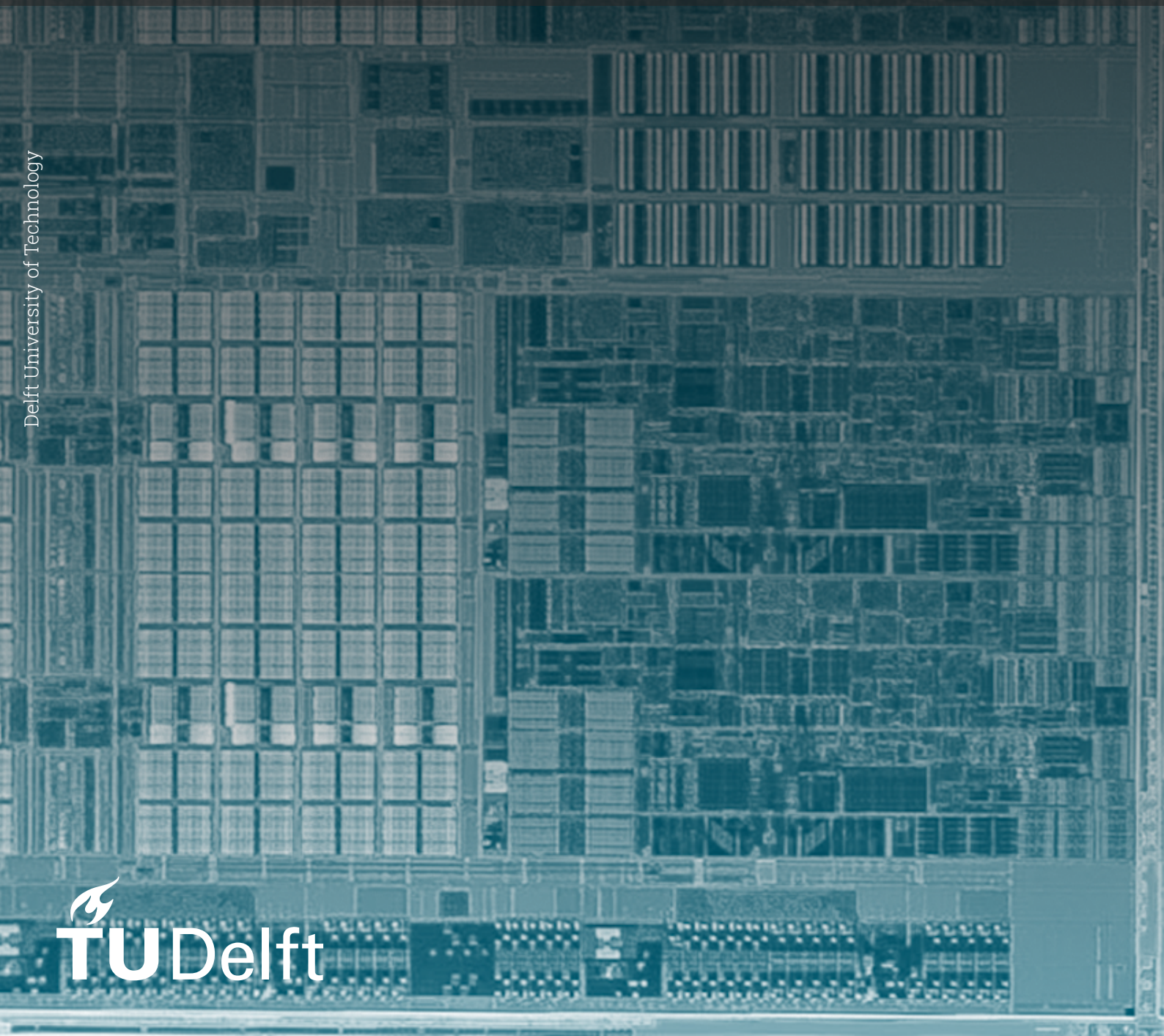
Failure analysis on Itanium

A comprehensive exploratory failure analysis study

CE4000: MSc Thesis

K.L. Geijsen



Delft University of Technology

TU Delft

Failure analysis on Itanium

A comprehensive exploratory failure analysis study

by

K.L. Geijsen

Student Name	Student Number
K.L. Geijsen	4892836

Instructor: prof. dr. ir. G.N. (Georgi) Gaydadjiev
Project Duration: November, 2023 - June, 2024
Faculty: Faculty of Electrical Engineering, Mathematics and Computer Science, Delft

Cover: Intel Itanium Poulson Die shot (Modified)
Style: TU Delft Report Style, with modifications

TU Delft

Failure analysis on Itanium

A comprehensive exploratory failure analysis study

Abstract - Itanium, a 64-bit microprocessor architecture jointly developed by Intel and Hewlett-Packard (HP), was launched in 2001 with the ambitious goal of replacing the dominant x86 architecture. Despite significant investment and technological innovation, Itanium failed to achieve widespread adoption and was discontinued in 2021. This thesis investigates the multifaceted reasons behind Itanium's commercial failure, exploring technical, historical, and business-strategic perspectives. Through a comprehensive literature review, automated sentiment analysis of public opinion, performance benchmarking, static analysis of compiled binaries, hardware complexity analysis, and differentiation strategic framework analysis this study identifies key factors that contributed to Itanium's lack of widespread market adoption.

The findings reveal that while Itanium offered technological advancements such as Explicitly Parallel Instruction Computing (EPIC) and a 64-bit architecture, it faced challenges in performance, compiler complexity, software compatibility, and market positioning. The emergence of competing architectures, notably AMD64, further eroded Itanium's potential market share. This thesis contributes to a deeper understanding of the complex interplay between technological innovation, market dynamics, and strategic decision-making in the context of technological adoption and abandonment.

The insights gained from this study offer valuable lessons for both the academic community and industry practitioners, particularly in the fields of microprocessor architecture, compiler design, and business strategy.

Author: Kamron Geijsen
Supervisor: prof. dr. ir. G.N. (Georgi) Gaydadjiev
Date: Monday 29th July, 2024
Faculty: Faculty of Electrical Engineering, Mathematics and Computer Science, Delft

TU Delft

Acknowledgements

I would like to express my gratitude towards the following people who have supported me throughout the project.

Georgi Gaydadjiev,
for being an above and beyond helpful advisor and supervisor to my project, and for linking me with extremely interesting key figures in the field.

Hans Mulder,
for his time and effort to share his experiences of developing Itanium at Intel

Mathijs Robbens,
for taking his time to read up on the topic, and sharing his expertise in business strategy frameworks

Jan Kist, Jasmijn Goudsmit, Miles Geijsen
for taking the time to proofread my thesis manuscript and providing constructive feedback

My parents,
for their love and support, and working hard to finance my Bachelor's and Master's degree studies

My girlfriend, friends, family
for graciously tolerating my extensive monologues about Itanium, whether they shared my passion for it or not.

Preface

The journey to understanding the rise and fall of technological innovations is often paved with curiosities, accidental discoveries, and profound questions about the 'path not taken'. My venture into the history and intricacies of the Itanium processor architecture started not in a classroom or a laboratory, but through a serendipitous brush with its story while exploring the broader landscape of Instruction Set Architectures (ISA). This Master's thesis, titled "Why did Itanium fail?", represents not just an academic inquiry but a personal exploration into a technology that, despite its promise, could not align its stars favorably in the fiercely competitive and rapidly evolving tech industry.

My initial encounter with Itanium was purely accidental, amidst my own endeavors to design a novel ISA. This discovery triggered a cascade of inquiries centered around the provocative question: "Why not this? Why has this not already been done?" Such questions are not unfamiliar in the realm of technology, where the graveyard of ambitious projects is vast and varied, including names like CELL, iAPX432, and Project Larrabee. My fascination was not just with Itanium but with the entire lineage and momentum behind the first x86-family processors—i186 through the Pentium. This lineage laid a cognitive map in my mind, contrasting the evolutionary paths of computing architectures.

As I delved deeper into Itanium's history, the words "promising," "ambitious," and "high performance" frequently surfaced, yet so did less favorable descriptors like "itanic," referring to its colossal missteps; "no platform," highlighting its lack of ecosystem support; and "complex compiler," pointing to the developmental challenges it faced. These dichotomies only served to deepen my fascination and steer my academic pursuit towards understanding why Itanium, despite its innovative potential, ended up discontinued.

My personal journey through the world of computing started early, growing up with x86 assembly language as both a playground and a puzzle. This lifelong journey has been one of demystifying the layers and intricacies of a dominant but flawed technology. My academic and practical engagements with ISAs—particularly RISC-V, through both discussion and implementation—have reinforced a crucial lesson: depth in one aspect of technology often belies the multifaceted and interconnected nature of success in this field. This understanding prompted me to not only investigate Itanium's performance but to also consider the broader array of factors including business strategy, compiler design, and development issues that influenced its fate.

This thesis also draws on my background in circuit design, ISA design, compiler theory, and language design, offering a multidisciplinary approach to the complex question of technological adoption and abandonment. The curiosity that sparked this research is part of a broader thematic interest that includes understanding why x86 continues to dominate despite its acknowledged flaws, why Itanium failed to replace it, and why architectures like CELL haven't found broader application.

In summary, this thesis does not merely seek to chart the reasons behind Itanium's failure but aims to offer a reflection on technological evolution, market dynamics, and the perennial battle between innovation and practicality. Through this work, I aspire to contribute to a deeper understanding not only of a single failed technology but of the broader mechanisms that govern technological success and failure in the intricate tapestry of the computing industry.

K.L. Geijsen
Delft, July 2024

Contents

List of Figures

List of Tables

Nomenclature

Abbreviations

Abbreviation	Definition
CISC	Complex Instruction set Computing
CPU	Central Processing Unit
EPIC	Explicitly Parallel Instruction set Computing
ILP	Instruction-level Parallelism
IPC	Instructions Per Cycle
ISA	Instruction Set Architecture
LLM	Large Language Model
RISC	Reduced Instruction set Computing
VLIW	Very Large Instruction Words
x86	80x86, referring to the IA-32 instruction set family
x64	x86-64, referring to the 64-bit extension to x86 (also known as AMD64, EM64T or Intel 64)

1

Introduction

To date, the largest disparity between a chip's investment and its commercial success can be found in the ambitious Itanium project, a collaboration between chip industry leader Intel and major hardware supplier Hewlett-Packard (HP). Despite the substantial financial investment and engineering efforts put into this project, the Itanium chip remains relatively unpopular. This stark contrast between the project's grand ambitions and its rapid descent into obscurity is particularly striking.

In the early 1990s, Intel feared its most popular CPU architecture, x86 – which currently still dominates the market – was not designed to overcome critical limitations on the horizon: memory limitations of 32-bit architectures and diminishing performance improvements from increasing CPU clock speeds. Anticipating these potential future challenges, Intel and HP joined forces to design a new CPU architecture to overcome these barriers. Despite Intel and HP's combined manpower, R&D, marketing, and substantial financial resources invested in Itanium, it failed to resonate within the market. This endeavor cost both companies and many customers billions in total investments, yielding at best merely marginal returns. This manuscript aims to clarify this discrepancy.

1.1. Motivation

Many have already asked themselves this question: "Why did Itanium fail?" However, there does not seem to be a fully accepted consensus on which key factor played the biggest role in this outcome. There are many competing theories in circulation, which will be covered in further depth in section 2.5. These theories are usually discussed in public forums, which lack a guarantee of technical credibility. On the other hand, Intel and HP, having the highest technical credibility, probably possess the closest thing to a satisfying answer to this question. However, they have not released any form of press release or report covering the key reasons why Itanium did not reach widespread adoption. This is possibly because it is not in their best interest to issue a report about why their architecture failed. Admitting their failure could harm their corporate image and could be seen as indirectly insulting to their engineers' abilities.

1.1.1. Broader perspective

Answering the main question, "Why did Itanium fail?" has broader implications than simply quenching one's curiosity. The discussion and analysis of this case study resonate beyond the lifespan of this project. The question lies at the heart of an even bigger, unanswered dilemma about the balance between innovation (research and investments) and commercial risks. For engineers or project leaders, it may provide insights into how to carve a path to success or how to steer clear of leading their projects to failure.

1.2. Research questions

As established by the title of this manuscript, "Why did Itanium fail?", the central question at the core of this study has already been formulated. However, it is beneficial to formally define the question for the

purposes of this quantitative study. Therefore, it has been rephrased as “What were the primary factors contributing to the commercial failure of the Itanium project?”. This question can be answered from various perspectives, including technical, historical, business strategic, and market analytical viewpoints.

To ensure a comprehensive analysis, the main question is further broken down into specific research questions:

R1. Can Itanium’s originally intended objectives be identified from existing literature?
R2. Which metrics can quantitatively and qualitatively assess Itanium’s success or failure?
R3. Which key technological or business strategic factor hindered Itanium’s market adoption?

1.2.1. Breakdown of sub-questions

As the project adopts an exploratory approach, hypotheses will be expanded dynamically. Potential issues of scope creep have been mitigated by initially framing hypotheses as research sub-questions. For example, one might ask, ”Did Itanium fail due to inadequate performance?” Each of these can then be systematically addressed. This strategy ensures that the study remains focused on addressing both the primary and secondary questions without excessive deviation into any singular area or study.

Since the last sub-question, R3: ”Which key technological or business strategic factor hindered Itanium’s market adoption?” is answered more broadly over many different aspects in this thesis, the question is first divided into several sub-hypotheses:

”Which key technological or business strategic factor hindered Itanium’s market adoption?”:

- Performance compared to competition
- Compiler complexity
- Inefficient program binaries
- Hardware complexity
- Negative market reception and reputation
- Incompatibility with existing software

These are rephrased into the following sub-questions, and each are covered in separate studies:

R3.1. How does EPIC compare with Superscalar in theoretical performance comparison?
R3.2. How does Itanium’s SPEC2000 performance compare with other architectures?
R3.3. How do IA-64’s program sizes compare with competitors?
R3.4. How significant is the proportion of NOPs in typical IA-64 binaries to performance?
R3.5. How much Instruction Level Parallelism is expressed in IA-64 compiled binaries?
R3.6. Do EPIC architectures reduce hardware complexity in CPU floor plans?
R3.7. How aware was the public of Itanium’s existence and purpose?
R3.8. What does Itanium’s popularity spikes imply about its public reception?
R3.9. How does the Differentiation Strategy framework shape Itanium’s market placement?

1.3. Research approach

At the start of the project, it was already clear that to answer the main question “Why did Itanium fail?”, it would involve an exploratory research project. Such a project entails updating the research direction based on the gathered data. This makes the project challenging to plan in detail in advance, or to define the finish condition. To mitigate this, this study has been broken up into six sections:

- **Features**: Study of Itanium’s features and differentiation from other ISA’s.
- **History**: Literature review and historical analysis to build the context around Itanium’s development, including motivation and purpose.
- **Performance issues**: An investigation into the performance-related problems frequently cited as key factors in Itanium’s failure. This includes comparative analysis by comparing emulation performance, benchmarks, and theoretical performance.

- **Hardware implementation**: An examination of the hardware design and implementation to identify potential issues that may have contributed to Itanium's downfall.
- **Compiler complexity**: An examination of the compiler complexity and a static analysis of the compiled code
- **Marketing missteps**: Market sentiment analysis and business strategy framework analysis

These sections are approximately ordered chronologically in which they would be researched. This ordering takes the duration of the project into account, it aims to balance different types of studies (Sociological vs Technical) to balance the cognitive load. Task switching and task diversity can help mitigate cognitive overload caused by sustained attention over long periods of time [7]. Given that this is a three-quarter project, any opportunity to help reduce the chance for cognitive overload and burnout could improve the longevity of the project.

1.3.1. Research planning

This explorative approach with expanding hypotheses on the fly does give a good amount of flexibility and space for exploration, however it also can lead to issues in planning and scope creep. In this project, the flexibility and space for exploration are essential. Given that the study may not have a clear-cut path beforehand, the exact planning must be explored and refined along the way. To mitigate this, approximately two months has been planned as a buffer at the end of the project, to absorb any challenges in planning and resource management. While this is not a hard deadline, it is a good indicator for what is "on schedule". The planning outline is depicted in the chart in Figure 1.1.

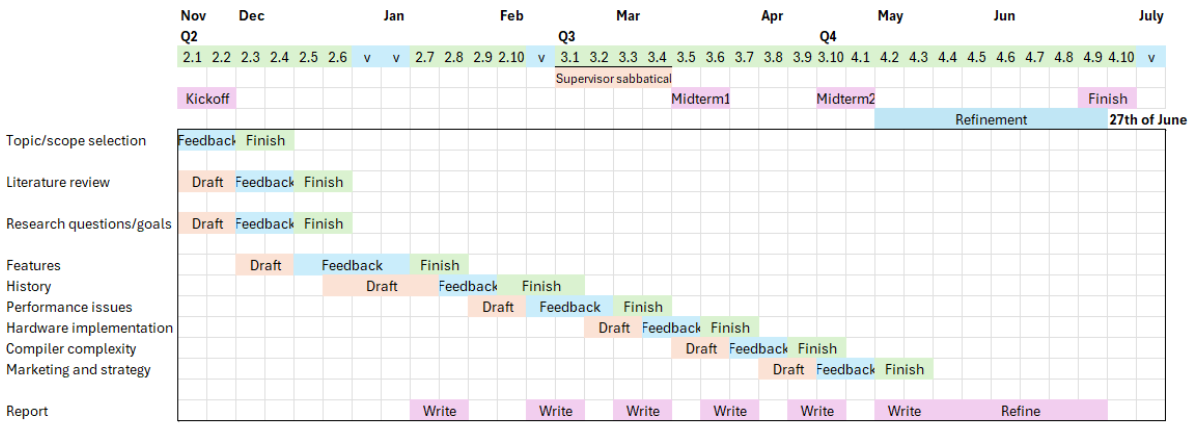


Figure 1.1: Initial research planning

1.4. How to Read this Thesis

This thesis is structured into multiple categories of studies, each organized into distinct chapters. Each chapter discusses new study results except for the background and conclusion chapters.

The subsequent chapter, chapter 2: Background, lays the groundwork with references and relevant information essential for understanding the remainder of the thesis. It provides a detailed overview of the Itanium processor architecture, its developmental context, and its intended purpose. Additionally, it expands on the context necessary for addressing the main research questions, including an elaboration on current literature and hypotheses.

This leads to the first research study in chapter 3: Automated Public Sentiment Analysis, which describes the methodology and results of studying public perception of Itanium through automated sentiment analysis.

The following chapter, chapter 4: Performance Comparative Analysis, compares Itanium's performance with other processor architectures using both real-world benchmarks and theoretical analysis.

Subsequently, chapter 5: Compiled Binary Static Analysis of IA-64, delves into the structural efficiency of Itanium's instruction set architecture (ISA). This chapter discusses methodologies for disassembling and analyzing compiled binaries, with a focus on the unique challenges posed by the IA-64 architecture, such as bundle NOP overhead and Explicitly Parallel Instruction Computing (EPIC) binary sizes.

This is followed by chapter 6: Comparative Analysis of Hardware Complexity, which explores the hardware complexities that potentially extended the development phase of the Itanium processor.

In the penultimate chapter, chapter 7: Differentiation Strategy Framework Analysis, a business strategy approach is employed to address the question, "Why did Itanium fail?". This analysis is informed by the conclusions drawn from preceding chapters.

The thesis concludes with chapter 8: Conclusion. This final chapter summarizes the findings from each study, providing a cohesive response to the research questions posed in the Introduction. It synthesizes the answers to the sub-questions, culminating in a definitive answer to the question, "Why did Itanium fail?". Additionally, this chapter outlines potential avenues for further research, building upon the groundwork laid by this thesis.

Each chapter in this thesis is interconnected, progressively building upon the information and insights gained from the preceding sections. However, each chapter has been structured to be self-contained, minimizing the amount of prior knowledge required from other chapters, thus allowing readers the flexibility to start at any point in the thesis as they desire.

