it is clear SNB is in former case, and IVB and HSW in the latter.

Table A.20: Microfusion Evaluation Experiment for the RS (P = 4)

#Line	Instruction	Purpose
1	VSQRTPD %xmm0, %xmm1	Jamming dispatch
...	VSQRTPD %xmm0, %xmm1	Jamming dispatch
5	VSQRTPD %xmm0, %xmm1	Jamming dispatch
6	VMOVMSKPD %xmm1, %r13	Jamming dispatch
7	ADD %r13, %r12	Jamming dispatch
8	VADDPS 64(%r12), %xmm1, %xmm2	Base Load
...	VADDPS 64(%r12), %xmm1, %xmm2	Base Load
23	VADDPS 64(%r12), %xmm1, %xmm2	Base Load
24	VADDPS %xmm1, %xmm1, %xmm2	Payload
25	VADDPS %xmm1, %xmm1, %xmm2	Payload
29	VADDPS %xmm1, %xmm1, %xmm2	Payload
27	VADDPS %xmm1, %xmm1, %xmm2	Payload
28	SUB $1, %rdi	Loop Control
29	JG .LOOP	Loop Control

The jam comprises 5 VSQRTPD *instructions, as well as a* VMOVMSKPD *and an integer* ADD*. The base experiment consists of (vector load + addition) instructions with a throughput of 1 instruction per cycle. The payload instructions (vector additions) have an overall throughput of 1 per cycle. Each of the uops in both the base load and the payload depend on the jam: FP additions depend on the last* VSQRTPD*, while address calculations depend on the* ADD*.*

Increasing the size of the payload allows us to detect how much the microfused instructions from the base load really take inside the RS.

2. If microfused uops take a single entry in the RS, then 15 extra other slots would be used by the new base load.

3. If not, then 30 extra slots will be taken.

Hence, if the shift is of $15 + 2 = 17$, then microfused uops take a single slot. If it is $30 + 2 = 32$, then microfused uops are split in the RS and need 2 slots.

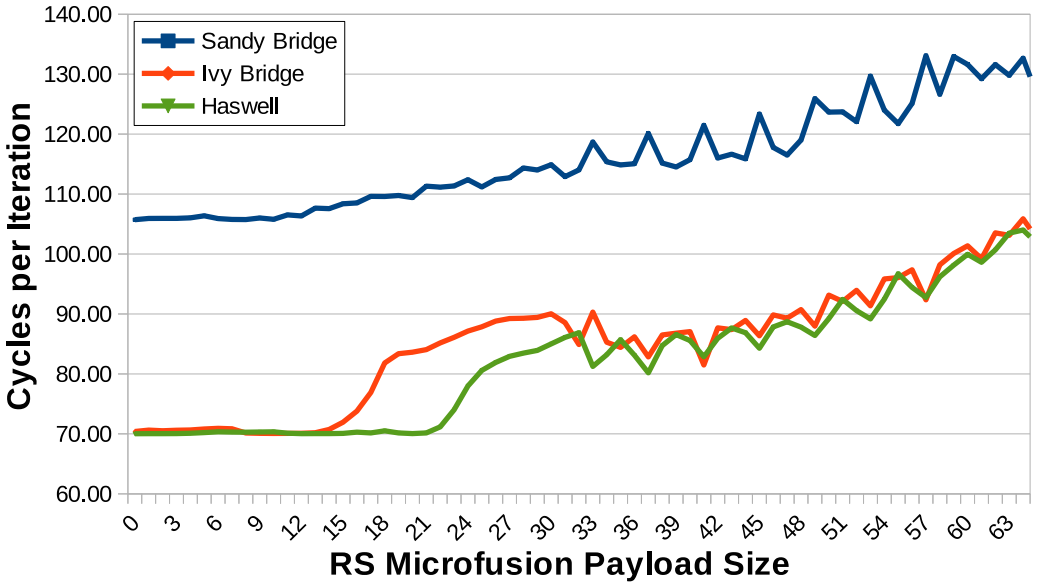


Figure A.10: RS Microfusion Evaluation

The performance drops gets shifted on the left compared to the regular RS size experiments (from Figure A.7) due to a) the extra instructions in the jam and b) the base load.