2

Background

This chapter provides a comprehensive overview of Itanium's development timeline, features and market goals. Starting with contextualizing the Itanium project, by providing a narrative of the development, release and decline in Section 2.1.

2.1. Timeline of Itanium's development

This section provides an overview of critical events and technological milestones related to Itanium's lifetime, ranging from the research and developments leading to Itanium, up to its decline in popularity.

2.1.1. The prelude to Itanium

x86 has been a great success story for Intel. Releasing new groundbreaking innovations with each generation 8086 up to this date. Many developments in the time graph in Figure 2.1 relate to the x86-family. With the great commercial success of Intel's 4004, 8008, 8080, Intel started developing the radically different 8800 architecture (also known as iAPX-432). Designed to be a high-level CPU, with features like function argument passing by value, a built-in memory management system, and custom support for the Ada language [19]. This CPU would later be known as Intel's first major commercial set-backs [4]. The design team experienced some substantial delays throughout the development of the processor, so Intel assembled a new team to work on a smaller project to promptly deliver 8086 in 1978. While it was not the intentions at the time, this CPU would become a staple in the processor landscape to this date.

The development of 8086 quickly followed by 80186 and 80286, both introducing incremental improvements of 8086. The next substantial update to the x86-family was the 80386, the first 32-bit architecture in the x86 line. A couple years later Intel developed another radically different CPU named 80860; a RISC processor with Very Long Instruction Word (VLIW) features [10]. Still hesitant due to the public outrage that was iAPX-432, but enough to get a slight foothold in some industries (such as air crafts). In the meantime, HP has been developing their own Instruction set, PA-RISC. With a focus on high-performance, it gained notable traction.

2.1.2. Itanium's development

With HP questioning the viability of RISC over the long run, they started researching other methods of performance improvements. They converged on to Explicitly Parallel Instruction Computing (EPIC). After proving its viability, they stepped towards Intel for a collaboration. Thus started project Tahoe - the provisional name given to the project. They would continue to develop this architecture, while Intel developed their x86 lines. Around the same time of Itanium's conception, Intel's now famous Pentium processor was released. It was a substantial improvement in performance over the previous iterations, introducing superscalar execution. There was slight uncertainty of the direction of x86, as there were hypotheses that it would not be able to scale for better performance compared to EPIC. Nonetheless, Intel still kept developing x86. Releasing Pentium Pro the year after, being the first out-

of-order superscalar x86 processor. These architectures only reaching about 2 to 3 instructions per cycle max, they hoped to get more performance out of EPIC, to continue the exponential performance improvements curve. Itanium faced some significant delays, starting at 1999, and ending up with a two year delay after the originally planned date, in 2001. The reception was met with immediate skepticism.

2.1.3. Itanium's active years

The Itanium processor underwent several revisions after its initial release. Soon after the debut of the original Merced processor, the subsequent iteration, codenamed McKinley, was introduced. This new processor promised significant performance enhancements compared to its predecessor. Despite these advancements, performance benchmarks generally indicated that Itanium systems continued to exhibit mediocre performance. Coupled with its high cost and incompatibility with existing Intel tools and systems, Itanium experienced challenging early years. Nevertheless, it managed to achieve a slow but steady increase in popularity.

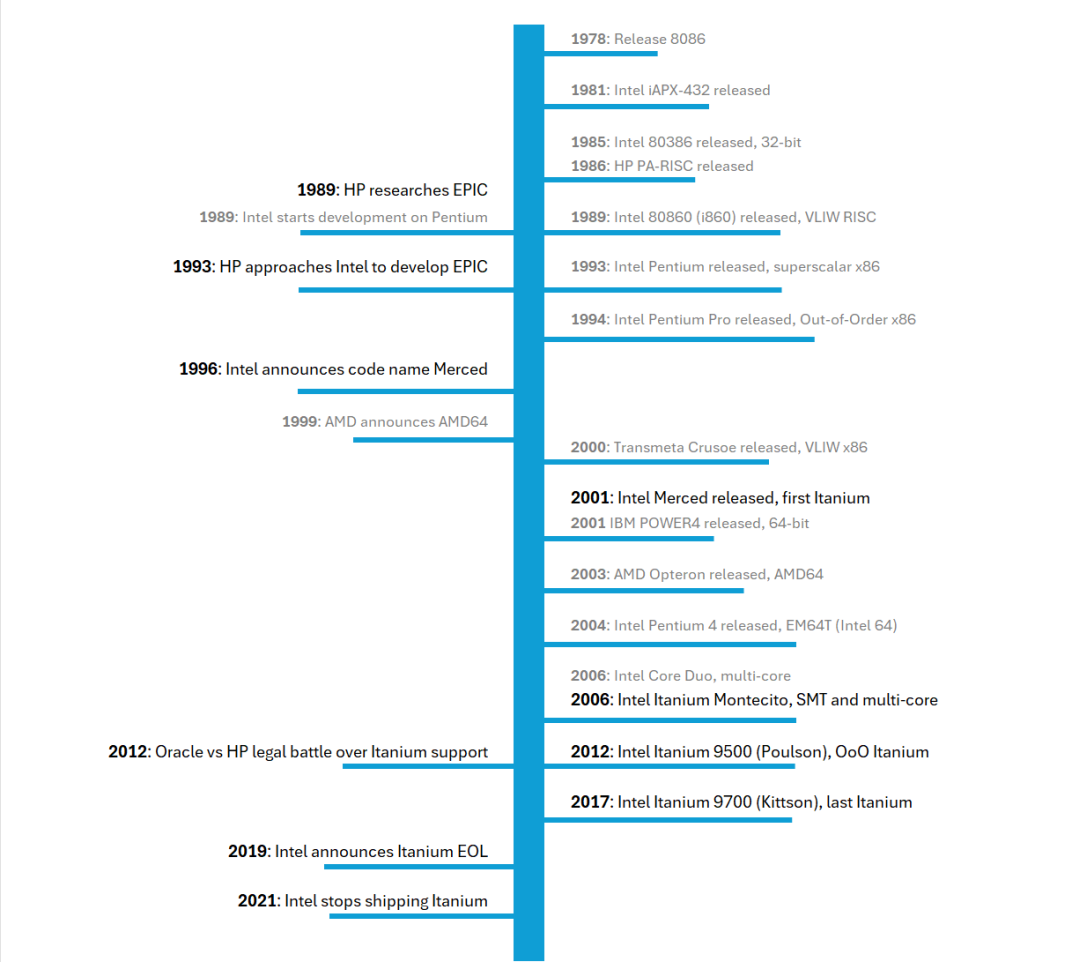


Figure 2.1: Timeline of events and releases related to Itanium

2.1.4. Itanium's decline in popularity

In 2003, just two years after Itanium's release, Intel was met with a great predicament. AMD, Intel's biggest competitor, had just released a 64-bit version of x86: AMD64. Intel had to decide quickly what to do with their own x86-line processors. Either let AMD have the 64-bit market of x86, which meant dooming Intel's x86-32 in favor of Itanium, or also pick up the competition for 64-bit x86, and ensure their spot at the top there. Intel put Itanium on back-burner, in a public announcement that they would also develop 64-bit x86 to compete for the same ground Itanium was trying to get a hold of [26]. In 2004, Intel released their own version of AMD64, originally with the name EM64T, for Extended Memory 64-bit

Technology. Companies were starting to see what was going on, and one after the other, hardware and software vendors alike, such as Windows, Dell, LLVM started to drop support of the now niche IA-64. In 2019, Intel declared that the Itanium product line would reach end-of-life (EOL) in 2021, after which they would discontinue shipping new Itanium chips.

2.2. Motivation behind Itanium

In 1993 Project Tahoe, the code name for Itanium before the name “Itanium” existed, is conceived, a collaboration between Intel and HP to design a “new 64-bit architecture”. By then, limitations of 32-bit memory addresses, frequency barrier and complexity of hardware (superscalar) were becoming prevalent in x86. So far, Intel has had a great track record at addressing barriers to improve the performance of x86 systems: Starting from the original 8086, there were first incremental improvements throughout the 80186 and 80286 in more efficient pipelining, additional instructions and frequency scaling. Then, the 80386 introduced 32-bit operations and addressing modes, allowing for larger memory limits. 80486 introduced cache and incorporated floating point, and 80586 — more commonly referred to as Pentium — was the first superscalar x86 processor. But x86 had some limitations in terms of its future proofing: Intel realized that the market for computations was moving more towards data centers and high-performance compute, rather than to the end customers. Analysts at the time argued that the need to upgrade a personal computer becomes less necessary. Any improvements would result in unnoticeable runtime improvements from 10 milliseconds to 2 milliseconds, which to the human eye is mostly indistinguishable. While at the scale of data centers, such an improvement is quite impactful, as such an improvement could mean needing to buy 80% fewer chips and lower operational costs. Itanium was strategically aimed at high-performance computing environments and was intended to replace the x86 (IA-32) architecture. The transition from IA-32 to IA-64 was not merely an upgrade but a significant leap aimed at setting a new standard for the next generation of computing.

2.3. Itanium processor architecture overview

The IA-64 instruction set is often described as “ambitious” due to its deviation from traditional Reduced Instruction Set Computing (RISC) and Complex Instruction Set Computing (CISC) architectures. to the newer VLIW/EPIC paradigm. Key features of IA-64 that support this architecture include: static parallel instruction scheduling, speculative execution and branch predication. Each of these features (and more) will be discussed in a comprehensive summary in this section.

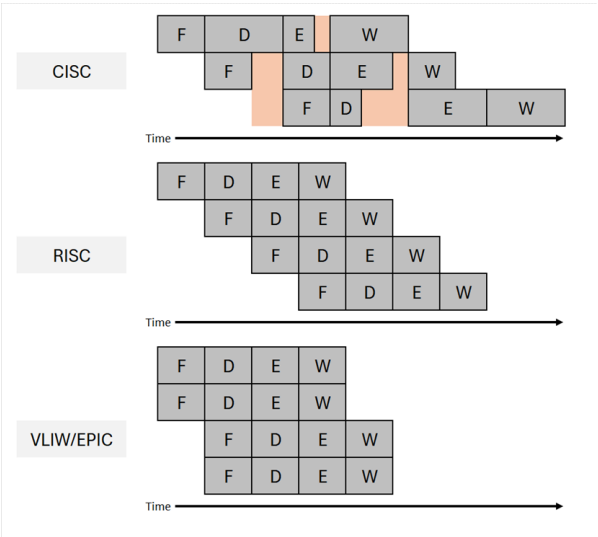


Figure 2.2: ILP comparison between CISC, RISC and VLIW/EPIC

2.3.1. Explicitly Parallel Instruction set Computing ISA

The core philosophy behind EPIC is static parallelism. Those familiar with VLIW may correctly assume that this works similarly. EPIC and VLIW share the same principles such as aiming to perform multiple

instructions per cycle in parallel, by explicitly encoding the parallelism of instructions in the machine code. How this is achieved is where they differ. VLIW encodes one large instruction which is executed over multiple units in parallel. Whereas EPIC encodes this parallelism in the form of bundles of smaller instructions. These instructions are then assigned a functional unit, through a technique named templating (bundle metadata). Templates can also assign the paralellism barriers between two sequential instruction groups. Figure 2.3 depicts the representation of a bundle in bits, Figure 2.4 show how multiple bundles can make up an instruction group. Some templates also have barriers inside the bundles, in the Figure 2.4 it shows such a template M;MI and M;MI; in templates in bundles 1 and 3, respectively.

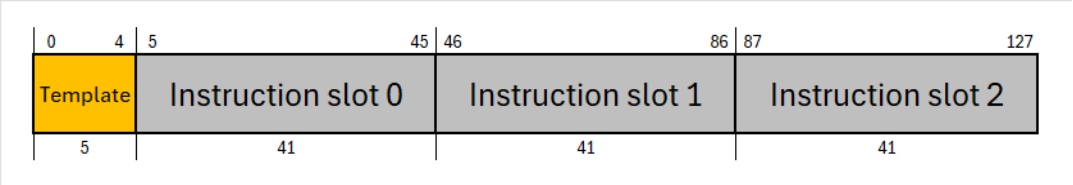


Figure 2.3: Representation of a bundle in bits

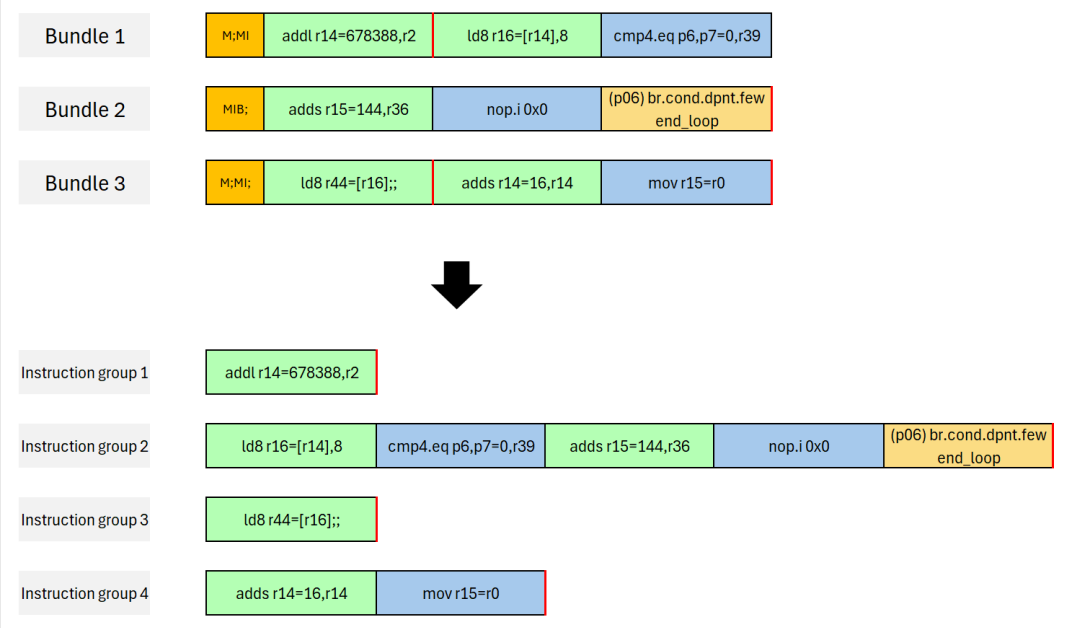


Figure 2.4: Representation how bundles relate to Instruction groups

In Itanium assembly, the instructions in the figure would looks like this:

```
1 [MMI]       addl r14,678388,r2;;            # executes in cycle 1
2             ld8 r16=[r14],8                 # executes in cycle 2
3             cmp4.eq p6,p7=0,r39             # executes in cycle 2
4 [MIB]       adds r15=144,r36                # executes in cycle 2
5             nop.i 0x0                       # executes in cycle 2
6       (p06) br.cond.dpnt.few end_loop;;     # executes in cycle 2
7 [MMI]       ld8 r44=[r16];;                 # executes in cycle 3
8             adds r14=16,r14                 # executes in cycle 4
9             mov r15=r0;;                    # executes in cycle 4
```

Static Instruction Level Parallelism scheduling

In static Instruction Level Parallelism (ILP), as depicted in the example assembly sample in figure 2.4 you can see that first one `add` instruction is executed, with no paralellism. Then in the next cycle, 5 instructions are executed in parallel: The `ld8` (load 64-bit or 8 bytes), `cmp4.eq` (compare 32-bit for

equality), `adds` (add with immediate), `nop.i` (No Operation) and `br.cond.dpnt.few` (Branch conditionally dynamically not taken hint expecting few instructions at destination). However the `nop` instruction does nothing, so this is omitted, leaving 4 effective instructions being executed this cycle. The next cycle is the `ld8`, followed by the `adds` and `mov` in the next cycle. In this basic demonstration, the instructions would finish executing after 4 cycles.

On a traditional systems, these instructions would all execute sequentially. An example of its machine code is depicted below:

```
1 lea rdx,[rsi+678388]          # executes in cycle 1
2 mov rdi,qword ptr [rax]       # executes in cycle 2
3 add rax,8                     # executes in cycle 3
4 lea rdi,[rbx+144]             # executes in cycle 4
5 cmp rcx,0                     # executes in cycle 5
6 je end_loop                   # executes in cycle 6
7 mov r8,qword ptr [rdi]        # executes in cycle 7
8 add rax,16                    # executes in cycle 8
9 mov rcx,0                     # executes in cycle 9
```

Assuming each instruction has a single cycle delay, it would lead to approximately nine cycles, instead of five in the parallel EPIC example.

Fixed-sized bundled instructions

Due to the fixed sizes of the bundles and limited amount of templates, when there is no way to schedule a legal template or bundle that perfectly fits `nop`s are inserted to pad the bundles.

Instruction unit templating

Each bundle contains a 5-bit template field. This limited amount templates and fixed size bundles trivializes decoding. These templates allocate execution unit classes to the instructions, to hint to the hardware what kind of occupations to expect early on in the pipeline.

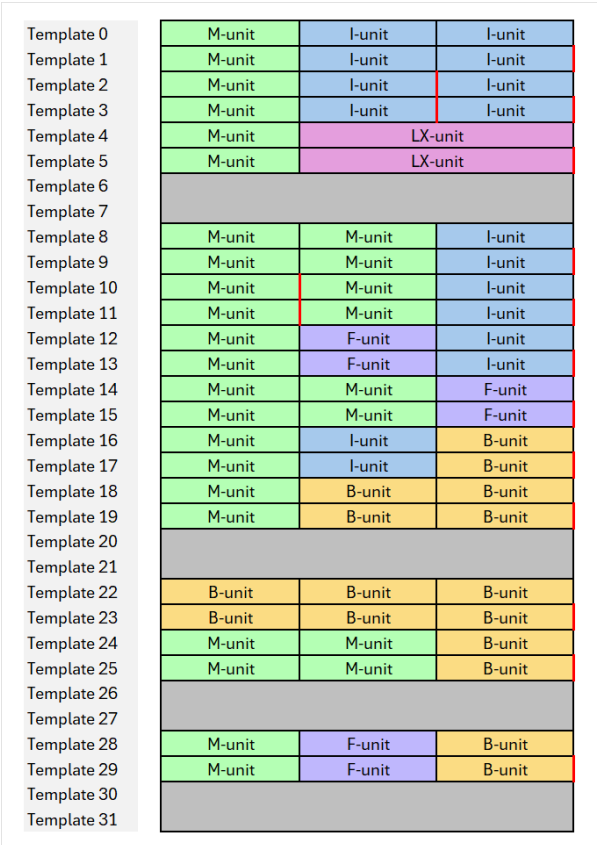


Figure 2.5: Itanium's Template Set

2.3.2. Large register files

Itanium also features a large register file, with 128 Integer and 128 Floating point registers. This allows for more independent data paths through different registers with no need for regard of false dependencies. The large amount of registers also lightens the register pressure, leading to fewer register spills onto the stack, and generally requires fewer memory accesses, as the values can be stored and saved over longer periods in the register files.

Reducing the amount of memory accesses also improves performance for the specific workload Itanium is designed for: Data center computing. With memory bandwidth being common bottlenecks on these systems, Itanium can inherently reduce this issue.

2.3.3. Rotating registers

Itanium incorporates a unique feature known as register rotation, a similar feature to register windowing popularized in the Sun SPARC instruction set architecture [14]. Rotating registers allows for better throughput through the use of software pipelining in loops. The larger register file mentioned earlier synergizes with this technique by providing more registers to hold intermediate values during loop iterations.

2.3.4. Speculative execution

Speculative execution is a fundamental technique for reducing latency in programs at the cost of potentially redundant computational work. Speculation can manifest in many different shapes and forms. In Itanium, there are three main types: branch prediction, predicated instructions, and memory speculation. The first type, branch prediction, is a ubiquitous optimization across CPU architectures. At a relatively low cost in terms of area and complexity, a branch predictor can significantly improve a CPU's performance under most circumstances. The other two types of speculative execution — predicated instructions and memory speculation — are more unique to Itanium.

Predicated instructions

Itanium features predicate registers, which are special registers that store the results of boolean expressions. Each predicate register can hold a value of true or false. Predicated instructions will fully execute, but only write back their result or fully complete a conditional jump if the instruction's predicate is resulted to true. If false, the instruction still goes through most of the pipeline stages (such as fetch, decode, execute). even multi-cycle instructions will commence executing. But when the predicate is known to be false, this instruction may be flushed from the pipeline, freeing up the functional units.

Memory speculation

Memory speculation is a more advanced feature, yet ubiquitous in modern CPUs. While modern CPU's use a plethora of different memory speculation techniques, thanks to their out-of-order execution designs, Itanium only features load speculation. A speculative load is commonly used to balance performance and correctness [29]. A load that occurs after a store where the store address is uncertain, *can* lead to a Read-after-Write (RAW) violation if not handled properly. What CPU's commonly do is to load the value in memory as soon as possible, then if the store has already occurred, then there is no issue. However if the store has not yet occurred, it will load the value speculatively, and perhaps even speculatively execute using this uncertain value. Only after the store is known, it will mark this speculative load as invalid if there is overlap in the addresses (i.e., the Store did in fact store in the address of the load) or there was no overlap (i.e., the Store was somewhere else in memory, and has causes no dependency issue with this load), then the speculative execution will simply resume. The load marked invalid will need to recover from the memory ordering violation, which is often up to the programmer/compiler to decide how to recover correctly. This usually entails redoing the same load, but this time the load order is correct as the store has fully completed.

2.3.5. Security and robustness

Itanium reatures a wide range of security, robustness and reliability features, such as Error Correcting Code (ECC) memory support, Machine Check Architecture (MCA) and No-eXecute (NX, also known as Trusted eXecution Technology or TXT) for security.

2.4. Historical Comparative Analysis of EPIC and Competing Architectures

The shift towards introducing Instruction level Paralellism (ILP) in CPUs became more prevalent from the 1980s to the early 2000s, with many competing architectures — many of them classified as Very Long Instruction Word architectures — employing vastly different approaches besides Itanium's EPIC. Some of the most notable approaches include:

- Intel i860 — VLIW;
- Cydrome Cydra — VLIW;
- Multiflow TRACE — VLIW;
- Intel Pentium — superscalar;
- IBM SCISM (Scalable Compound Instruction Set Machine);
- Transmeta — VLIW + dynamic binary translation.

This list is not exhaustive, but it provides enough examples to illustrate the context of EPIC compared to other approaches.

Traditionally, one of the first popular approaches to ILP was found in VLIW. In VLIW, a single ”instruction” may execute multiple smaller sub-instructions, referred to as operations. For example, an instruction could perform a LOAD (load from memory) operation while also performing an ADD (integer addition) operation in parallel. While intuitively this could be seen as one instruction, engineers viewed this from another perspective as two separate instructions and started to treat them as such. Since sequential instructions have certain sequential properties such as data hazards and pipelining, whereas a single large instruction that performs multiple operations in parallel can lead to lower execution times. This approach is called VLIW because its instructions are considered single but very large. In Intel's i860, this concept is particularly apparent. Although the i860 is generally considered more of a RISC model than VLIW, it supports several instructions that perform two operations in parallel, making it the first example of Intel's commercially shipped VLIW CPUs.

Some other CPUs architectures leaned more into the fact that these sub-instructions could be seen as separate instructions, even allowing way more than two different instructions to run in parallel. Cydrome's Cydra and Multiflow's TRACE [15] are classic examples of wide VLIW architectures, lead by Bantwal (Bob) Rau and Joseph (Josh) Fisher, respectfully. These CPUs used a similar approach to VLIW, of having a fixed-width instruction size, and having different units encoded in the instruction. For example, one instruction could perform a MEM (memory access), INT (integer operation), INT and FP (floating point operation) operations in parallel. For example, when an instruction would perform a LOAD, ADD and FMUL (floating point multiply) in parallel, the LOAD would occupy the MEM slot, the ADD would occupy the INT and the FMUL would occupy the last field. This would leave one INT field unused, which would be filled with a NOP (no operation). Since it was a challenge to find multiple instructions which could execute in parallel at a certain cycle, many of the slots were filled with NOPs. This caused the program binaries to be very large.

IBM's novel Scalable Compound Instruction set Machine (SCISM) [31] approach mitigated the amount of unproductive NOPs in the instruction encodings. While SCISM's key features go well beyond the scope of this thesis, its novel approach to allow instructions to have explicit hazard boundaries reflects a new approach to VLIW: Dynamic VLIW. Instead of one large VLIW instruction having fixed slots, one VLIW instruction could be seen as having a variable amount of slots based on a dependency boundary tag. In SCISM, the example illustrated in the previous paragraph will be encoded as traditional RISC or CISC instructions with tags. With a LOAD, ADD and FMUL instructions executing in parallel, SCISM would only mark the boundary tag in the last instruction (FMUL), indicating that these three will be executed in parallel. In hardware, these individual instructions would be compounded into a single VLIW instruction.

This last approach shares many similarities with EPIC, by making the instruction boundaries explicit and letting the functional unit or VLIW slot allocation (called compounding in SCISM) to be done in hardware, but the main difference lays in the fact that Itanium is more strict in a sense, also requiring each

instruction to be assigned a unit type (e.g., an INT unit as in the example) which means while it does not specify which slot it will use exactly, it simplifies the hardware allocation complexity. Additionally, SCISM performs this instruction compounding through instruction dependency checking during CPU stalls (such as cache misses or slow operations). This dependency checking is expensive in terms of hardware and latency, hence why it is only active during stalls.

Traditional systems such as RISC and CISC were developing similar features as instruction compounding in an approach now commonly named superscalarity [1]. However such superscalar systems do not require any boundary tags in software, and perform all dependency checking in hardware. This approach is significantly more complex, as mentioned in the previous paragraph. However, the great expense in hardware makes up for the great improvement in performance in the form of out-of-order (OoO) execution, which allows instructions to not only execute in parallel, it can execute instructions at any time when it has been shown to have no dependencies on other unfinished instructions. To bring up the example from before again, the LOAD, ADD and FMUL instructions could execute in the MEM, INT and FP units in parallel due to the hardware figuring out they do not share any dependencies, but the last unused INT instruction slot can now be filled with an instruction which would normally be executed later, but due to it having no dependencies it could execute the same cycle. This approach does not rely on the compiler for any instruction ordering or dependency checking, performing everything in hardware. This superscalar OoO approach was popularized in Intel's Pentium CPU.

Another radically different approach to extracting ILP was in Transmeta's Crusoe [13], which ran traditional CISC IA-32 (also known as x86) code, on its natively VLIW CPU using a translation layer. This technology was dubbed CMS (Code Morphing Software) which dynamically translated IA-32 instruction into traditional VLIW instructions under the hood. This approach was a middle ground between superscalar and SCISM, as the compounding mechanism was similar to SCISM but did not require any dependency boundaries from the compiler.

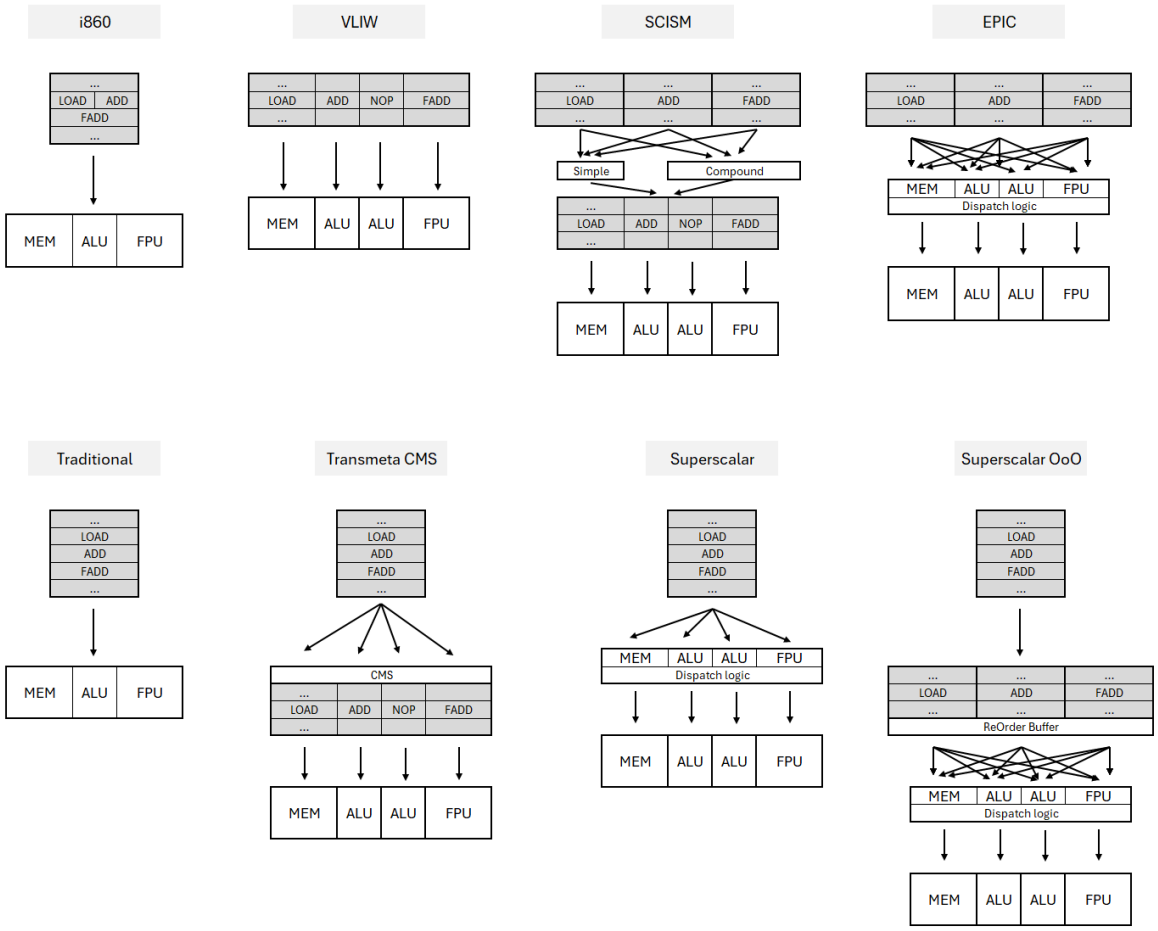


Figure 2.6: Comparison of different multi-issue architectures, including traditional single-issue, Intel i860, VLIW, SCISM, EPIC and superscalar

An overview of how EPIC differs compared to these other approaches is depicted in figure 2.6, EPIC would be most similar to SCISM and in-order superscalar, leaning closer to SCISM. While SCISM gives the hardware hints of dependency boundaries, it still needs to allocate the instruction to the units in hardware. This is marginally improved on Itanium, which explicitly specifies the hardware unit type for each instruction. A more substantial difference is in the execution backend. While SCISM has a VLIW backend, which explicitly specifies the timings of instruction executions, Itanium has a dynamic score-boarding system which keeps track of instruction dependencies to stall individual operations instead of entire VLIW instructions, similarly to superscalar.

2.5. Current hypotheses implied from common criticisms

There are currently a wide range of answers to the question. Some about technical aspects, such as lacking CPU performance, and others from a business perspective, vendor lock-in between Intel – HP. Just according to the first Google search, there is already a wide range of potential issues:

Poorer than advertised performance

A popular claim was that Intel's promises for Itanium's performance were significantly higher than its performance in practice. In reality, the performance was similar at best, but most often worse than contemporary processors.

Developmental issues

Itanium was delayed multiple times. It was originally set out to be released in 1999, but in July 1997 this release date was postponed to the Q3 of 1999. In May 1998 it was delayed to Q2 of 2000. In July 2000 they pushed the delay into Q2 of 2001

Too expensive

The initial pricing of the original chips was set at $4,227, with the cheapest models priced at $1,177. These prices were among the highest for processors at the time, which is hypothesized to have contributed to initial customer hesitance in adopting the technology. This is especially notable when compared to contemporaneous models such as the Pentium III Xeon, which were priced between $425 and $3,692, offered better performance, despite being released earlier.

Table 2.1: Itanium Merced prices at release (May 2001)

Merced	Price (USD)
Itanium 800MHz 4MB L3	$4,227
Itanium 800MHz 2MB L3	$1,980
Itanium 733MHz 4MB L3	$4,227
Itanium 733MHz 2MB L3	$1,177

Incompatible with x86

While the first Itanium chips had a compatibility engine to run legacy 32-bit x86, it performed extremely poorly. It had a comparable execution speed to the original Pentium from 1991. Despite Pentium's 10x lower clock speed. This compatibility engine seemed to clearly be meant for just that: Compatibility — not for competitive performance. However, with ecosystems still running legacy x86, it would make their existing code run 5-20x slower.

Too difficult to compile

Many attributed the slow execution times of Itanium to its compiler. With a small software/compiler ecosystem for Itanium,

The advent of AMD64 (x64)

The downfall of Itanium started as soon as AMD64 was released with AMD's 64-bit Opteron CPU in 2004. Interest in Itanium plummeted and Intel shifted their strategy more towards x86.

2.5.1. Hypothesis credibility

As mentioned earlier in the Motivation section 1.1, the credibility of these hypotheses are quite varying. While some claims are based on real data, such as performance and pricing, there still seem to be some caveats with simply taking these claims at face value. These criticisms may be useful in laying the groundwork of the full picture.

2.6. Quantitive study of Itanium's market success

Before answering the question "Why did Itanium fail?", we must first define what the line is between success and failure.

2.6.1. Itanium's goals

Itanium's goals can (partially) be derived from not only direct sources, but also indirect sources:

- The renaming of i386 to IA-32 was a strategic step to set the stage for IA-64, indicating a clear commitment to advancing beyond the existing architecture.
- As Intel was not upgrading IA-32 to 64-bit, it was evident that IA-64 was not just a side project but the future of Intel's architecture.
- Sales presentations and strategic communications from Intel indicated that IA-64 was expected to replace IA-32 entirely.

- The development of IA-64 was crucial not just for Intel but also for maintaining and enhancing partnerships, notably with HP.
- To out-compete its rivals, IA-64 needed to surpass the performance and efficiency of contemporary architectures.

To objectively assess whether Itanium achieved its intended goals, it is necessary to define specific metrics that reflect its performance and acceptance in the market. These metrics encompass technical specifications, market impact, and broader industry implications.

The quantitatively measurable metrics are defined as follows:

- Performance: Measured in execution time (s).
- Energy Efficiency: Measured in power draw throughout the execution ($W * s$ or J).
- Size: Measured in either die area (mm^2) or transistor count.
- Cost: Measured in total cost of buying and maintaining the processor ($\$/yr$).
- Scalability: Measured in performance increase over adding more processors.

Next the qualitative metrics are defined as follows:

- Niche Purposes: How well it performs for a specific workload.
- Versatility: How well it performs across many different workloads.
- Academic Adoption: How suited it is for teaching and research.
- Security: Built-in security against vulnerabilities.
- Accessibility: Required licenses to use or implement CPU/ISA.
- Tools and Platform Support: Maturity of the existing toolings.
- Future-Proofness: Its adaptability/extendability to changes throughout time.
- Public Sentiment: The public perception and industry sentiment towards the architecture.

By carefully analyzing these metrics, a detailed picture of Itanium’s performance against its initial goals and the market expectations can be derived.

3

Automated Public Sentiment Analysis

The first step in understanding why Itanium did not achieve widespread success is to analyze the market's perception of the processor. This chapter aims to answer the following research questions through an automated sentiment analysis:

R3.7. How aware was the public of Itanium's existence and purpose?
R3.8. What does Itanium's popularity spikes imply about its public reception?

Gaining insight into this reception can reveal common points of criticism and acceptance, highlighting both strengths and shortcomings. As detailed in the Background section, Itanium was initially intended to replace the x86 architecture across diverse applications, targeting data centers, client computing, embedded systems, and other specialized areas such as for governments and research [18]. This means that the target market was quite broad. Given the broad scope of potential customers, understanding the general public sentiment towards Itanium is crucial.

This chapter describes the sentiment analysis study, describing the data collection and filtering process in section 3.1, followed by an explanation of the automated sentiment analysis methodology using large language models (LLM) in section 3.2, and finally representing and discussing the results in section 3.3.

3.1. Methodology

The market analysis consists of three parts: Firstly, collecting articles and papers related to Itanium, from e.g. scientific databases and news article websites. Secondly, figuring out the public's reception of Itanium by summarizing key points of criticism and acclamation.

The following study is an automated public reception analysis. Rather than the traditional approach of a manual literature study, this analysis processes a large amount of papers - around 3,000 - in a realistic time-frame. While the results could arguably be better quality when reviewed manually, the general trends can still be concluded from the data. Additionally, an automated study can stretch over a way larger amount of papers, and is more consistent in research reproducibility. The automated sentiment analysis approach is able to give an indication of the possible strengths and weaknesses of Itanium.

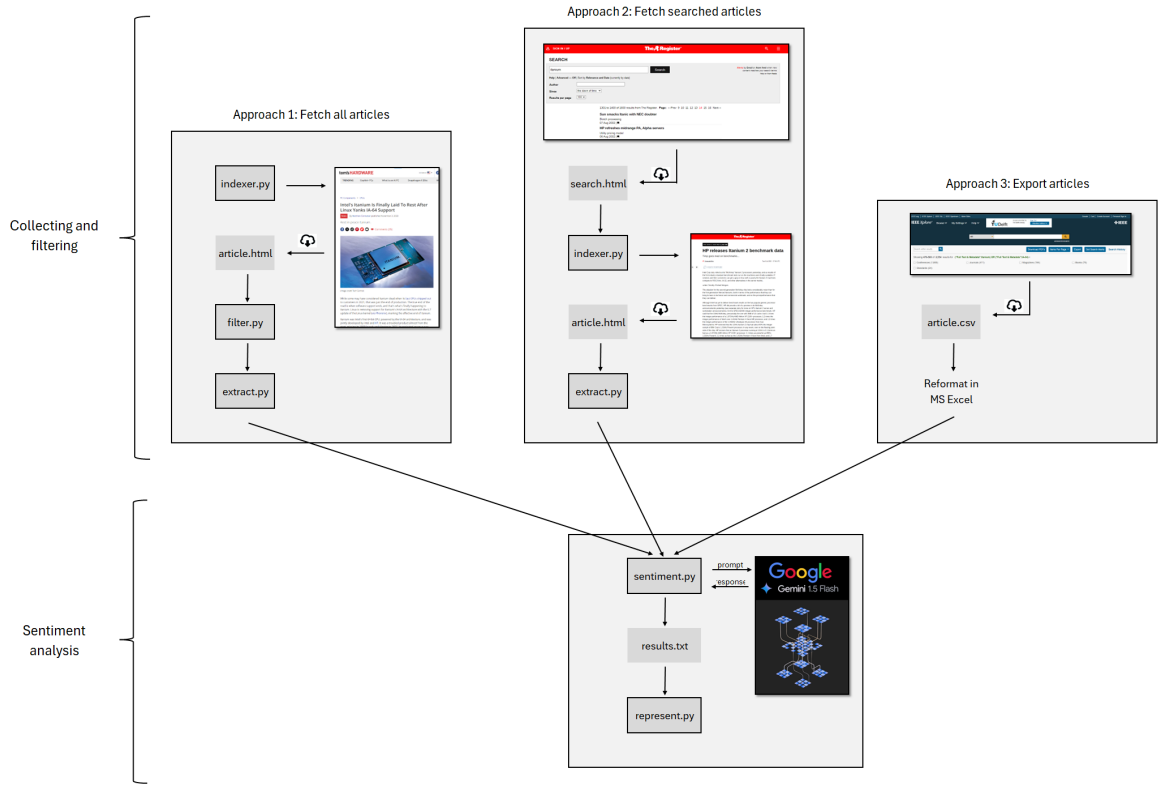


Figure 3.1: Sentiment analysis workflow overview

3.1.1. Article and paper collection and filtering

To determine which online papers and articles to gather, the data must be considered to effectively address the research question according to the following three key aspects: the relevance of the paper to Itanium's commercial success, the article's underlying agenda (including potential biases and intentions), and the expected credibility of the platform's authors.