Table A.21: RS Microfusion Evaluation

Uarch	New Bump	Original Bump	Difference	Fusion
SNB	14	48	{34}	No
IVB	[16-20]	[49-51]	{33-31}	No
HSW	[23-26]	51	{28,25}	No

The New Bump *column indicates the position of the performance break point in the RS Microfusion experiments in terms of P.* Original Bump *does the same for the RS Size experiments. The* Difference *is the subtraction of* New Bump *from* Original Bump *(or of the matching mins and maxes when relevant).*

In all the cases, the Difference *is closer to 32 than it is to 17, indicating uops are not microfused in the RS in any of the studied microarchitectures. However, the result for HSW is not as clear cut as for SNB and IVB, which combined to the important offset in the measured size of the RS vs. the official one (see Table A.15) suggests the full capacity of the RS might only be achievable when microfused uops are used.*

Table A.21 provides a detailed comparison between the curves, and shows that microfused uops take 2 entries in the RS on all studied microarchitectures.

Appendix B

Note on the Load Matrix

This section was placed in a separate appendix as unlike Appendix A, its results were not used throughout the rest of the manuscript. However, they will be very relevant in future work, especially when trying to use UFS on loops operating beyond L1.

B.1 Load Matrix Presentation

The Load Matrix (LM) keeps track of all load uops from the time they were issued until they are completed [140]. It is not clear what its role exactly is, but it probably works hand-in-hand with the LB to prevent read-write conflicts between load and store uops.

B.2 Quantifying Load Matrix Entries

The experiment used to quantify LM entries is a variation of the one used to quantify RS entries (in Section A.6), except the payload is made entirely of varying numbers of load instructions.

Instruction "*VMOVSS mem, xmm*" was chosen due to having a high reciprocal throughput of 2 on all target microarchitectures. Furthermore, we used 5 *VSQRTPD* instructions to create a $5 * 21 = 105$-cycle jam on SNB ($5 * 14 = 70$ for IVB/HSW), leaving enough time for around $105 * 4 = 420$ (resp. $70 * 4 = 280$) uops to be issued during its execution.

Table B.1: Resource Quantifying Experiment Example for the LM (P = 2)

#Line	Instruction	Purpose
1	VSQRTPD %xmm0, %xmm1	Jamming dispatch
...	VSQRTPD %xmm0, %xmm1	Jamming dispatch
5	VSQRTPD %xmm0, %xmm1	Jamming dispatch
6	VMOVMSKPD %xmm1, %r13	Jamming dispatch
7	ADD %r13, %r12	Jamming dispatch
8	VMOVSS (%r12), %xmm2	Base Load
9	VMOVSS (%r12), %xmm2	Base Load
10	SUB $1, %rdi	Loop Control
11	JG .LOOP	Loop Control

This is the LM Quantifying Experience's assembly code for 2 payload instructions. In this case, the payload consists of register loads, each of which needing 1 LM entry to be issued (and until they are dispatched and completed).

An example assembly code for $P = 2$ is presented in Table B.1.

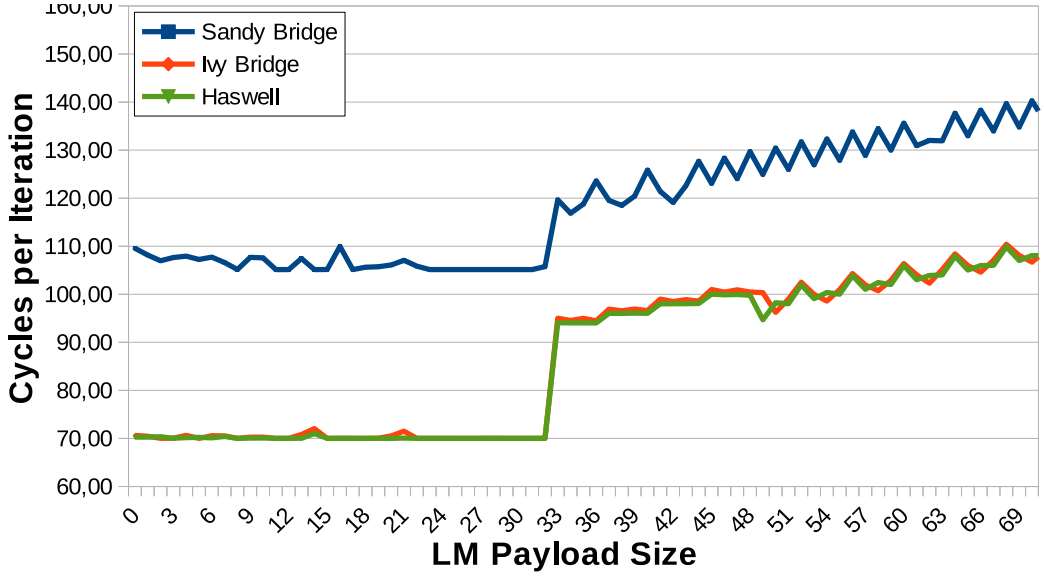


Figure B.1: Quantifying Load Matrix Entries

The performance drops significantly on point 33 for SNB, IVB and HSW. This suggests they all have 32 LM entries.