Online platform filters

Some of the first online platform candidates that come to mind are news articles, forums, corporate press releases and academic research about the topic. Each of these platforms have certain bias and relevance guarantees.

Table 3.1 indicates that technology news websites generally have the highest quality data for this research due to several factors: The articles give professional commentary about technology, while covering a wide range of possible technological and commercial topics around Itanium. Academic journals come in second-place, since they have high credibility and often cover a wide range of technological features and use cases of Itanium in an objective manner, but will shy away from direct criticisms and acclamations of the CPU, as these articles are usually written with an objective stance. Financial news websites also may have some technological credibility, but since the focus is more on the business strategic aspects of Itanium, it generally does not cover in the technological shortcomings or benefits of the CPU in-depth. While forums can have high-quality data in certain discussion boards or by specific experts, the credibility variability is challenging to make assumptions about and filter on, which can not assert a clean dataset. While technology news websites do not always have the highest credibility, these sites least assert a certain level of credibility and structure, making it easier to analyze. Corporate press releases (such as Intel Newsroom) have excellent first-hand data about the benefits of the CPU, but will generally not cover the product's shortcomings or general audience's criticisms, making it heavily biased.

In conclusion, this automated market sentiment study will contain only technological news articles and

Table 3.1: Pros and Cons of Various Online Platform Types for Sentiment Analysis of Itanium

Online Platform Type	Credibility	Agenda	Representative of Users
Technology News	Medium Reputation based credibility	Possible influence Paid sponsorships	Moderate Tech enthusiasts
Academic Publications	High Peer reviewed	Objective	Low Regardless of commercial success
Financial News	Low No technical focus	Possible influence Partnerships	Medium
Forums	Low	Varied	High Broad spectrum
Press Releases	High	Highly biased	Low

research papers for their best balance in commercial coverage, credibility and data bias compared to the other sources.

News and academic websites

To make a listing of the top technology news websites which would cover information about Itanium is a subjective challenge. However simply prompting Google News “Intel Itanium” returned results of seven large news websites:

1. AnandTech;
2. Ars Technica;
3. CNET;
4. HPC Wire;
5. Techpowerup;
6. The Register;
7. Tom’s Hardware.

These websites are each popular websites, covering a wide range of international tech while having a large user base of more than 100,000 users. While this is not a comprehensive list, adding more would arguably lead to diminishing returns in answer quality and extend the study duration. Additionally, the results are mainly used as indicators for the further studies rather than as academic experiment results. Specifically these websites were selected in this study based on: Google News search Index, the websites’ Terms of Services (for legal compliance for automated web scraping) and manual credibility inspection based on a sample study. The credibility study also extends to the next steps of the automated senitment analysis, where the credibility is monitored through outliers.

Next, to make the listing of the top digital research libraries is generally more straight forward, as there are tools and websites that help with finding the correct scientific database given any topic. However, for this specific case study, it still does pose a significant challenge due to two main issues: Paywalls and automated scraping prevention. Some websites block automated access entirely, requiring a paid subscription to access their content. Other research repositories, such as IEEE Xplore, Research Gate, Google Scholar, do not allow scraping or automated indexing of any kind.

Below are the eight tested digital research repositories:

1. IEEE Xplore;
2. Google Scholar;
3. ResearchGate;
4. ACM Digital Library;

5. ArXiv;
6. ProQuest;
7. Scopus;
8. Web of Science.

IEEE Xplore and Scopus allowed “exporting” the search results, which includes titles, authors and abstracts. These result entries are regarded as “articles”, having a similar structure and length as the technology news websites, and can be used in the following steps. ArXiv also supported exporting, but is mostly targeted to Computer Science, only having 3 papers related to Itanium.

Article filters

Unsurprisingly, most articles and papers on these platforms are not related to Itanium. So generally it is important to filter these articles based on their relevance to Itanium, through filtering based on keywords or titles. However, sometimes the opinions about Itanium do appear in seemingly unrelated articles: For example, in an article about the Pentium Division bug some author also briefly express their concerns about similar issues occurring in the Itanium chips. Here, Itanium would not be in the title nor keywords, but still did express a judgment of Itanium. To also include these opinions, a slightly more lenient ’relevance to Itanium’ was used to filter the data.

Preferably, all articles even with weak mentions of Itanium or IA-64 are included in this study. However, some of these news websites have restrictions on techniques to retrieve the data (e.g. not allowing web scraping), or are infeasible to scrape completely within the time-frame, making the relevance search more attractive. Because of this, three steps of different techniques are used to filter the articles:

1. Fetched all articles, locally searched for terms (Itanium, IA-64). Locally extract titles, dates, keywords and article bodies
2. If infeasible, fetched only articles based on returned web results. Locally extract titles, dates, keywords and article bodies
3. If infeasible, export titles, dates, keywords and abstracts into a CSV

For the first approach, scraping the entire website was only possible for Tom’s Hardware and AnandTech. Using a rate-limiter of one request per 45 seconds makes sure that the automated HTML downloader does not overload the website.

For the second approach, the technology news websites (CNET and The Register) were infeasible to trivially index through a URL march. So instead, all articles were indexed through a keyword search for “Itanium” or ”IA-64” using their default search functionality and listing all article URLs returned by the search indexer of relevant articles to Itanium. Then, the (unique) article URLs were listed and fetched, to obtain the article HTML.

Further reads on the ethical and legal considerations can be found in the sub section 3.1.2 below.

For the third approach, the IEEE Xplore and Scopus simply use a user-friendly export all search results button, which has no further ethical or legal considerations. The trade-off is just that it has a rather strict relevance filter.

3.1.2. Ethical and legal considerations

Collecting and analyzing articles does raise some ethical implications for the website and article owners. These are implicit stakeholders in this project. The website may be network congestion loaded through downloading these thousands up to tens of thousands of articles. Additionally, there may be some ethical and legal implications in the redistribution of the articles. In this study, the articles are (temporarily) cached on an academic drive before being analyzed. Some parts of the analysis involves sending the article to a Google Large Language Model Gemini. This data will be used by Google (a subsidiary of Alphabet Inc.) to train their LLM’s in the future, as stated in their user agreements. These technology and research websites have not opted out to allow their data (being the articles) to be part of this training process at Alphabet Inc, meaning that this methodology adheres to their Terms of Service.

To mitigate these potential issues, the HTML scraper (e.g. download script) can be modified to hold a certain rate-limit (e.g. requests per second, bandwidth limitations) to make sure the website is still operational for its users. Secondly, while redistribution can be legally dubious, it is important to adhere to their Terms of Service in using the content for text generation (through Google Gemini), and none of their direct content will be published. Only keywords and statistics of the sentiment towards Itanium is extracted and publicly shared in this study for academic ends, and is not used for commercial or illegally redistributional ends.

3.1.3. Automated Sentiment Analysis using Large Language Models

With all these articles collected, the data processing starts. The chosen LLM was decided early on to be Google Gemini 1.5 Flash, due to its relatively great reasoning performance given it is a free publicly available model [2]. Another free contender was Meta Llama 2.0 or Llama 3.0, which are open-source. However, after multiple processing runs, the model used up all computing resources, yet still was slow and seemed to under-perform, relative to Gemini 1.5 Pro/Flash. This low performance was probably caused by the fact that only the smallest model, Meta Llama2 7B would run on my system. Meta Llama2 13B and Llama 70B exceeded the capacity of the research system equipped with 10GB RTX 3080. The Google Gemini 1.5 Flash has a free-to-use rate-limit of 15 requests per minute and 1500 requests per day. Surpassing these free rate-limits will be billed according to their API pricing model: $0.125 per one million input tokens and $0.375 per one million output tokens.

The first and most basic study is a sentiment analysis of the full paper. This consists of two sections: Prompt engineering and data extraction.

Prompt engineering

Given that the Gemini 1.5 Flash had the best performance for free model, the study is limited to use a unidirectional chat model. Meaning the prompt should be formatted as a query to an agent, as opposed to text completion. The prompt is given below:

```
f"""Given this article:
Title:
{title}

Article:
{body}

I want you to analyze the sentiment of this article towards Itanium. Can you anwer by first
    extensively analyzing the topic/goal of the article, then explain how positive or
    negative they talk about Itanium, concluding with a grade between brackets square []
    according to the scoring I provide below?
Please choose the most fitting Sentiment score
[-2] - Negative (Clearly shedding negative light on Itanium)
[-1] - Signs of critique (Sceptical of Itanium, or indirectly negative)
[0] - Neutral (Indifferent or no answer)
[1] - Somewhat positive (Advocate of Itanium, or indirectly supportive)
[2] - Positive (Putting Itanium on a positive spotlight)
"""
```

The prompt consists of four techniques: Context, explicit task, steps to approach and formatting. The context is given in the string interpolation `{title}` and `{body}` variables. The explicit task is defined as `I want you to analyze the sentiment of this article towards Itanium`. The following steps are as explicit instructions on how to structure the response:

1. `Can you anwer by first extensively analyzing the topic/goal of the article`;
2. `then explain how positive or negative they talk about Itanium`;
3. `concluding with a grade between brackets square [] according to the scoring I provide below`.

Then finally the formatting and grading system `grade between square brackets [] according to the scoring I provide below`, followed by the grading system for more explicit categories to ensure consistency between runs.

Instructing the Large Language model to first explicitly state its reasoning before answering often improves the reasoning capabilities. This scratchpad allows the Large Language model to ”think out loud” through a technique called chain-of-thought prompting and catch small mistakes before jumping to conclusions [34][33]. Additionally, this chain-of-though prompting makes the reasoning behind certain decisions more transparent for debugging [27]. The prompt also consists of a zero-shot context, which is generally more fitting with explicit instructions and self-explanation, giving the model the room to reason its own ways rather than following the exact structure of reasoning for each paper.

Data extraction

The chat model returns a “response text”, which, if it followed the instructions faithfully, should contain the concluding grading between square brackets. This can sometimes be parsed incorrectly, but simply regenerating a new response with the same prompt will generally fix this issue. This has been built into the prompt generator. When a prompt is correct, it should only contain one square bracket pair with the sentiment score, such as in the example below.

```
"""## Sentiment Analysis of the Article Regarding Itanium:

**Topic/Goal of the Article:**

The article focuses on SGI's strategic shift towards Linux and Intel's Itanium processors.
    The primary aim is to inform about:

* SGI's move away from MIPS chips and Irix OS towards Itanium and Linux.
* The potential for a government-funded supercomputer running Linux.
* SGI's efforts to contribute to the Trillian project, aiming for Linux on systems with
    hundreds of processors.
* The wider adoption of Itanium by companies like HP and the development of Itanium-based
    supercomputers (like VA Linux's "Chiba City").

**Sentiment Towards Itanium:**

The article generally presents a **positive** view of Itanium, highlighting its potential and
     adoption by major players.

* **Positive aspects:**
    * SGI is transitioning to Itanium, indicating confidence in its future.
    * The article emphasizes the lower prices for chips made in larger quantities, which
        Itanium is designed for.
    * HP, a major player in the industry, is also embracing Itanium.
    * The article mentions VA Linux's "Chiba City" supercomputer, showcasing Itanium's
        capabilities.
    * It is pointed out that Itanium-based systems are expected to be available in the second
         half of 2000, indicating anticipation and excitement for the technology.

* **Neutral aspects:**
    * While the article mentions the delays faced by Itanium, it doesn't frame them as
        detrimental to the technology's future.
    * It also briefly mentions the spin-off of SGI's Cray Research supercomputer division,
        but it's not directly related to Itanium's prospects.

* **Negative aspects:**
    * The article does mention delays to the chip, but this is presented as a fact rather
        than a negative criticism.

**Sentiment Score:**

Based on the above analysis, the article's overall sentiment towards Itanium is **[1] -
    Somewhat positive**. While not explicitly praising Itanium, the article highlights its
    potential and adoption, indicating a positive outlook."""
```

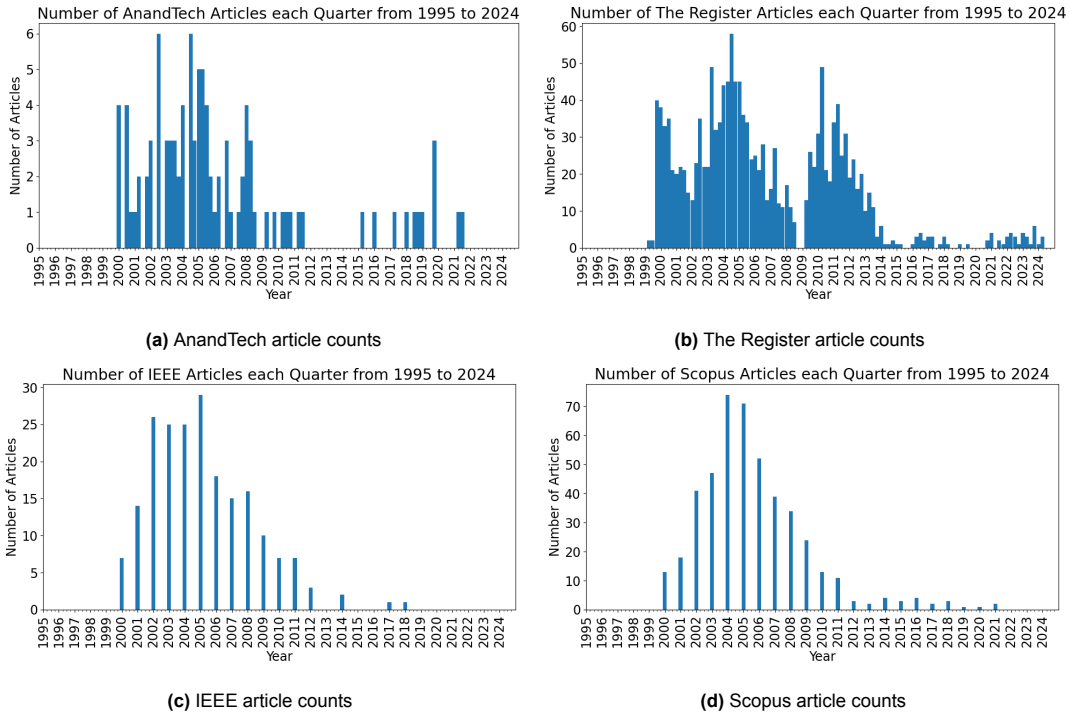
3.2. Results

The collected papers are grouped and counted per year, to get a sense of the popularity of Itanium along the years. The graphs in Figure 3.2 show that the popularity of Itanium peaked around 2002-2006, which was right after its release date, June 2001. Note that both Scopus and IEEE’s export only denotes the publication years, not the months or quarters. The graphs in Figure 3.3 show the yearly

distribution of the sentiment values.

There seems to be a common rise of interest in Itanium in each of the graphs as expected, due to the snowballing of higher popularity leads to more articles, leading to higher popularity. However it seems to peak around 2003 or 2004, where it begins to taper off, and slowly begin to fall.

The general sentiment towards of Itanium seems to be more negative than positive in the news websites AnandTech and The Register. The articles found on the research repositories seemed to be more positive, constructive view on Itanium. IEEE and Scopus does seem to lean more into the objective sentiment towards Itanium. Upon inspection, the negative sentiments towards Itanium only seem to be regarded “negative” in favor of another comparable architectures, rather based on negative opinions of Itanium. Additionally, IEEE even seemed to have a positive trend line from 2004 up to 2010.



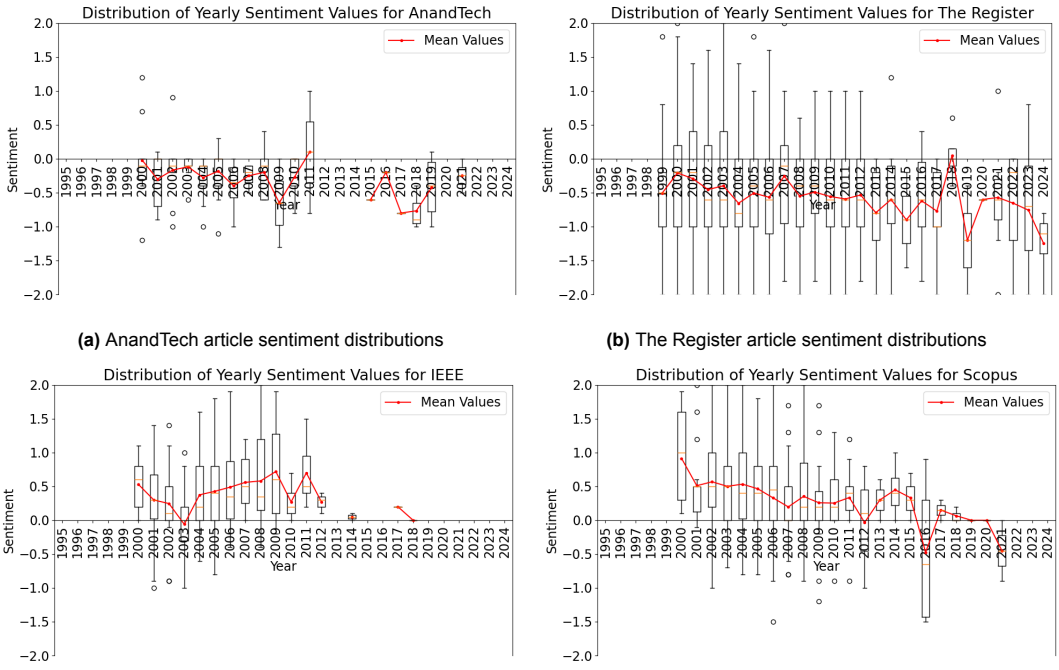
(a) AnandTech article counts

(b) The Register article counts

(c) IEEE article counts

(d) Scopus article counts

Figure 3.2: Comparison of popularity of Itanium articles



(a) AnandTech article sentiment distributions

(b) The Register article sentiment distributions

(c) IEEE article sentiment distributions

(d) Scopus article sentiment distributions

Figure 3.3: Comparison of popularity of Itanium articles

3.3. Conclusion

The automated market sentiment analysis of Itanium provides valuable insights into a some of the CPU's causes for success and failure. To reiterate the relevant research questions for this study:

R3.7. How aware was the public of Itanium's existence and purpose?
R3.8. What does Itanium's popularity spikes imply about its public reception?

These will each be answered in the following subsections.

3.3.1. Was the public aware of Itanium's existence and purpose?

As the popularity graphs point out, there is a notable peak in article counts around 2003, which suggests that Itanium did seem to catch the public's attention. However, as mentioned in the previous answer, the performance and price concerns seem to mostly be dependent on the use cases. This shows that there was some uncertainty about the purpose of Itanium, that it did not clearly (or correctly) convey its target audience.

3.3.2. What does Itanium's popularity spikes imply about its public reception?

According to this analysis, instead of a general denunciation of Itanium, it seemed to have a positive peak at the start of its release. However, quickly after it seemed to dive down in popularity and acclamation. Some historical events, mentioned in chapter 2, may have played a role in this:

- April 2003: The release of AMD64 in Opteron, the 64-bit extension of the already popular x86.
- February 2004: The release of EM64T in Xeon/Pentium 4, Intel's variant of x86-64. This marks the moment Intel moving away from promoting the need for a completely new architecture to upgrade to 64-bit, towards allowing users to upgrade the existing x86 family.
- December 2004: Intel's press release about repositioning Itanium to High-end users, instead of all-round.

Some of the positive sentiment spikes around 2012 may have been caused due to new breath being blown into Itanium, with the hype and release of the Poulson architecture in November 2012. It was presented in a paper called "A 32nm 3.1 Billion Transistor 12-Wide-Issue Itanium Processor for Mission

Critical Servers.” Being the largest chip to date, and having many new technological advancements in its design, it seemed to have caught the attention of many.