We can also observe IVB and HSW's curves are smoother than SNB's, suggesting better resource allocation / reclaiming mechanisms were implemented.

Results for varying payload sizes are shown in Figure B.1. The result from the experiment described in Table B.1 is presented on coordinate X = 2. A performance degradation appears for point 33 in all microarchitectures, showing that the 33rd load was the one too many in all cases. This strongly suggests that the LM has 32 entries in SNB, IVB and HSW.

Table B.2: LM Size: Measured vs. Official

Uarch	Measured	Official
SNB	32	?
IVB	32	?
HSW	32	?

All studied microarchitectures appear to have 32 LM entries. We could not find official numbers to compare these results with.

We could not find any official figures to compare these numbers with (See Table B.2).

Bibliography

[1] G. E. Moore *et al.*, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998. 1

[2] R. R. Schaller, "Moore's law: past, present and future," *Spectrum, IEEE*, vol. 34, no. 6, pp. 52–59, 1997. 1

[3] TOP500 project, "Top #1 systems," http://top500.org/featured/top-systems, 2015. 2

[4] Intel, "2.2: Intel microarchitecture code name sandy bridge," *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Sep. 2014. 5

[5] ——, "2.2.7: Intel microarchitecture code name ivy bridge," *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Sep. 2014. 5

[6] ——, "2.1: The haswell microarchitecture," *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Sep. 2014. 5

[7] ——, "Product specifications," http://ark.intel.com. 5

[8] P. Taylor, "Baytrail uncore performance monitoring events," Apr. 2014. [Online]. Available: https://software.intel.com/en-us/articles/baytrail-uncore-performance-monitoring-events 5

[9] A. L. Shimpi, "Intel's sandy bridge architecture exposed," http://www.anandtech.com/show/3922/intels-sandy-bridge-architecture-exposed. 5

[10] ——, "Intel's ivy bridge architecture exposed," http://www.anandtech.com/show/4830/intels-ivy-bridge-architecture-exposed. 5

[11] ——, "Intel's haswell architecture analyzed: Building a new pc and a new intel," http://www.anandtech.com/show/6355/intels-haswell-architecture. 5

[12] ——, "Intel's silvermont architecture revealed: Getting serious about mobile," http://www.anandtech.com/show/6936/intels-silvermont-architecture-revealed-getting-serious-about-mobile. 5

[13] D. Kanter, "Intel's sandy bridge microarchitecture," http://www.realworldtech.com/sandy-bridge/7/. 5

[14] ——, "Intel's haswell cpu microarchitecture," http://www.realworldtech.com/haswell-cpu/. 5

[15] ——, "Silvermont, intel's low power architecture," http://www.realworldtech.com/silvermont/. 5

[16] A. Fog, "The microarchitecture of intel, amd and via cpus/an optimization guide for assembly programmers and compiler makers," 2014. 5

[17] S. R. Shenoy and A. Daniel, "Intel architecture and silicon cadence: The catalyst for industry innovation," *Technology at Intel Magazine*, 2006. 5, 132

[18] H. Wong, "Intel ivy bridge cache replacement policy," Jan. 2013. [Online]. Available: http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/ 10

[19] S. Raikin, D. J. Sager, Z. Sperber, E. Krimer, O. Lempel, S. Shwartsman, A. Yoaz, and O. Golz, "Tracking mechanism coupled to retirement in reorder buffer for indicating sharing logical registers of physical register in record indexed by logical register," Dec. 16 2014, uS Patent 8,914,617. 10

[20] B. Kuttana, "Technology insight: Intel silvermont," 2013. [Online]. Available: https://software.intel.com/sites/default/files/managed/bb/2c/02_Intel_Silvermont_Microarchitecture.pdf 13

[21] E. Oseret *et al.*, "CQA: A code quality analyzer tool at binary level," *HiPC '14*, 2014. 15, 32, 44, 62, 88, 95, 128

[22] MAQAO, "Maqao project," http://www.maqao.org, 2013. 15, 18, 44, 47

[23] J. Muir, "Using the rdtsc instruction for performance monitoring," Technical report, Intel Corporation, Tech. Rep., 1997. 17, 29

[24] B. Sprunt, "The basics of performance-monitoring hardware," *IEEE Micro*, no. 4, pp. 64–71, 2002. 17

[25] D. Zaparanuks, M. Jovic, and M. Hauswirth, "Accuracy of performance counter measurements," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on.* IEEE, 2009, pp. 23–32. 17

[26] Intel, "Profile function or loop execution time," *Intel C++ Compiler XE 13.1 User and Reference Guides.* [Online]. Available: https://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/GUID-96F454BF-364A-40C9-9B55-BFFAA8FD171D.htm 17

[27] ——, "Intel vtune amplifier xe," www.intel.com/software/products/vtune, 2013. 18, 41, 43, 58

[28] A. C. de Melo, "The new linux perf tools," in *Slides from Linux Kongress*, 2010. 18

[29] A. S. Charif-Rubial, "Maqao performance analysis and optimization tool," http://www.vi-hps.org/upload/material/tw17/MAQAO.pdf, 2015. 18

[30] Q. Wu and O. Mencer, "Evaluating sampling based hotspot detection," in *Architecture of Computing Systems–ARCS 2009.* Springer, 2009, pp. 28–39. 18

[31] J. Levon and P. Elie, "Oprofile: A system profiler for linux," http://oprofile.sourceforge.net, 2013. 18, 58

[32] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra, "Using papi for hardware performance monitoring on linux systems," in *Proc. Conf. on Linux Clusters: The HPC Revolution*, 2001, pp. 25–27. 18

[33] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on.* IEEE, 2010, pp. 207–216. 18, 30

[34] A. Yasin, "A top-down method for performance analysis and counters architecture," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on.* IEEE, 2014, pp. 35–44. 18

[35] D. Levinthal, "Performance analysis guide for intel core i7 processor and intel xeon 5500 processors," *Intel Performance Analysis Guide*, vol. 30, 2009. 18

[36] P. Calafiura, S. Eranian, D. Levinthal, S. Kama, and R. A. Vitillo, "Gooda: The generic optimization data analyzer," in *Journal of Physics: Conference Series*, vol. 396, no. 5. IOP Publishing, 2012, p. 052072. 18

[37] S. Koliaï, Z. Bendifallah, M. Tribalat, C. Valensi, J.-T. Acquaviva, and W. Jalby, "Quantifying performance bottleneck cost through differential analysis," in *Proceedings of the 27th international ACM conference on supercomputing.* ACM, 2013, pp. 263–272. 18, 29, 32, 44, 53, 63, 87, 88, 123

[38] C. Valensi, "Madras: Multi-architecture binary rewriting tool technical report," 2013. 18

[39] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002. 19

[40] V. S. Pai, P. Ranganathan, and S. V. Adve, "Rsim: An execution-driven simulator for ilp-based shared-memory multiprocessors and uniprocessors," in *Proceedings of the Third Workshop on Computer Architecture Education*, vol. 178. Citeseer, 1997. 19

[41] B. Cmelik and D. Keppel, *Shade: A fast instruction-set simulator for execution profiling.* Springer, 1995. 19

[42] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002. 19

[43] A. Patel, F. Afram, S. Chen, and K. Ghose, "Marss: a full system simulator for multicore x86 cpus," in *Proceedings of the 48th Design Automation Conference.* ACM, 2011, pp. 1050–1055. 19

[44] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson *et al.*, "Mambo: a full system simulator for the powerpc architecture," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 4, pp. 8–12, 2004. 19

[45] 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.* IEEE, 2007, pp. 23–34. 20

[46] G. H. Loh, S. Subramaniam, and Y. Xie, "Zesto: A cycle-level simulator for highly detailed microarchitecture exploration," in *Performance Analysis of*

Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on. IEEE, 2009, pp. 53–64. 20, 128

[47] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1. ACM, 2003, pp. 318–319. 20

[48] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sampled simulation of multi-threaded applications," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on.* IEEE, 2013, pp. 2–12. 20

[49] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators," in *Proceedings of the 40th Annual IEEE/ACM international Symposium on Microarchitecture.* IEEE Computer Society, 2007, pp. 249–261. 20

[50] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanović, "Ramp gold: an fpga-based architecture simulator for multiprocessors," in *Proceedings of the 47th Design Automation Conference.* ACM, 2010, pp. 463–468. 20

[51] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation: Raising the level of abstraction in architectural simulation," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on.* IEEE, 2010, pp. 1–12. 20

[52] N. Nethercote, "Cachegrind," http://valgrind.org/docs/manual/cg-manual.html. 20

[53] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob, "Cmp$im: A pin-based on-the-fly multi-core cache simulator," in *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA*, 2008, pp. 28–36. 20