3.3.3. Other observations

Some of the key points that are brought up in these articles, is performance, development delays, software support and price. Performance is a large metric and driver for Itanium’s success, so it is not surprising it is mentioned as often as it is. However, while the topic is mentioned often, the sentiment in the reports vary widely; there are approximately as many articles suggesting great performance as others suggesting terrible performance. Other often mentioned topics are the development delays as well as the lack of software support, met with a negative sentiment as anticipated. The sentiment towards the price is also quite variable, with some saying Itanium is worth the money in specific use cases, but is an expensive endeavor in most other cases.

4

Performance Comparative Analysis

As indicated in the success analysis in section 2.6 and market analysis section 3.3 a significant factor in Itanium's widespread adoption was its performance. This chapter aims to answer the following research questions through a performance comparative analysis:

R3.1. How does Itanium's SPEC2000 performance compare with other architectures?
R3.2. How does EPIC compare with Superscalar in theoretical performance comparison?

Itanium's performance received significantly mixed criticism and is one of the key topics to understanding its market failure. This section divides into two parts: the implementation and the compiler.

4.1. Background on theoretical EPIC ILP

The fundamental definition of CPU performance is rather straightforward: execution time in seconds. Historically, CPU designers leveraged advancements in hardware which sustain a 35% improvement each year [16]. However, this trend of frequency scaling faced significant challenges between 2000 and 2005 due to power and thermal limitations. Despite these challenges, engineers continued to enhance performance through various strategies:

- Increasing computations per instruction (e.g., a Multiply circuit is strictly stronger than Add, as it generally consists of multiple adders together);
- Increasing instructions per cycle (e.g., employing techniques like pipelining and Single Instruction Multiple Data (SIMD));
- Implementing multitasking across multiple instruction streams (e.g., thread-level parallelism);
- Distributing tasks across multiple systems or processors.

All Instruction Set Architectures (ISA) are capable of multitasking or running on multiple processors or systems. But the design of an ISA can significantly influence the efficiency of computations per instruction and instructions per cycle. During the development of Itanium, the popular ISA paradigms were Complex Instruction Set Computing (CISC) and Reduced Instruction Set Computing (RISC). They each have their strengths and weaknesses, but in terms of Instruction Level Parallelism (ILP), both are quite limited. These two approaches can roughly be defined as the difference between "Increasing computations per instruction" and "Increasing instructions per cycle", where CISC has stronger, more complex instructions, while RISC has smaller, simpler instructions. RISC is generally better at utilizing its functional units to the fullest, as it has a clean and streamlined instruction pipeline for instruction level parallelism. However, another approach that trumps both in terms of ILP is Very Long Instruction Words (VLIW). Figure 2.2 compares CISC, RISC and VLIW and how they are able to improve performance through ILP.

VLIW feeds instructions into different instruction units in parallel, which theoretically be scaled to any amount of parallel instruction streams. But in practice there are not that many workloads that have

such amounts of parallelism to make full use of these parallel units. So the main challenge of VLIW is to fill execution units with (effective) work. Table **??** compares the different programming paradigms. for their strengths and weaknesses.

Table 4.1: Comparison of CISC, RISC, and VLIW/VLIW Architectures

Feature	CISC	RISC	VLIW/EPIC
ILP	Multiple cycles per instruction	One instruction per cycle	Multiple instructions per cycle
Instruction Size	Variable length	Fixed size	Fixed size
Code Density	High code density	Lower code density	Lower code density
Architectural Registers	Fewer registers	More registers	Many registers
Compiler Complexity	Generally simpler compilers	More complex compilers	Most complex compilers
Pipelining	Complex pipelining	Simplified pipelining	Superpipelined
Decode	Complex decode	Simple decode	Simple decode
Energy Consumption	Higher	Lower	Medium
Debug Complexity	Medium	Low	Medium

ILP in VLIW is best due to its superpipelined system. CISC and RISC both use scalar pipelines, of which RISC is better optimized due to its simpler instructions.

CISC code density is generally best due to its variable size, allowing more frequent instructions to be shorter. RISC instructions are fixed-width, so even very simple instructions (such as register - register move, or return subroutine) still require just as many bits as the most complex instruction. The same applies to VLIW, however VLIW also adds additional padding NOPs. Some systems like SCISM [32] mitigate these NOPs by using a “stop bit” to encode the end of a group of parallel instructions, which greatly improves code density, close to the scale of RISC.

Compiler complexity of CISC is generally manageable due to its complex high-level instructions. Idioms like `add [rax],5` are simpler to translate into CISC code than RISC or VLIW. Due to RISC’s reduced count of instructions, there often is not a direct idiom translation into instructions, which requires slightly more work from the compiler to break down high-level computations into smaller instructions and to optimize [16]. VLIW offloads the most work to the compiler, requiring it to perform dependency management to avoid hazards - such as read-after-write (RAW), write-after-read (WAR) and write-after-write (WAW). This adds tremendous amount of complexity to the compiler optimization.

Decoding CISC instructions is a complex task, due to its variable instruction lengths and many addresing modes. RISC and VLIW generally have the simplest decoding due to their fixed encoding and usually field-aligned encodings.

CISC generally has a higher energy consumption due to its additional complexity in the decode and microcode instructions. While this is not a universal rule, it is generally accepted that RISC has a lower energy consumption [6] [30], executing the same instruction stream. VLIW is also more energy efficient than CISC, but due to the larger unit count and wasted NOPs it would arguably runs with slightly slower energy efficiency than RISC.

4.2. Methodology

The performance characteristics of EPIC and Out of Order superscalar execution is compared in theoretical calculations-based model. The calculation-based comparison is done by taking other factors than performance into account: Dynamic data patterns, units utilization, and compiler decisions. It proves the inefficiencies of EPIC compared to dynamically scheduled CPU’s in the case of uncertain dynamic data patterns.

In a second comparative analysis study of Itanium weighing up to its peers, the SPEC2000 results are collected and discussed in Section 4.6

4.3. Theoretical performance upper bound

Itanium's EPIC aims to deliver high computational performance by executing many instructions in parallel. While this heavily depends on the compiler to extract instruction parallelism in practice, a performance upper bound analysis can bring insight to Itanium's theoretical maximum performance.

Using Intel's publications on Itanium such as software manuals and architecture overview, some exact specifications can be used in these calculations:

- Two bundles (6 instructions) are fetched from cache per cycle.
- Up to six instructions are issued to the execution backend per cycle.
- Execution backend consists of: three branch ports, two integer ports, two memory ports and two floating point ports.
- There are three branch units, four integer arithmetic and logic units (ALU's), four multimedia integer units, two load/store units, two floating point execution units.
- ALU's are globally bypassed accross all four ALU's.
- Simple integer operations (e.g. `add`, `sub`, `and` and `cmp`) have one cycle latency.
- Floating point units have a five-cycle delay to calculate floating point multiply-accumulate (FMAC) instructions, but are fully pipelined allowing a new instruction to be issued each cycle, resulting in one FMAC instruction executing per cycle throughput. These floating point units comply to the IEEE 754 standard.
- Integer multimedia units have two-cycle delay and are also pipelined.
- Both the Integer multimedia units and the Floating point units support single instruction multiple data (SIMD) instructions, executing the same operation over multiple packed data points in parallel. Having 64-bit vectors, it allows one 64-bit integer, two 32-bit integers, four 16-bit integers or eight 8-bit integers to be packed in one vector for Integer vector units, or one 64-bit double-precision float, or two 32-bit single-precision floats packed in one vector for Floating point vector units.
- The system is clocked at 800 MHz.

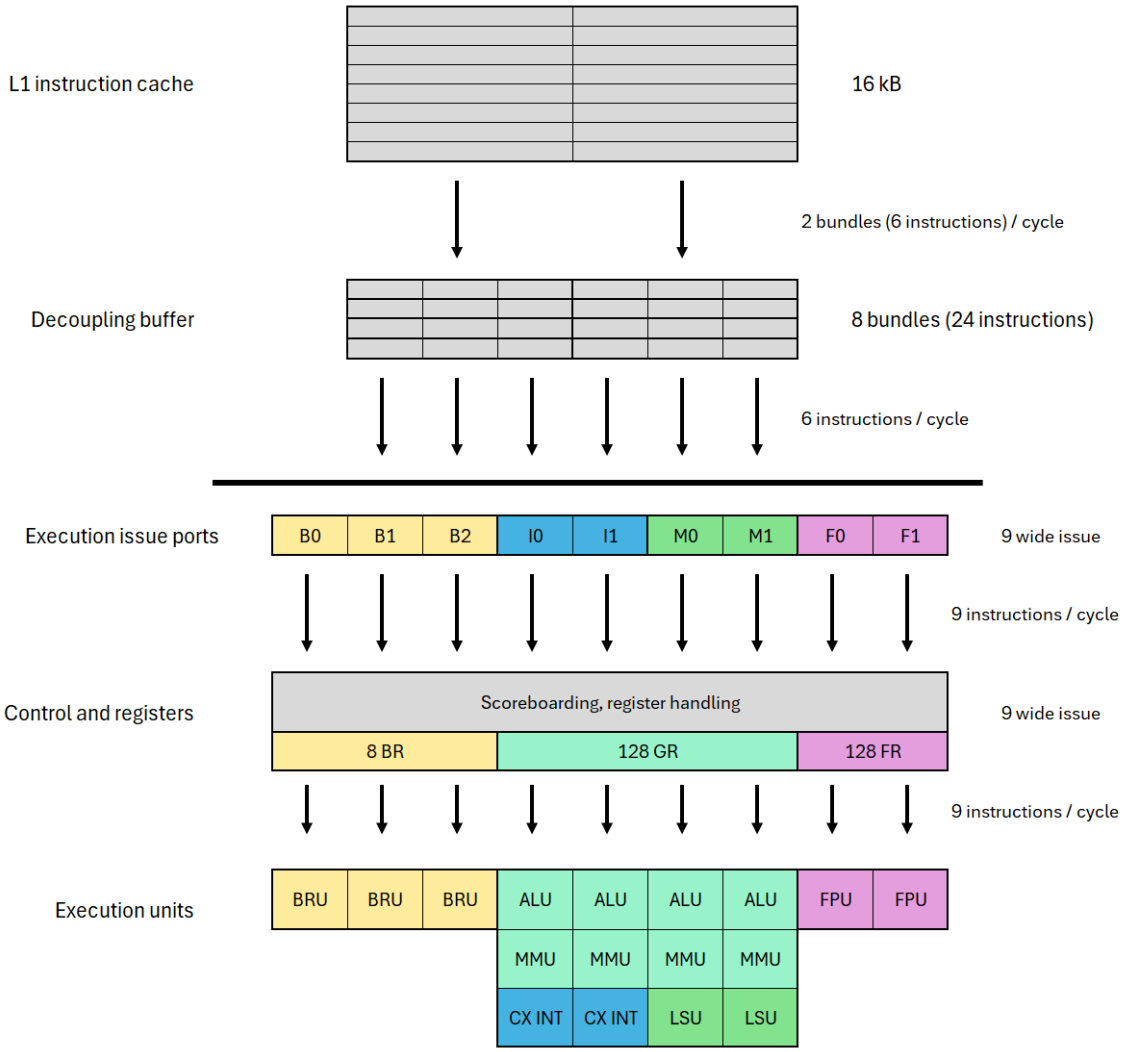


Figure 4.1: Itanium execution flow overview

4.3.1. Peak execution rate

The peak execution rate in basic arithmetic is calculated by first determining the performance bottleneck. As illustrated in figure 4.1, Itanium has four parallel ALU's, which together can deliver four operations per cycle. Since each of the previous steps in the execution flow is theoretically able to keep up with feeding these ALU's with four instructions per cycle, the bottleneck is actually the ALU execution units.

Given a full instruction pipeline and no instruction dependencies, Itanium can deliver four integer instructions per cycle. With this system clocked at 800 MHz, it could deliver 3200 MOPs (Mega operations per second).

When making use of the multimedia SIMD instructions, multiple packed data points can be executed in one operation. Itanium has four multimedia units, which each support four different formats: 1x64-bit, 2x32-bit, 4x16-bit and 8x8-bit. While these units do have an execution latency of two cycles, it can issue one instruction per unit per cycle due to execution unit pipelining. This results in 3200 MOPs, 6400 MOPs, 12.8 GOPs and 25.6 GOPs of 64-bit, 32-bit, 16-bit and 8-bit data points, respectively.

Similar to the integer multimedia units, Itanium can deliver one double-precision floating point or two single-precision floating point operations in one cycle using packed SIMD. Having two parallel FPU SIMD units and regarding FMAC (Fused Multiply and Accumulate) as two separate floating point oper-

ations (`d = a * b + c`), it can deliver 3200 MFLOPs (Million Floating point operations per second) in 64-bit double-precision floating point and 6400 MFLOPs in 32-bit single-precision floating point.

This closely resembles the values in the synthetic benchmark run by [5], which reports execution rates of around 6000 MFLOPS in single-precision floating point. They show a large (100x100 up to 1500x1500) square matrix multiplication comparison between Itanium 2 and Pentium 4.

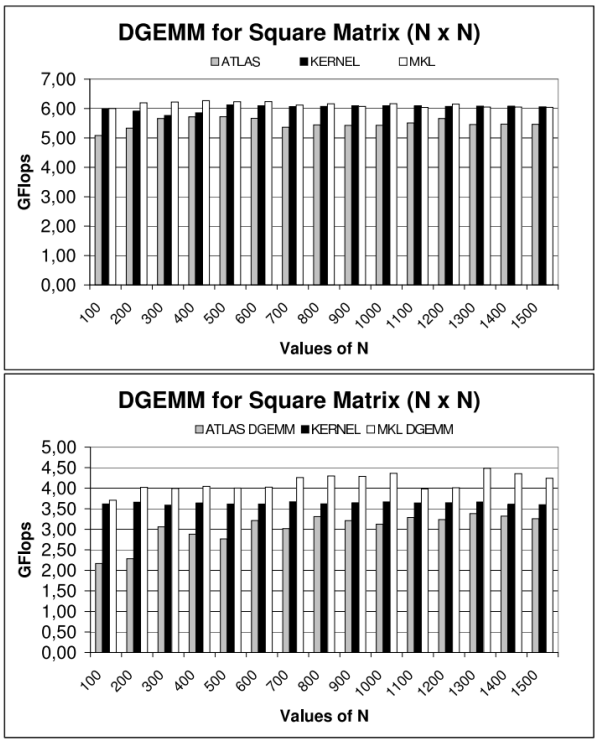


Figure 4.2: Synthetic benchmark comparison on Itanium (top) and Pentium (Bottom)

4.4. Theoretical Comparison with Out of Order Superscalar

Since Itanium is an EPIC machine, it relies on the compilation techniques to reach the same levels of ILP as an Out of Order superscalar RISC or CISC architecture. However there are some cases where this is theoretically impossible. For example, the following Pseudo assembly shows an instruction sequence which an OoO design could speculatively optimize dynamically in hardware, but EPIC would need to optimize statically in software.

```
1 mul r1=r2,r3
2 cmp.eq p6,p7=0,r1
3 (p06) mul r4=r5,r6
```

This can be compiled in two different ways: Eager speculation or lazy in-order, as depicted in figure 4.3. For this example, we will assume the compiler for an OoO system generate instructions similar to the lazy implementation.

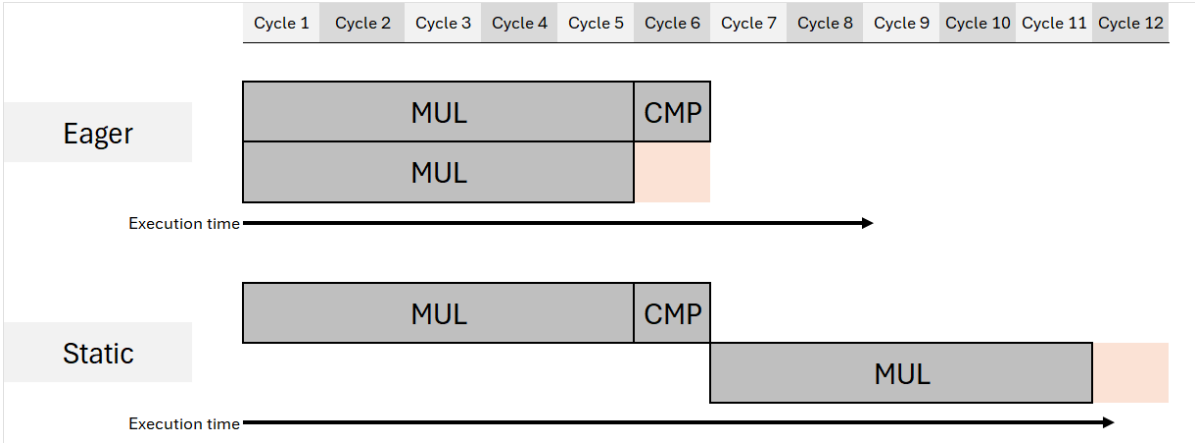


Figure 4.3: Eager (Speculative) vs Lazy (In-order) execution

Table 4.2: Comparison of Lazy, Eager, and Dynamic Execution Strategies

	Lazy		Eager		Dynamic	
	Often nx	Often x	Often nx	Often x	Often nx	Often x
Units used	1	1	2	**2**	2	1
Latency	6	**11**	6	6	6	6

Table 4.2 shows that which ever one the compiler chooses for EPIC, if the data makes the `CMP` to more frequently result in executing the second multiply than not, then the lazy implementation is (unnecessarily) slow, by roughly 2x slower. If the `CMP` turns out to be more frequently "Do not execute the second multiply", then the CPU is wasting resources on an instruction that is never/rarely going to be used. The compiler can try to predict which of the two cases is more likely going to occur, but that can only be said for certain at run-time. In both cases, the OoO implementation will execute the second multiply only when needed, and will have no delay when executing. In no case is the lazy/eager execution faster or more efficient than the OoO, especially if the data patterns are known beforehand. The OoO does add hardware overhead in the form of branch prediction, and OoO execution circuitry overhead.

4.5. Emulated performance characteristic analysis

This section explores the development of the custom IA-64 emulator, designed to execute IA-64 binaries and analyze run-time behavior. Although the tool did not reach a functional stage where definitive results could be obtained, the process yielded insights into the design complexity. This section aims to detail the challenges encountered, the knowledge gained, and the provisional results.

In analyzing the performance characteristics of Itanium, the most evident approaches are either emulation or simulation. While these terms are often used interchangeably in the context of running foreign instruction sets, the terms have slightly different definitions. Emulation aims to copy the behavior of the instruction set execution, while a simulator copies the system. Generally, emulators are used if the goal is to run the executable, while simulators are used to analyze the system behavior during runtime. For the purpose of this study, which is to analyze the system's performance characteristics, a simulator would be more appropriate.

4.5.1. Existing IA-64 simulator and emulator ecosystem

After extensive online searches through forums and search engines (including automated searches run through Large Language Models, such as OpenAI's GPT's and Google Gemini 1.5 Pro's online search functionalities), the current ecosystem of IA-64 simulators and emulators seems to be lacking. With only the following showing up as results: Gem5, QEMU, Ski, Rosalia64 and ia64sim.

- Gem5: A popular instruction set simulator with detailed timing tracing. However, it does not

support IA-64.

- QEMU: A popular open-source emulator collection, supporting most existing ISA's. For some ISA's it also includes timing tracing at a system level. It does not support IA-64 officially, but some attempts were made according to its commit history, forks and official discussion boards.
- Ski: An IA-64 simulator developed by HP. It does not compile on my system or virtual machine, due to the how old the software is (2008). Compilation raised errors of packages it could not download as the URLs no longer exist.
- Rosalia64: An often referenced, yet non-existing IA-64 compiler. The open-source project only consists of few files which partially implements two IA-64 instructions.
- ia64sim: A partially implemented open-source project. While it seems to work on simple tests, it is an emulator, so it does not give timing information about its runtime.

4.5.2. Custom IA-64 simulator implementation

Due to the currently infantile state of IA-64 simulation ecosystem, creating a basic self-made simulator based on Itanium and IA-64 documentation could be useful to achieve this specific goal of analyzing performance characteristics. Intel and HP have released two useful documents covering instruction and system behavior: `Intel IA-64 Architecture Software Developer's Manual` and `Intel Itanium Processor Reference Manual for Software Optimization`, respectively.

The simulator aims to mimic Itanium execution properties, such as instruction latencies, branch delay slots, feed-forward delays and instruction issue splitting. Most of these phenomena are extensively documented in the software optimization manual. While this evidently does not mimic Itanium hardware exactly, it serves as a foundational platform to approximate how the Itanium architecture might behave under different computational loads and scenarios. This approach allows for a simplified yet insightful exploration into the instructions per cycle (IPC) rates, taking the dynamic behaviors into account through execution of instructions on the CPU, such as:

- Instruction Fetch and Decode: Emulating how instructions are fetched from memory, filled in the instruction buffer, and decoded.
- Execution Latency: Simulating the time it takes for different instructions to execute, which is vital for understanding the performance bottlenecks in high-complexity computations.
- Branch Prediction: Implementing a synthetic branch predictor which can simulate the latency and speculation costs of taking the wrong branches
- Execution unit Allocation and latency: Simulating the execution unit allocation and latencies, which includes issue splitting, execution latencies and feed-forward delays.

While this system partially implements the Itanium hardware, it cannot exactly replicate it, due to specification availability and project time constraints. The simulator does, however, guarantee theoretical upper-bound performance, as it does not implement systems that induce other delays:

- Cache accesses are assumed to have one cycle delay (in practice, the delays are 2+, depending on whether it is a L1, L2 or L3 hit/miss);
- Each instruction is idealized to take exactly one cycle of execution delay;
- Conditional and unconditional control flow instruction have 0 cycle delay bubbles;
- The rotating register stack engine has infinite size and no allocation/deallocation delay;
- The instruction pipeline consists of only two stages: One cycle to fetch and buffer instructions, and one cycle to execute and write back the results, instead of the ten-staged pipeline (of Itanium Merced) [24]

4.5.3. IA-64 simulator benchmarks

The executed programs are small algorithms, due to the simulator's implementation limitations. The following algorithms were chosen for their simplicity, yet ubiquitousness:

- memcpy: the common implementation of memcpy, a common C standard function. It copies the memory bytes form a source pointer to the destination;

- indexof: a commonly used function, to find the index of an element in an array;
- arrsum: another commonly used function, which calculates the total sum of each element in the array.

Below are the implementations of these three functions:

```
void memcpy(void *dst, void *src, int n) {
        for (int i=0; i<n; i++) {
        ((char*) dst)[i] = ((char*) src)[i];
    }
}
```

```
int indexof(long *arr, int len, long v) {
        for(int i = 0; i < len; i++) {
        if(buff[i] == v) return i;
    }
        return -1;
}
```

```
long arrsum(long *arr, int len) {
    long sum = 0;
        for(int i = 0; i < len; i++) {
        sum += val;
    }
    return sum;
}
```

Each of these functions are compiled using the most recent installment of the Linux GCC compiler which supports compilation to IA-64 (`ia64-pc-linux-gnu-gcc (GCC) 9.5.0`). The compiled binaries are analyzed using the standard object code analyzer (`GNU objdump (GNU Binutils for Ubuntu) 2.38`), to output the raw instruction bytes, omitting the ELF (Executable and Linkable Format) headers. All three are compiled and analyzed using the following Linux bash commands:

```
gcc-ia64 memcpy.cpp -o o -c -O3 ; objdump -d o > memcpy.o3.txt
gcc-ia64 indexof.cpp -o o -c -O3 ; objdump -d o > indexof.o3.txt
gcc-ia64 arrsum.cpp -o o -c -O3 ; objdump -d o > arrsum.o3.txt
```

This uses the GCC `-O3` flag to indicate full optimizations. The `-c` flag tell GCC to compile to object code without linking, as this binary will not be executed on this system. The objdump `-d` flag indicates to extract the disassembled instructions of the executable code sections, thus extracting only the instruction binaries.

As a simple test, each of these functions are called with the list:

$$arr = \{5, 1, 7, 2, 3, 8, 9, 0, 4, 6\}; len = 10$$

and the following arguments:

$$memcpy(dst, arr, len * 8)$$
$$indexof(arr, len, 0)$$
$$arrsum(arr, len)$$

Where the `dst` array is a pointer far away in memory, as a simple test of the function and asserting no pointer overlap, as according to the C memcpy standard [20].

These instructions are then interpreted and simulated in the custom simulator. The simulator is modified slightly to debug output statistics such as the amount of instructions executed in each cycle.

Table 4.3: Performance Metrics of memcpy, indexof and arrsum in the custom Itanium simulator

Function	Total Cycles	Instructions per Cycle	NOPs per Cycle
memcpy	170	2.529411765	0.541176471
indexof	27	2.037037037	1.074074074
arrsum	24	2.5	0.625

4.5.4. IA-64 simulator results

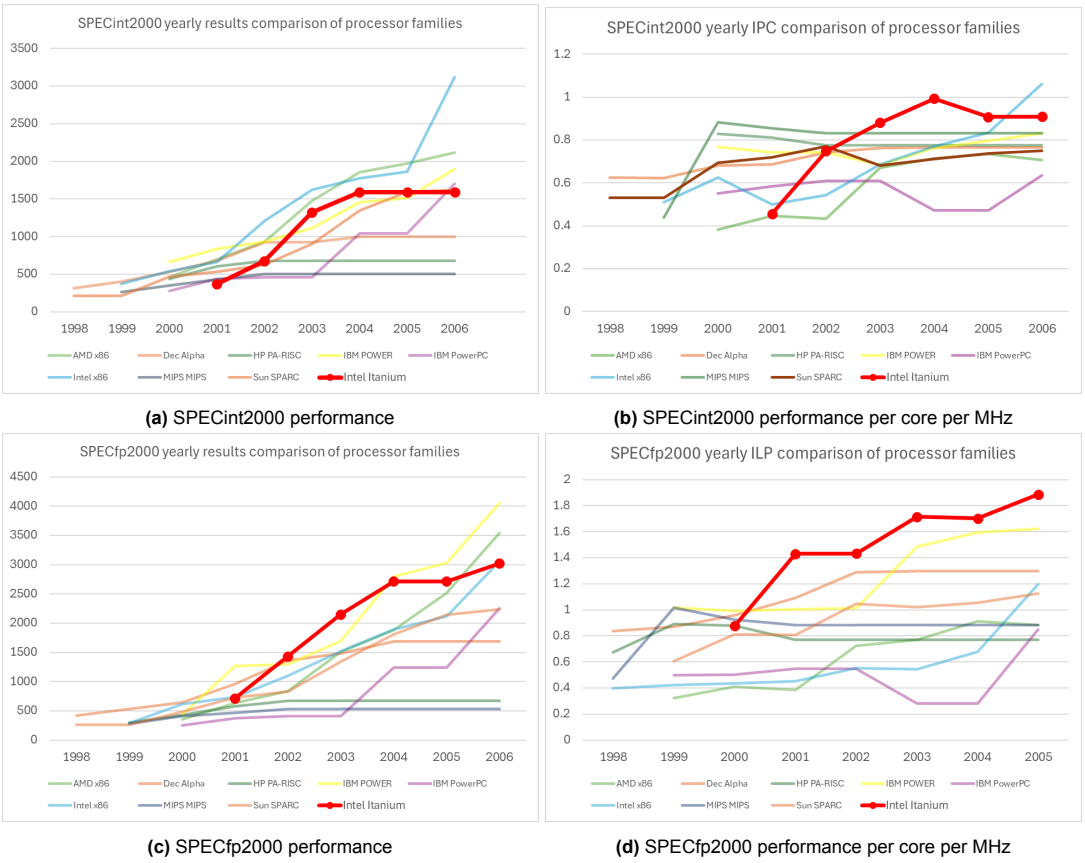
The results of the simulator's executions are listed below in table 4.3.

The results indicate that all three functions run at about about 2 or 2.5 instructions per cycle. The indexof and arrsum functions take only relatively few cycles to execute, which indicates good execution performance. However, the compiled memcpy function takes 170 cycles, significantly longer than the other despite looking similar in operation, just a single for-loop. This is because the GCC compiler emits code that copies the bytes over byte-by-byte, rather than per 64-bit chunk. This effectively slows down execution by approximately 8x.

Note again that these results are still approximations, as the simulator is still missing many aspects of the hardware execution. The exact specifications of this hardware are not publicly available, most likely because it contains proprietary Intel hardware implementation secrets. The custom simulator can still give an approximation of an idealized system, giving upper-bound performance guarantees.

4.6. SPEC2000 benchmarks comparative analysis

SPEC2000 is a well known performance evaluation standard developed and maintained by ATA. The benchmark primaraly assesses the execution speed of a processor, but also factors in other measurements such as energy consumption and system specifications. There are existing records archived on the SPEC website [28]



(a) SPECint2000 performance

(b) SPECint2000 performance per core per MHz

(c) SPECfp2000 performance

(d) SPECfp2000 performance per core per MHz

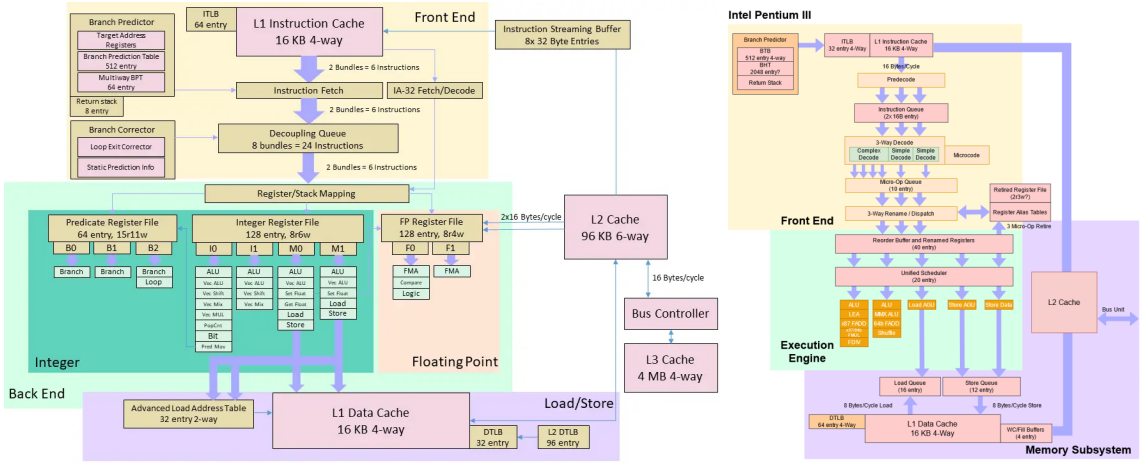
Figure 4.4: SPEC2000 comparisons on the basis of absolute performance and normalized by MHz and cores

According figure 4.4a, the absolute performance of x86 cores seemed generally better than Itanium. Since one of the main focuses of Itanium is to introduce more ILP, it would be an interesting case study to also inspect how well it performs in that regard. The graph in figure 4.4b shows that Itanium did indeed have higher performance per cycle on average over the runtime. However, its slightly lower performance than x86 competitors is most likely caused by their lower clock-frequencies. Itanium's absolute lower performance could be attributed to the limitations of extracting ILP from software, which does not balance out its lower frequencies.

The graphs in figure 4.4c and figure 4.4d shows that Itanium had very competitive floating point performance. It outperformed most systems at in absolute performance and having the best IPC scores.

4.6.1. Discussion

Most of the Intel x86 CPUs outperforming Itanium in the graph use the P6 micro-architecture, introduced in 1994 in the Pentium Pro. Since the Itanium Merced included a six-wide decode and 11-wide issue as shown in figure 4.5a, compared to close competitor Intel's Pentium III (P6), with only three-wide decode and five-wide issue, shown in figure 4.5b, Itanium should have more parallel units to do more instructions in parallel in theory. In practice, this graph suggests that either there was not enough ILP to extract in software, or the 40-wide Reorder Buffer (ROB) of P6 allows it to extract close to the same amount of ILP. Which when paired with higher clock frequencies (Merced at 800 MHz and P6 at 1400 Mhz) meant P6 would outperform Itanium in most cases.



(a) Merced micro-architecture diagram [8]

(b) P6 micro-architecture diagram [9]

Figure 4.5: Block diagram of Merced and P6 side-by-side

4.7. Conclusion

This chapter discussed two studies, a theoretical execution comparison between static EPIC and dynamic OoO performance and a comparison of SPECint2000 benchmarks. These two studies aim to answer the questions:

R3.1. How does EPIC compare with Superscalar in theoretical performance comparison?
R3.2. How does Itanium's SPEC2000 performance compare with other architectures?

4.7.1. How does EPIC compare with Superscalar in theoretical performance comparison?

The theoretical performance model comparison proved how extracting static ILP for EPIC would result in either the same performance as OoO, or worse in terms of latency, or in use of parallel units. In this example there is no case where static ILP outperform OoO, since the compiler has to make an assumption at compile-time whether the branch will be taken often or not, whereas the OoO hardware can use predictions based on runtime heuristics to decide at dynamically whether the branch is taken or not.

4.7.2. How does Itanium's SPEC2000 performance compare with other architectures?

According to SPEC2000 benchmarks, Itanium did perform quite competitively compared to contemporaries. While not reaching the top level in performance in SPECint2000

4.7.3. Other observations

According to the studies conducted in this chapter, the performance of Itanium was comparable to its competitors. The theoretical performance comparison shows that the peak execution of Itanium rivals Superscalar, with the only downside of potentially more speculative execution being executed than needed. The

5

Static Analysis of IA-64 Binaries

A widely speculated reason for Itanium's lack of widespread adoption was its poor performance due to the compiler not being able to extract enough ILP from the source code of applications. This chapter aims to analyze and discuss the IA-64 compiled binaries, Without being able to run the IA-64 software, there is still a lot of runtime characteristics data in these compiled binaries for IA-64 to answer the following questions:

R3.3. How significant is the proportion of NOPs in typical IA-64 binaries to performance?
R3.4. How do IA-64's program sizes compare with competitors?

5.1. Methodology

The static analysis study consists out of three steps:

1. Data collection: Compiled binary generation or collection;
2. Disassembly: Converting this compiled binary into a more interpretable intermediate language (e.g., Assembly);
3. Analysis: Interpreting and running analyses over this data.

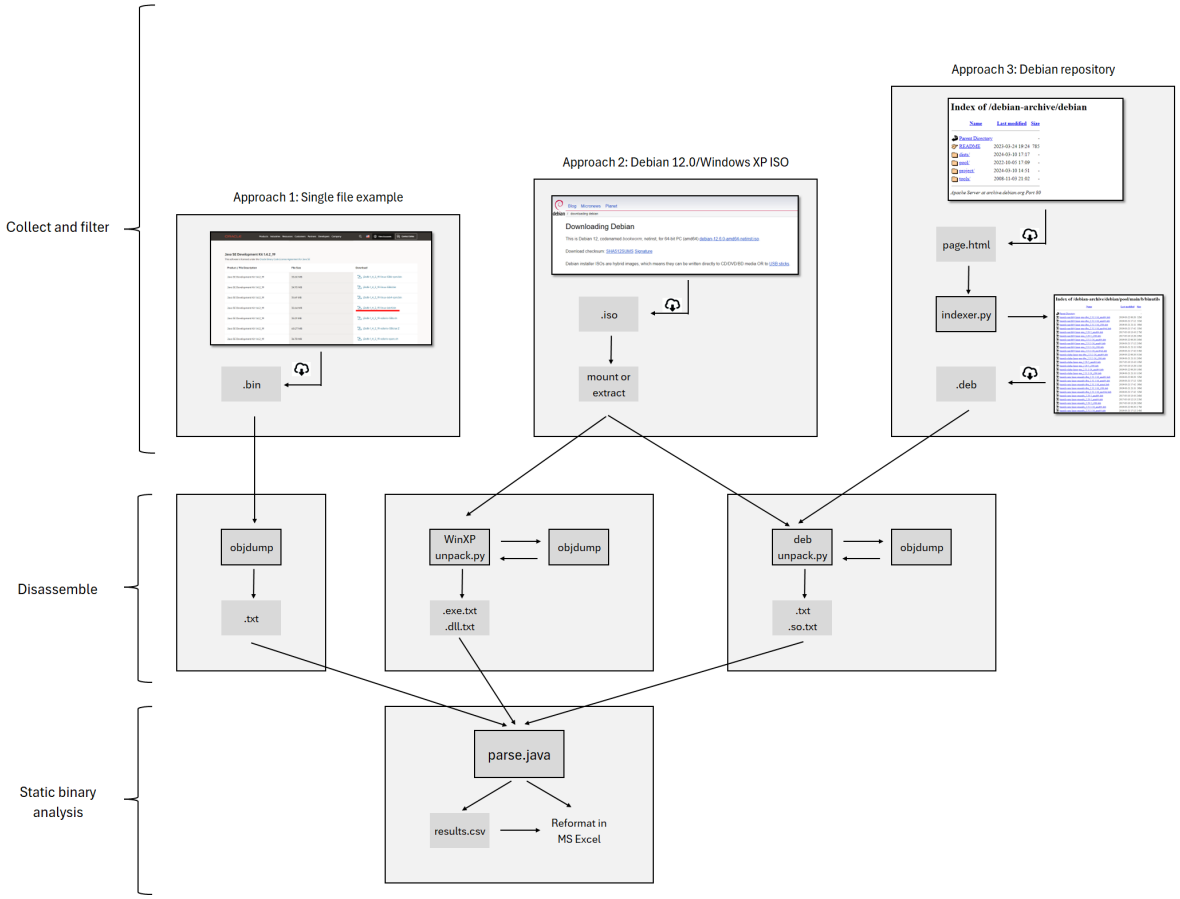


Figure 5.1: Static analysis process overview

5.1.1. Data collection

There are many approaches possible to collect static analysis data. The two main ways are through generation, or through collection. Generating includes compiling source code to machine code binaries or transscription. Collecting data can be done though existing databases, public data sets, and in some cases locally on your own machine. Finding and collecting binaries for popular ISA's such as x86 or ARM is quite trivial: Most (open source) software installers include downloads for these popular architectures. But for a bit less mainstream architectures, such as PowerPC and MIPS, they appear a bit less frequent. IA-64, on the other hand, peaked in popularity around the start of 2000's, then has mostly been dropped from support lists. A few archives did still include IA-64 for downloads, such as Oracle's "Java SE Development Kit 1.4.2_19", which is easily retrievable from their website for both IA-64 Windows and IA-64 Linux. This download includes a short (about 4 kB) section of machine code text. While quite a bit of static analysis can be done here already, the dataset may still be considered as too small.

Since Itanium's goals was to achieve high performance and widespread adoption, it would seem that the aforementioned found dataset (Oracle's "Java SE Development Kit 1.4.2_19") would not suffice. Firstly, it is not a great example of performance-critical code, being a software installer, which mostly interfaces with the OS and would most likely not be heavily optimized as the bottleneck would be the network connection or disk writing. Secondly, it is a very small data set, so it would not suffice to show the "wide-spread" use cases of this ISA.

The next data collection source found was the Linux Debian 12.0 ISO (disk installer). This installer is not directly interpretable, but can only be opened on a Linux machine or using a virtual machine. This has about 300 MB of machine code. This does include quite high-quality wide-spread data, but since it only contains the Linux kernel and standard software (such as `vim`, `apt` and `gcc`), it does not have a

good representation of high-performance code.