[54] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, "Using a codelet program execution model for exascale machines: position paper," in *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era.* ACM, 2011, pp. 64–69. 21

[55] M. C. Easton and R. Fagin, "Cold-start vs. warm-start miss ratios," *Communications of the ACM*, vol. 21, no. 10, pp. 866–872, 1978. 22, 31

[56] J. McCalpin, "Stream benchmark," *Link: www. cs. virginia. edu/stream/ref. html# what*, 1995. 22

[57] L. W. McVoy, C. Staelin *et al.*, "lmbench: Portable tools for performance analysis." in *USENIX annual technical conference.* San Diego, CA, USA, 1996, pp. 279–294. 22

[58] D. Callahan, J. Dongarra, and D. Levine, "Vectorizing compilers: A test suite and results," in *Proceedings of the 1988 ACM/IEEE conference on Supercomputing.* IEEE Computer Society Press, 1988, pp. 98–105. 22

[59] S. Maleki *et al.*, "An evaluation of vectorizing compilers," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '11, 2011, pp. 372–382. [Online]. Available: http://dx.doi.org/10.1109/PACT.2011.68 22, 79, 92, 93, 95, 118

[60] J. C. Beyler, N. Triquenaux, V. Palomares, F. Chabane, T. Fighiera, J.-P. Halimi, and W. Jalby, "Microtools: Automating program generation and performance measurement," in *ICPPW, 2012*. IEEE, 2012, pp. 424–433. 22, 41, 44

[61] H. Wong, "Measuring reorder buffer capacity," May 2013. [Online]. Available: http://blog.stuffedcow.net/2013/05/measuring-rob-capacity/ 23, 106, 111, 133, 140

[62] C. Akel, Y. Kashnikov, P. de Oliveira Castro, and W. Jalby, "Is source-code isolation viable for performance characterization?" in *Parallel Processing (ICPP), 2013 42nd International Conference on*. IEEE, 2013, pp. 977–984. 23

[63] Y.-J. Lee and M. Hall, "A code isolator: Isolating code fragments from large programs," in *Languages and Compilers for High Performance Computing*. Springer, 2005, pp. 164–178. 23

[64] E. Petit, G. Papaure, F. Dru, and F. Bodin, "Astex: a hot path based thread extractor for distributed memory system on a chip," in *High-Performance Embedded Architecture and Compilation Industrial Workshop (HiPEAC)*, 2006. 23, 26, 27

[65] "Codelet finder." [Online]. Available: https://runtime.bordeaux.inria.fr/prohmpt/Meetings/r01/anr-08-cosi-013-02_r01_caps_entreprise.pdf 23

[66] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2001, pp. 3–14. 23

[67] P. D. O. Castro, C. Akel, E. Petit, M. Popov, and W. Jalby, "Cere: Llvm-based codelet extractor and replayer for piecewise benchmarking and optimization," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 1, p. 6, 2015. 23

[68] M. Popov, C. Akel, F. Conti, W. Jalby, and P. d. O. Castro, "Pcere: Fine-grained parallel benchmark decomposition for scalability prediction," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 2015, pp. 1151–1160. 24

[69] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, "Numerical recipes: The art of scientific computing," 1992. 26, 28, 90, 93, 95, 118

[70] J. Noudohouenou, V. Palomares, W. Jalby, D. C. Wong, D. J. Kuck, and J. C. Beyler, "Simsys: A performance simulation framework," in *Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, ser. RAPIDO '13. ACM, 2013. 28, 61, 88, 90, 95, 129, 131

[71] J. Noudohouenou, "Performance prediction based on codelet driven application characterization," Ph.D. dissertation, Versailles-Saint-Quentin-en-Yvelines, 2013. 28

[72] M. H. Jamal and A. Waheed, "Precise measurement of execution time of concurrent, symmetric, and short tasks," in *Int. CMG Conference*, 2008, pp. 149–160. 29

[73] R. Dementiev, "Monitoring integrated memory controller requests in the 2nd, 3rd and 4th generation intel core processors," https://software.intel.com/en-us/articles/monitoring-integrated-memory-controller-requests-in-the-2nd-3rd-and-4th-generation-intel. 30, 72

[74] Intel, "1.5: Uncore pmu summary tables," *Intel Xeon Processor E5 v2 and E7 v2 Product Families Uncore Performance Monitoring Reference Manual*, Feb. 2014. 30

[75] S. Laha, J. H. Patel, and R. K. Iyer, "Accurate low-cost methods for performance evaluation of cache memory systems," *Computers, IEEE Transactions on*, vol. 37, no. 11, pp. 1325–1336, 1988. 31

[76] R. Jain, "Techniques for experimental design, measurement, simulation, and modeling, 1991," 1991. 31

[77] D. J. Lilja, *Measuring computer performance: a practitioner's guide*. Cambridge University Press, 2000. 31

[78] S. Touati, "Towards a statistical methodology to evaluate program speedups and their optimisation techniques," *arXiv preprint arXiv:0902.1035*, 2009. 31

[79] C.-h. Hsu and W.-c. Feng, "A power-aware run-time system for high-performance computing," in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, ser. SC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 1–. [Online]. Available: http://dx.doi.org/10.1109/SC.2005.3 35

[80] K. Livingston, N. Triquenaux, T. Fighiera, J. C. Beyler, and W. Jalby, "Computer using too much power? give it a rest (runtime energy saving technology)," *Computer Science-Research and Development*, vol. 29, no. 2, pp. 123–130, 2014. 35

[81] M. Horowitz, T. Indermaur, and R. Gonzalez, "Low-power digital design," in *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*, Oct 1994, pp. 8–11. 35

[82] T. Mudge, "Power: A first class design constraint for future architectures," in *High Performance Computing - HiPC 2000*. Springer, 2000, pp. 215–224. 35

[83] D. Brodowski, "Linux kernel cpufreq subsystem," *URL http://www. kernel. org/pub/linux/utils/kernel/cpufreq/cpufreq. html*. 35

[84] F. Talbart *et al.*, "Codelet tuning infrastructure," https://github.com/franck-talbart/codelet_tuning_infrastructure, 2015. 41

[85] Z. Bendifallah, W. Jalby, J. Noudohouenou, E. Oseret, V. Palomares, and A. C. Rubial, "Pamda: Performance assessment using maqao toolset and differential analysis," *Tools for High Performance Computing 2013*, pp. 107–127, 2014. 43, 131

[86] S. S. Shende and A. D. Malony, "The tau parallel performance system," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 287–311, May 2006. [Online]. Available: http://dx.doi.org/10.1177/1094342006064482 43, 58

[87] M. Burtscher, B.-D. Kim, J. R. Diamond, J. D. McCalpin, L. Koesterke, and J. C. Browne, "Perfexpert: An easy-to-use performance diagnosis tool for hpc applications." in *SC*. IEEE, 2010, pp. 1–11. 43, 58

[88] Acumem, "Acumem threadspotter," http://www.paratools.com/threadspotter. 43

[89] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahàm, D. Becker, B. Mohr, and F. JÃ$\frac{1}{4}$lich, "The scalasca performance toolset architecture," in *STHEC*, 2008. 43, 58

[90] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "Vampir: Visualization and analysis of mpi resources," *Supercomputer*, vol. 12, pp. 69–80, 1996. 43, 58

[91] D. Barthou, A. C. Rubial, W. Jalby, S. Koliai, and C. Valensi, "Performance tuning of x86 openmp codes with maqao," in *Parallel Tools Workshop*. Dresden, Germany: Springer-Verlag, sep 2009. 44, 47

[92] S. Koliai, S. Zuckerman, E. Oseret, M. Ivascot, T. Moseley, D. Quang, and W. Jalby, "A balanced approach to application performance tuning," in *LCPC*, 2009, pp. 111–125. 44, 47

[93] A. S. Charif-Rubial, "On code performance analysis and optimisation for multicore architectures," Ph.D. dissertation, Oct. 2012. [Online]. Available: http://tel.archives-ouvertes.fr/tel-00842601 44

[94] F. Real, M. Trumm, V. Vallet, B. Schimmelpfennig, M. Masella, and J.-P. Flament, "Quantum Chemical and Molecular Dynamics Study of the Coordination of Th(IV) in Aqueous Solvent," *J. Phys. Chem. B*, vol. 114, no. 48, pp. 15 913–15 924, 2010. [Online]. Available: http://dx.doi.org/10.1021/jp108061s 45

[95] C. Staelin and H. packard Laboratories, "lmbench: Portable tools for performance analysis," in *USENIX Annual Technical Conference*, 1996, pp. 279–294. 48

[96] J. Liu, W. Yu, J. Wu, D. Buntinas, S. Kini, D. K, and P. Wyckoff, "Microbenchmark performance comparison of high-speed cluster interconnects," *IEEE Micro*, 2004. 48

[97] S. R. Alam, R. F. Barrett, J. A. Kuehn, P. C. Roth, and J. S. Vetter, "Characterization of scientific workloads on systems with multi-core processors," in *IISWC*, 2006, pp. 225–236. 48