To find both high-performance and wide-spread code, just one step further down this path leads to the Linux package installer, Aptitude's repository. This is quite simple to access when a running on an Itanium machine or emulator, but it is also accessible through the Debian FTP (File Transfer Protocol). This can be interfaced using a browser, or through software like `vsftpd` (Very Secure FTP Daemon). The website url, https://archive.debian.org/, contains the entire repository of Aptitude installable packages. These packages' names contains the title, version and ISA. By filtering by ISA you can install all available software for IA-64. For example, `binutils_2.22-8+deb7u2_ia64.deb`, for which `binutils` is the package title, `2.22-8+deb7u2` is the detailed version, and `ia64` signifies that it is compiled for IA-64 machine code. With these filters in place for only the most recent versions (to omit repeating the same software), and only for IA-64, only 100 GB of data remains. This data includes a wide range of software, from Blender (3D computer graphics software) to Mozilla Thunderbird (email client).

5.1.2. Disassembling the binaries

The next step in the Static Analysis is to put the data in better representable format, and to filter for the actually useful data. The aforementioned collected binaries do not just contain machine code data, but also constants and initialization data. As these binaries are generated for Linux, they mostly use the ELF (Executable and Linkable Format). This data is organized in data sections. Each denoted with their data attributes, such as read-only, or executable. For the purpose of this static analysis, just the executable sections will be analyzed.

The binaries seem to have a similar structure, which seems to be since Debian packages are generally compiled with GNU compiler `GCC` and use `dpkg-buildflags` by default. Since the same compiler configurations are used for each of these collected `dpkg`s, the sections have a similar structure: Starting with `.init`, `.plt` and `.text`. The first two sections are used or dynamic library loading, whereas `.text` typically contains the actual compiled source code.

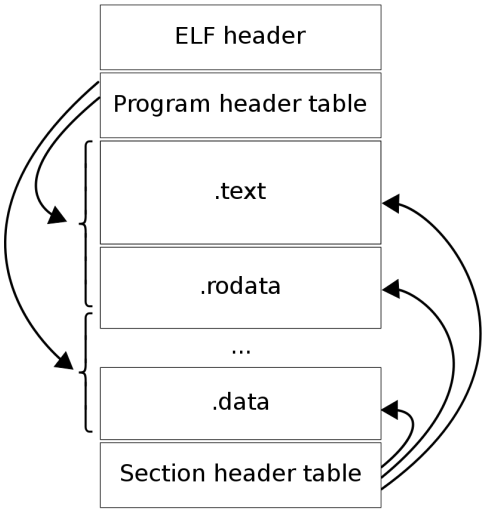


Figure 5.2: Enter Caption

5.1.3. Ethical considerations

Collecting and analyzing these binaries raises ethical implications concerning the website and software owners. They are implicit stakeholders in this study. Downloading this 100GB of (filtered) data may put the website under heavy load. Additionally, there may be ethical and legal implications regarding the analysis of the software, such as reverse engineering software and libraries used by the source code.

To mitigate these potential issues, the data collector (e.g., download script) can be modified to hold a certain rate-limit (e.g., requests per second, bandwidth limitations) to make sure the website is still operational for other users. Secondly, while decompiling or disassembling of software could violate intellectual property laws, this typically concerns purposes such as reproducing, distributing or modifying

the software. However this academic research study only extracts and publicly shares statistics regarding the underlying ISA from a large set of software, such as instruction frequency or bundle scheduling. This leaves the software binaries anonymous and untraceable.

5.2. Analyzing binary sizes

Needless to say for a Very Long Instruction Word (VLIW)-style architecture - the instructions are indeed long. One of the first observations with the compiled binaries is the size of the binary files. However, these binary files also include other sections like data and global offset tables. When only comparing the `.text` sections, we find some interesting data regarding the sizes.

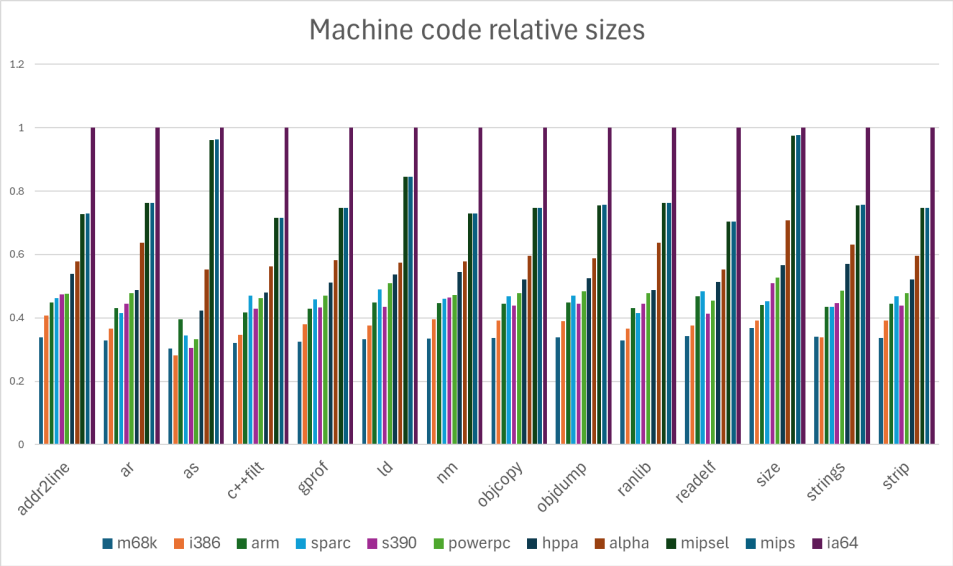


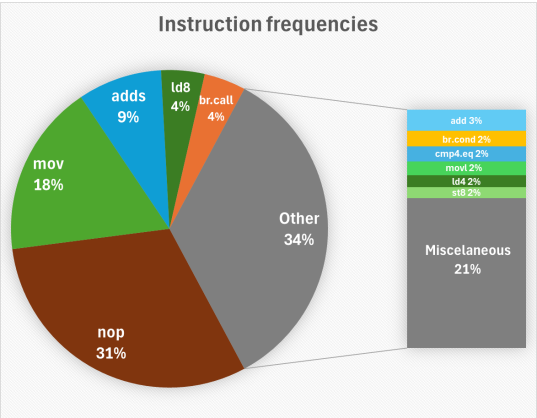
Figure 5.3: Machine code sizes (relative to IA-64)

5.2.1. Bundle NOP overhead

?? The size of IA-64 binaries, often criticized for being disproportionally large, relates to Itanium's most infamous instruction: NOP (No OPeration). Statistical analysis shows that a significant amount of IA-64 instructions comprise of NOPs. As shown in figure 5.4a 31% of instructions comprise of NOPs. These NOPs are inserted in the machine code primarily due to bundle template constraints, with others inserted for instruction address alignment or, less commonly, timing purposes [11].

```
...
09 38 10 1c 10 1c  [MII]       ldfs     f7=[r14],4
00 00 00 02 00 60              nop.i    0x0
46 99 01 84                    adds     r51=20,r51;;
09 30 00 1c 50 18  [MMI]       ldfs.a   f6=[r14]
40 38 bc 20 3b 00              stfs     [r47]=f7,4
00 00 04 00                    nop.i    0x0;;
09 00 00 00 01 00  [MMI]       nop.m    0x0
60 00 38 20 32 00              ldfs.c.clr  f6=[r14]
00 00 04 00                    nop.i    0x0;;
0c 00 00 00 01 00  [MFI]       nop.m    0x0
80 00 18 0c 44 00              fmpy.s.s0  f8=f6,f6
00 00 04 00                    nop.i    0x0
0b 00 18 5e 90 19  [MMI]       stfs     [r47]=f6;;
60 00 3c 20 30 00              ldfs     f6=[r15]
00 00 04 00                    nop.i    0x0;;
...
```

(a) Sample assembly showing bundles with Nops



(b) Instruction frequencies over large compiled binary dataset

Figure 5.4: Sample assembly showing Bundles with Nops

Figure 5.4b shows an example code snippet (Blender:`4000000000c81d70`) where there are 7 NOPs. Figure 5.4a shows the frequency distribution of the most common IA-64 instructions. In IA-64, `mov` is used for almost all register-register movements, including general purpose (GP) register to control register (CR). Unlike x86, `mov` is not used for memory loads and stores. Those are, also as entries in the pie chart, ld and st.

5.2.2. EPIC Instruction sizes

Register encoding overhead

One big factor in the encoding size of IA-64 is the 7-bit register encoding for each of the three operands. As figure 5.5 shows, these encodings always allow the use of all 128 registers. However, `A5` here is an exception, which shows how a register encoding can be foreshortened, only exposing 4 possible registers for r_3.

One possible way to optimize the binary sizes of IA-64 is to look into reducing this register encoding size. Figure 5.6 shows that one register, being r14, is used 18% of the time. Registers r14 and r1 together make up 27%, which means just two registers make up a quarter of all register accesses. furthermore, table 5.1 shows that only 32 of the 128 registers make up 90% of register accesses, and that the top 16 registers make up 75%. According to the statistics, Registers r0-r63 are accessed 97% of the time. Which means that if this 'highest bit' were ignored, up to 9 bits (3 operands, 3 instructions per bundle) could be freed up to encode more useful instructions. Or perhaps allow the 41-bit instructions to be slightly shortened and allowing more bits for the template encoding, which in turn could eliminate a bunch of unnecessary NOPs from these bundle restrictions.

This optimization itself may not be though-out or useful, but the idea that there are still many opportunities for compacter code does solidify the idea that IA-64 was not designed with code binary size as a high priority.

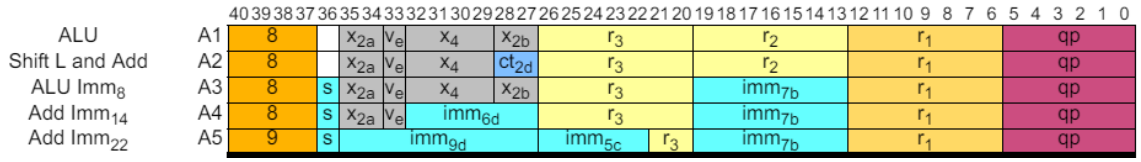


Figure 5.5: Encoding of simple IA-64 instructions

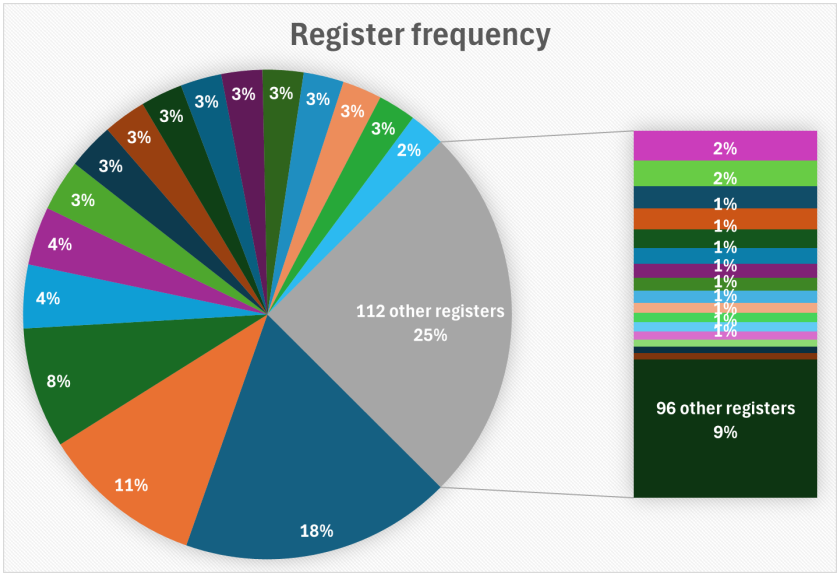


Figure 5.6: Register frequency pie chart

Table 5.1: Top Cumulative Data (in Percentages)

Top	Cumulative (%)
Top-8 registers	54.00%
Top-16 registers	75.08%
Top-32 registers	90.62%
Top-64 registers	97.35%

Predication encoding overhead
Similarly to section 5.2.2, the predicate registers have even more space for improvement in terms of optimizing the use of encoding bits. Only two out of the 64 predicate registers are used 90% of the time, p06 and p07. The top 12 out of the 64 are used 98% of the time. This indicates that the amount of predicate registers could greatly have been reduced to e.g. 16. Alternatively, that only 16 are encoded for all instructions in 4 bits instead of 6, and only some instructions (like `mov`/`br.cond`/`add`) would be able to use all 64. This optimisation would free up 2 bits per instruction, or 6 bits per bundle, again for space to encode better bundle sets or more useful opcode space.

5.3. Bundle groupings and theoretical upper bounds performance

The main challenge in static scheduling arises from the unpredictability of runtime events, such as:

- Cache accesses;
- Complex operations like mul/div, complex FP, SIMD, and transcendental instructions;
- Branch prediction;
- Synchronization and atomic instructions;
- OS or other threads interfering with caches or task switching.

In comparison, OoO Superscalar architectures handle these tasks dynamically, often with better efficiency but at the cost of increased hardware complexity. This difference in delays across systems adds complexity in making portable code across IA-64 platforms, as optimal scheduling heavily depends on the latencies of instructions.

5.3.1. Instruction level parallelism

[**subsection:ilp**] The large dataset of compiled binaries allows for a quite trivial yet principal study to calculate the average Instruction Group size. As a reminder, instruction bundles are fixed size, having 3 slots for instructions. Instruction groups, on the other hand, consists of any number of instructions between explicit dependency boundaries. These groups are generally indicative of ILP, or at least an upper bound of parallelism. The effective parallelism may be lower due to instruction execution latencies, unexpected stalls (like cache miss, branch mispredict) or feed-forward stalls. While LX instructions take up two instruction slots, as shown in figure 5.7, they are counted together as one long instruction.

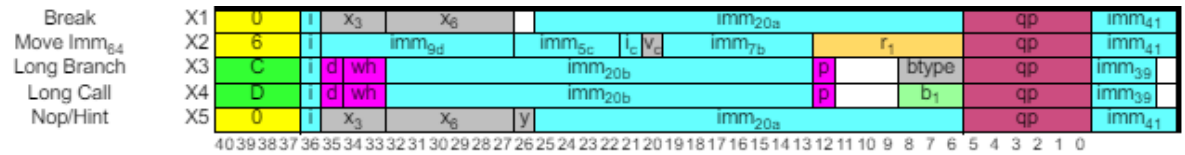


Figure 5.7: LX instruction encodings

The average instruction level parallelism (ILP), is roughly the average instruction group size, can be calculated as the following:

$$0 * f_0 + 1 * f_1 + 2 * f_2 + ... + 7 * f_7$$

where f_n is the relative frequency for n-way parallel.

$$= 0 * 0.02 + 1 * 0.33 + 2 * 0.31 + ... + 7 * 0.01$$

results in

$$= 2.26$$

This calculated value for the average program group size does not directly imply the IPC at runtime of the runtime, but are indicative.

5.4. Bundle sets optimization

Since **??** concluded that Itanium's current encoding leads to such a high amount of NOPs (30%) due to bundling constraints, leads to the next question: Is it possible to improve IA-64's bundle set, so that the optimal bundle schedule has fewer NOPs in a wide range of compiled software?

If the encoding of IA-64 could have been redesigned to omit a significant amount of NOPs, it would improve instruction density and performance. Significantly better instruction density could cause better cache utilization and bandwidth, allowing for smaller caches for more hardware estate for other more useful units, smaller chip sizes overall and lower energy consumption. In terms of performance, it could lead to lower cache utilization, allowing for more bandwidth for data I/O and lower amount cache misses, or lower amount of bundle splitting in the bundle issue unit [12] due to pointless NOPs.

5.4.1. Methodology

Given the previous calculations that the average instruction group size in Itanium machine code is roughly 2.3, that already means that the current schedule which mostly consists of 3-instruction boundaries, will lead to an average of occupancy of $1 - \frac{2.3}{3} = 23.333\%$ NOPs already.

Table 5.2: Effective ILP (Instruction Group Size) in IA-64 Binaries

Instruction Group Size	Percentage (%)
1	33%
2	31%
3	18%
4	8%
5	4%
6	3%
7+	1%
0	2%

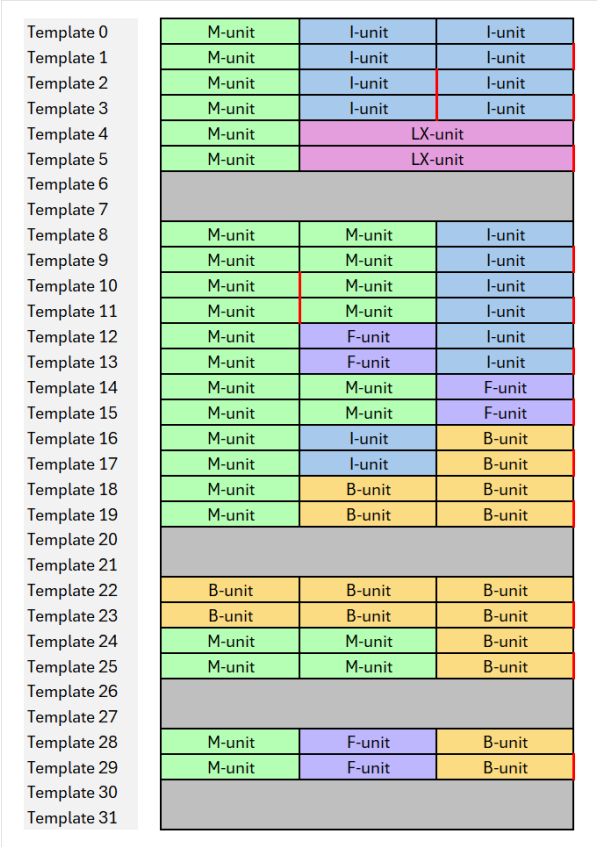


Figure 5.8: IA-64 bundle set

To design a better bundle set, which could get scheduled in to better programs, there are a few constraints to ensure correct execution. Some of the key restrictions are:

- LX must be in the last two slots;
- Any bundle must contain at most two barriers;
- Schedule must be “legal”, meaning any combination of instructions should be possible (with enough NOPs). Some Bundle sets lead to situations where it is impossible to schedule (with a trivial example being a bundle set that does not contain any F-units, makes it impossible to schedule F-instructions);
- Must contain at least one bundle with B as last entry (which is a restriction on br.cloop, br.ctop, br.cexit, br.wtop and br.wexit instructions);
- Must contain at least one bundle with I as first entry (which is a restriction on alloc and flushrs, which must be the first instruction of an instruction group and thus bundle);
- Set must contain only 24 bundle elements (for the same forward compatibility as the original);
- Should contain no duplicate bundles in the same set (otherwise it would not be optimal);
- Should not schedule more than two M, I or F in the same instruction group within the bundle, as that is trivially not possible to schedule in one cycle (Itanium 1 only had 2 M, I and F units).

With these constraints, there are still about `10^80` different combinations of bundle sets. So it is quite a challenge to prove the new schedule is strictly the best. However, just finding one example of a better schedule already can show that EPIC still has potential. If this original really is the optimal schedule, then the NOP rate cannot be lowered from 30% from optimal scheduling alone. Meaning EPIC has no better scheduling.

Template 0	I-unit	I-unit	I-unit
Template 1	I-unit	I-unit	I-unit
Template 2	I-unit	M-unit	I-unit
Template 3	I-unit	M-unit	I-unit
Template 4	M-unit	I-unit	I-unit
Template 5	M-unit	I-unit	M-unit
Template 6	M-unit	M-unit	M-unit
Template 7	M-unit	M-unit	M-unit
Template 8	B-unit	M-unit	M-unit
Template 9	M-unit	B-unit	B-unit
Template 10	B-unit	B-unit	I-unit
Template 11	B-unit	I-unit	I-unit
Template 12	B-unit	M-unit	M-unit
Template 13	I-unit	I-unit	F-unit
Template 14	B-unit	M-unit	F-unit
Template 15	B-unit	M-unit	F-unit
Template 16	I-unit	F-unit	M-unit
Template 17	I-unit	F-unit	M-unit
Template 18	F-unit	B-unit	F-unit
Template 19	F-unit	B-unit	F-unit
Template 20	F-unit	B-unit	F-unit
Template 21	F-unit	M-unit	F-unit
Template 22	B-unit	LX-unit	
Template 23	M-unit	LX-unit	
Template 24			
Template 25			
Template 26			
Template 27			
Template 28			
Template 29			
Template 30			
Template 31			

Figure 5.9: The new bundle schedule

5.4.2. Results

This schedule results in 20% NOPs. While this is an improvement over the current IA-64 encoding, it shows that there are still limitations to EPIC, or at least this 5-bit template constraint on their bundling.