[98] A. Charif-Rubial *et al.*, "MIL: A language to build program analysis tools..." ser. HiPC, 2013. 50, 90

[99] E. Baysal, D. Kosloff, and J. Sherwood, "Reverse time migration:geophysics," 1983. 56

[100] Gprof, "The gnu profiler," http://sourceware.org/binutils/docs-2.18/gprof/index.html, 2013. 58

[101] M. Martonosi, A. Gupta, and T. Anderson, "Memspy: Analyzing memory system bottlenecks in programs," in *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, 1992, pp. 1–12. 58

[102] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: tools for performance analysis of optimized parallel programs http://hpctoolkit.org," *Concurr. Comput. : Pract. Exper.*, vol. 22, no. 6, pp. 685–701, Apr. 2010. [Online]. Available: http://dx.doi.org/10.1002/cpe.v22:6 58

[103] O. Sopeju, M. Burtscher, A. Rane, and J. Browne, "Autoscope: Automatic suggestions for code optimizations using perfexpert," in *2011 ICPDPTA*, Jul. 2011, pp. 19–25. 58

[104] W. Yoo, K. Larson, L. Baugh, S. Kim, and R. H. Campbell, "Adp: automated diagnosis of performance pathologies using hardware events." in *SIGMETRICS*, P. G. Harrison, M. F. Arlitt, and G. Casale, Eds. ACM, 2012, pp. 283–294. [Online]. Available: http://dblp.uni-trier.de/db/conf/sigmetrics/sigmetrics2012.html#YooLBKC12 58

[105] W. Yoo, K. Larson, S. Kim, W. Ahn, R. H. Campbell, and L. Baugh, "Automated fingerprinting of performance pathologies using performance monitoring units (pmus)," in *3rd USENIX Workshop on Hot Topics in Parallelism (HotPar '11)*, USENIX. Berkeley, CA: USENIX, 05/2011 2011. 58

[106] Intel Corporation, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, Sept. 2014, no. 248966-030. 62, 89

[107] A. Fog, "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus," Mar. 2015. [Online]. Available: http://www.agner.org/optimize/instruction_tables.pdf 62, 64, 105, 107

[108] Intel, "2.3.3: Execution core (operations with data-dependant latencies)," *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Sep. 2014. 65

[109] ——, "2.2.2.1: Legacy decode pipeline - microfusion," *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Sep. 2014. 65

[110] ——, "2.2.2.4: Micro-op queue and the loop stream detector (LSD)," *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Sep. 2014. 66, 108

[111] ——, "2.2.2.2: Decoded icache," *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Sep. 2014. 67

[112] Intel Open Source Technology Center, "Perfmon," https://download.01.org/perfmon/, 2015. 72

[113] A. Arcangeli. (2010, Aug.) Transparent hugepage support. [Online]. Available: http://www.linux-kvm.org/images/9/9e/2010-forum-thp.pdf 73

[114] M. B. Rani Borkar and S. Jourdan, "Advancing moore's law on 2014 - broadwell converged core," Aug. 2014. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/presentation/advancing-moores-law-in-2014-presentation.pdf 73

[115] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ser. ISCA '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 338–. [Online]. Available: http://dl.acm.org/citation.cfm?id=998680.1006729 75

[116] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A mechanistic performance model for superscalar out-of-order processors," *ACM Trans. Comput. Syst.*, vol. 27, no. 2, pp. 3:1–3:37, May 2009. [Online]. Available: http://doi.acm.org/10.1145/1534909.1534910 75

[117] J. Treibig and G. Hager, "Introducing a performance model for bandwidth-limited loop kernels," in *Parallel Processing and Applied Mathematics*. Springer, 2010, pp. 615–624. 76

[118] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009. 76

[119] P. Joseph, K. Vaswani, and M. J. Thazhuthaveetil, "Construction and use of linear regression models for processor performance analysis," in *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*. IEEE, 2006, pp. 99–108. 77

[120] J. J. Yi, D. J. Lilja, and D. M. Hawkins, "A statistically rigorous approach for improving simulation methodology," in *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*. IEEE, 2003, pp. 281–291. 77

[121] D. C. Wong, V. Palomares, E. Oseret, Z. Bendifallah, M. Tribalat, W. Jalby, and D. J. Kuck, "Vp3: A vectorization potential performance prototype," ser. WPMVP '15, 2015. 79, 131

[122] J. M. Cebrián, L. Natvig, and J. C. Meyer, "Performance and energy impact of parallelization and vectorization techniques in modern microprocessors," *Computing*, 2013. 79

[123] "YALES2 public page." [Online]. Available: http://www.coria-cfd.fr/index.php/YALES2 83, 95, 123

[124] V. Moureau *et al.*, "From large-eddy simulation to direct numerical simulation of a lean premixed swirl flame: Filtered laminar flame-pdf modeling," *Combustion and Flame*, 2011. 83, 123

[125] F. Réal *et al.*, "Further insights in the ability of classical nonadditive potentials to model actinide ion-water interactions," *Journal of Computational Chemistry*, 2013. 85

[126] G. C. Evans *et al.*, "Vector seeker: A tool for finding vector potential," ser. WPMVP '14, 2014. 92

[127] J. Holewinski *et al.*, "Dynamic trace-based analysis of vectorization potential of applications," *SIGPLAN Not.*, 2012. 92

[128] A. Rane, R. Krishnaiyer, C. J. Newburn, J. Browne, L. Fialho, and Z. Matveev, "Unification of static and dynamic analyses to enable vectorization," pp. 367–381, 2014. 92

[129] C. Haine, O. Aumage, P. Enguerrand, and D. Barthou, "Exploring and evaluating array layout restructuration for SIMDization," 2014. 92

[130] "The AVBP code." [Online]. Available: http://www.cerfacs.fr/4-26334-The-AVBP-code.php 95, 124

[131] J. D. Little, "A proof for the queuing formula: L= λ w," *Operations research*, vol. 9, no. 3, pp. 383–387, 1961. 96

[132] D. H. Bailey, "Little's law and high performance computing," in *In RNR Technical Report*. Citeseer, 1997. 96

[133] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi, "Dynamically allocating processor resources between nearby and distant ILP," in *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*. IEEE, 2001, pp. 26–37. 101

[134] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*. IEEE, 2003, pp. 129–140. 101

[135] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark, "Branch prediction, instruction-window size, and cache size: Performance trade-offs and simulation techniques," *Computers, IEEE Transactions on*, vol. 48, no. 11, pp. 1260–1281, 1999. 102

[136] J. S. Griffith, S. R. Gupta, and G. J. Hinton, "Method and apparatus for binding instructions to dispatch ports of a reservation station," Nov. 18 1997, uS Patent 5,689,674. 102

[137] B. Sutanto, S. T. Srinivasan, M. C. Merten, C. Y. K. Lai, A. J. Christiansen, and J. M. Deinlein, "Method and apparatus for implementing dynamic port-binding within a reservation station," Jan. 1 2015, uS Patent 20,150,007,188. 103

[138] L. Djoudi, D. Barthou, O. Tomaz, A. Charif-Rubial, J.-T. Acquaviva, and W. Jalby, "The design and architecture of maqao profile: an instrumentation maqao module," in *Sixth Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology (EPIC-6) March 11, 2007 In*

conjunction with the IEEE/ACM International Symposium on Code Generation and Optimization, San Jose, CA. IEEE, 2007, p. 13. 107

[139] Intel, "2.2.2.1: Legacy decode pipeline - macro-fusion," *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Sep. 2014. 111

[140] C. Hewett. (2012) Back end memory bound. [Online]. Available: https://software.intel.com/en-us/forums/topic/328062 115, 151

[141] Intel, "2.2.4: The execution core," *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Sep. 2014. 115

[142] ——, "2.1.3: Execution engine," *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Sep. 2014. 115

[143] ——, "3.5.2.4: Partial register stalls," *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Sep. 2014. 115

[144] G. Paoloni, "How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures," *Intel Corporation, September*, 2010. 123

[145] "AVBP application progress." [Online]. Available: http://www.cerfacs.fr/~cfdbib/repository/TR_CFD_12_133.pdf 124

[146] Intel, "Intel architecture code analyzer (IACA)," Jun. 2012. [Online]. Available: https://software.intel.com/en-us/articles/intel-architecture-code-analyzer 128

[147] G. H. Loh, S. Subramaniam, and Y. Xie, "Zesto," Jan. 2009. [Online]. Available: http://zesto.cc.gatech.edu 128

[148] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," *ACM SIGARCH Computer Architecture News*, vol. 25, no. 3, pp. 13–25, 1997. 128

[149] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011. 128

[150] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 52. 128

[151] W. Heirman, T. Carlson, and L. Eeckhout, "Sniper: scalable and accurate parallel multi-core simulation," in *8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES-2012)*. High-Performance and Embedded Architecture and Compilation Network of Excellence (HiPEAC), 2012, pp. 91–94. 128