5.5. Conclusion

The different studies which were based on the static analysis of IA-64 machine code shows some interesting characteristics of Itanium's runtime. Each of the studies, analyzing binary sizes, NOP overhead calculations and ILP upper bounds have their corresponding research questions:

R3.3. How do IA-64's program sizes compare with competitors?
R3.4. How significant is the proportion of NOPs in typical IA-64 binaries to performance?
R3.5. How much Instruction Level Parallelism is expressed in IA-64 compiled binaries?

How do IA-64's program sizes compare with competitors?

IA-64 compiled binaries are significantly larger than most other architectures, aside of MIPS. Given the same compiler and compiler flags, the different ISAs seem to follow a similar size distibution across multiple different files. i386 (also known as x86-32 or IA-32) and m68k have the smallest binaries with a few RISC models following up. Dec Alpha, MIPS and IA-64 have proportionally larger binaries. With IA-64 having a on average 2.8x larger binaries than m68k. This is attributed to the amount of NOPs and inefficient instruction encodings.

How significant is the proportion of NOPs in typical IA-64 binaries to performance?

Around 30% of instructions are NOPs in IA-64 binaries. This could also have an impact on performance, due to instruction cache entries and memory bandwidth being used with unproductive NOPs. Instruction groups that could have executed in one cycle may also take additional cycle to execute due to cache line boundaries causing a split issue, which delays the pipeline of the split off instruction groups [12].

How much Instruction Level Parallelism is expressed in IA-64 compiled binaries?
Section 5.3 concluded that average instruction group sizes are about 2.3 in IA-64 binaries. As depicted in figure 4.5a, Itanium has nine functional units. This implies that roughly $2.3/9 = 25.5\%$ of the CPU is effectively in use on average as upper bound. Actual performance depends on latencies of instructions and stalls.

6

Comparative Analysis of Hardware Complexity

This chapter delves into the hardware complexities that has likely contributed to the delayed development of the Itanium processor. The hypothesis that hardware complexities was the main cause of the delays is supported by multiple sources. The hardware complexity can be estimated using CPU floor plan analysis.

One noted benefit of Itanium's architecture is that its Explicitly Parallel Instruction Computing (EPIC) paradigm can manage out-of-order (OoO) execution in software, omitting the need for hardware mechanisms. This should, in theory, simplify the hardware. But that conflicts with the original hypothesis of Itanium being delayed due to hardware issues. To first tackle this contradiction, this chapter first aims to answer the question:

> **R3.6.** Do EPIC architectures reduce hardware complexity in CPU floor plans?

6.1. Methodology

As a third-party analysis, this study ventures into speculative domains. A large community of collaborators have offered valuable insights in reverse-engineering the floor plans of the hardware designs. The full credibility in this chapter may not be a guarantee, however to answer the question there seem to be at least some useful conclusions to derive from these approximations.

A significant community of enthusiasts who share and discuss images of delidded chips pose a good basis of this study: Detailed die shots of the logic lay-outs of the hardware. From here, the floor plans of the CPU's can be derived. Floor plans indicate the area, and in some cases may shed light on the complexities of certain micro-architectural units.

The analysis of the floor plans consists of three main parts: Selecting CPUs and corresponding die shots for comparison, collecting data to complete the floor plans and finally, analyzing these (approximated) floor plans for anomalies and complexity comparisons. The analysis will have an emphasis on comparing the hardware complexity of the Pentium with Itanium. The key differences between these two being Pentium featuring OoO circuitry, whereas Itanium lacks of OoO hardware due to its EPIC architecture.

6.2. Selecting CPU architectures to compare

The CPUs selected for detailed comparison are chosen on the basis of many aspects:

- Die shot quality. This is quantitatively measured in image quality (image pixel dimensions) of effective area;

- Existing floor plans, both speculation and firsthand data from Intel. This is qualitatively compared, as only few architectures have detailed floor plan annotations. Only a limited few have firsthand credibility;
- Comparable CPU's in terms of time of release, target segment, performance.

Based on these criteria, the following CPU's were filtered:

Table 6.1: Specifications of Various CPUs [25]

CPU Microarchitecture	Node	Cache (I$+D$+L2$)	Die Size (mm^2)	Transistors (millions)	Release Date
Intel P5	800 nm	8kB+8kB	293.92	3.1	1993
Intel P54C	600 nm	8kB+8kB	148	3.3	1994
Intel P6	250 nm	16kB+16kB	306	5.5	1999 Q1
Intel Klamath P6	350 nm	16kB+16kB	203	7.5	1997 Q1
Intel Deshutes P6	250 nm	16kB+16kB	131	7.5	1998 Q1
Intel Katmai P6	250 nm	16kB+16kB	127	9.5	1999 Q1
Intel Coppermine P6	180 nm	16kB+16kB+256kB	106	28	1999 Q4
Intel Tualatin P6	130 nm	16kB+16kB+512kB	80	44	2001 Q2
Intel Merced	180 nm	16kB+16kB+4MB	330	295	2001 Q2
Intel McKinley	180 nm	16kB+16kB+3MB	421	221	2002 Q3
Intel Madison 9M	130 nm	16kB+16kB+9MB	374	592	2003 Q2
AMD K5	500 nm	8kB+16kB	250	4.3	1996 Q1
AMD K5 PR100	350 nm	8kB+16kB	161	4.3	1996 Q1
AMD K6	300 nm	32kB+32kB	250	8.8	1997 Q1
AMD K6-2	250 nm	32kB+32kB	81	9.3	1998 Q1

The entries in table 6.1 were filled in with x86-guide, a community-driven CPU specification archive [25]. Most of the entries had easily accessible die shots online, and seemed to all be from around the same time frame.

6.2.1. Collecting data for the floor plans

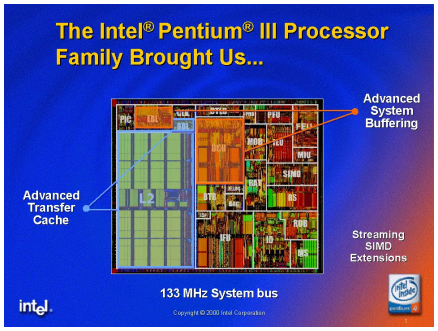
After defining which CPUs to consider, the next step is to explore the available data for each model. The results of researching these models and their corresponding floor plans and diagrams are filled in intable 6.2 below.

Below are some examples of these floor plan annotations:

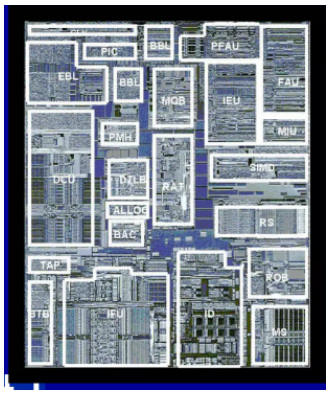
Table 6.2: Availability of High-Resolution Images and Annotations for Various CPUs

Name	Difficulty to Find High-Res Images	Annotated	Block Diagram
Intel P5 (Pentium)	Hard*	None	Yes
Intel P54C (Pentium)	Easy	Fully	Yes
Intel P6 (Pentium Pro)	Easy	None	Yes
Intel Klamath P6 (Pentium II)	Easy	Mostly	Yes
Intel Deshutes P6 (Pentium II)	Easy	None	Yes
Intel Katmai P6 (Pentium III)	Easy	Fully	Yes
Intel Coppermine P6 (Pentium III)	Easy	None	Yes
Intel Tualatin P6 (Pentium III)	Easy	None	Yes
Intel Itanium (Merced)	Easy	Partially	Yes (WikiChip)
Intel Itanium 2 (McKinley)	Easy	Partially	Yes
Intel Itanium 2 (Madison 9M)	Easy	Fully	Yes
AMD K5	Hard*	None	Yes (Article)
AMD K5 PR100	Hard*	None	Yes (Article)
AMD K6 (Little foot)	Easy	None	Yes (WikiChip)
AMD K6-2 (Chompers)	Easy	None	Yes (Wikipedia)

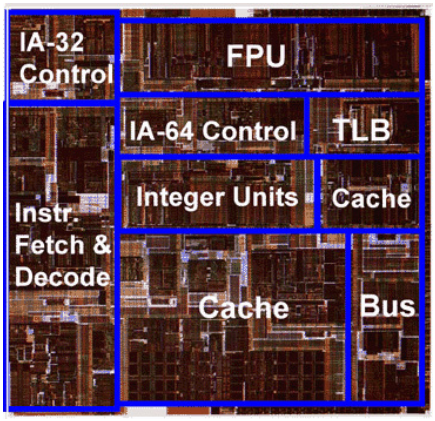
*Note: Hard means there were no images available in popular databases, but does not necessarily imply they do not exist online



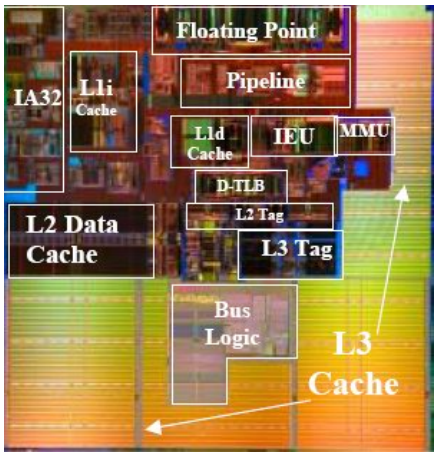
(a) An example of firsthand floor plan details from Intel for the Coppermine P6 architecture



(b) An example of inferred floor plan details based on speculation for the Katmai P6 architecture



(c) Speculated floor plan details of Intel Merced



(d) Speculated floor plan details of Intel Madison (same layout as Intel Madison 9M but with less cache)

Figure 6.1: Examples of floor plan depicting the varying depth and credibility

Figure 6.1 depicts four different floor plans and the variability in floor plan credibility and extensiveness. Figure 6.1a shows an low quality image, but high credibility as a firsthand source from an Intel press release. Next to it, figure 6.1b shows a highly detailed floor plan, with low quality image and unknown credibility. Figure 6.1c depicts an often cited floor plan of Merced. The image has few floor plan annotations, making it a challenge to deduce the actual functional units. Finally, figure 6.1d depicts a good quality floor plan, again with unknown credibility. The annotations do seem to concur with the annotations in figure 6.1c.

6.3. Results

With all the results (block diagrams, specifications and high-definition base images) collected, all this data can be put together into one floor plan image.

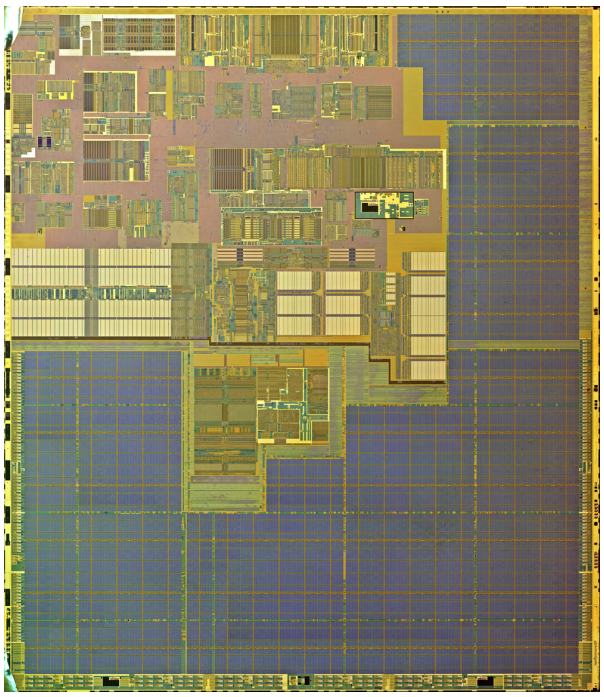
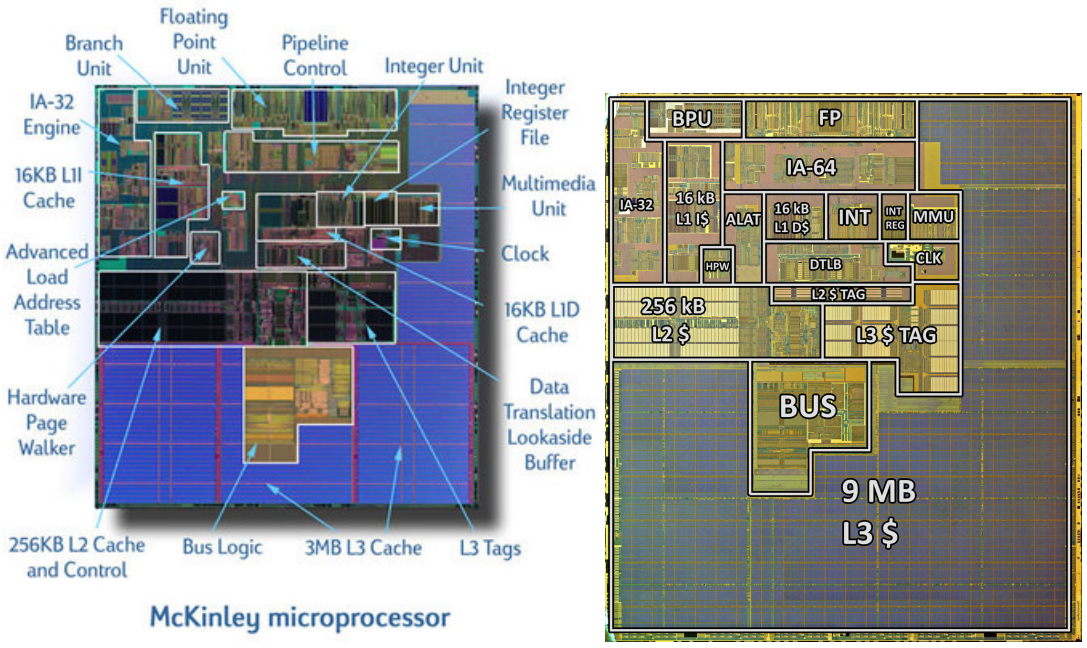


Figure 6.2: Clean start for Madison 9M with no annotations

6.3.1. Inferred floor plans from data

Starting with a base image, we first realized that the core seemed pretty much identical to the McKinley micro-architecture. Upon closer inspection, the two architectures seem close to identical, except for the L3 size and L3 tags, which concurs with the specifications found about the two architectures. Using a partially filled out floor plan from McKinley in figure 6.3 seems to align well with the clean Madison in figure 6.2.



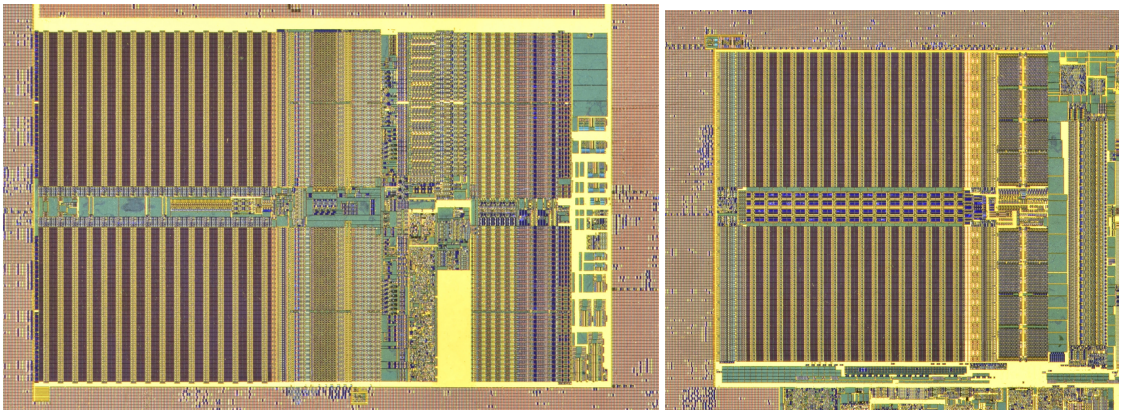
(a) Speculated McKinley floor plan

(b) Madison floor plan filled in with McKinley data

Figure 6.3: Similar looking structures, hypothesized to be L1 Instruction and Data caches

Given this start, the start of the Madison floor plan can be filled in, as shown in figure 6.3.

Looking closer at the details, the L1 instruction and data caches can be found exactly in the high definition image where it was expected according to the McKinley floor plan labels. The structures figure 6.4a and figure 6.4b look quite similar in size and complexity, which strenghtens the hypothesis that these are L1 caches.



(a) The identified L1 Instruction cache

(b) The identified L1 data cache

Figure 6.4: Similar looking structures, hypothesized to be L1 Instruction and Data caches

In trying to uncover the floating point (FP) unit, this structure marked in red in figure 6.5 was found to be similar as the red marked structure in figure 6.5. The latter is known to be the register file. When doing pixel measurements, the FP register file was 1,556 px tall, and the int register file 1,216 px. According to figure 6.7, the floating point register file should be 82 bits wide, and the integer 64. If we do $1556/1216 * 64$, we get 81.9, which is very close to the expected 82 bits of the floating point register file. This strengthens the theory that these annotations are correct, and that these are indeed the respective register files. The two units at either side of the FP register file might be the FP units, which - again, according to figure 6.7 - has two data paths for its floating point.

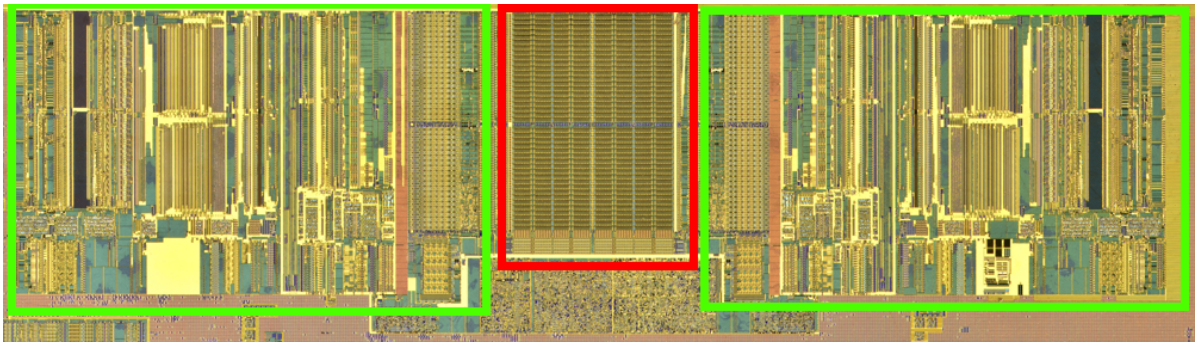


Figure 6.5: Hypothesized fp register file

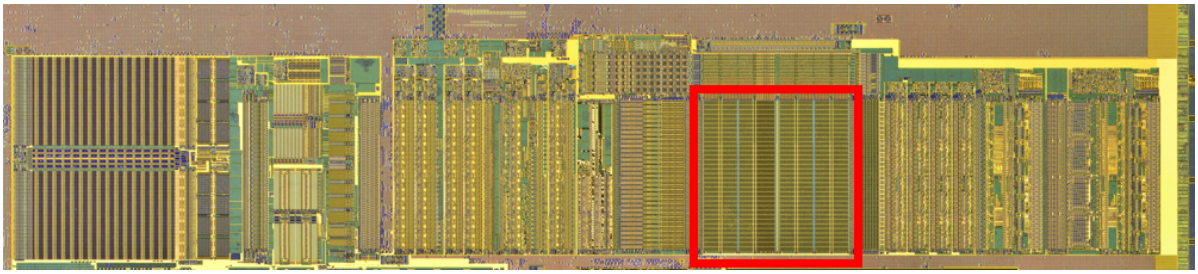


Figure 6.6: Hypothesized integer register file

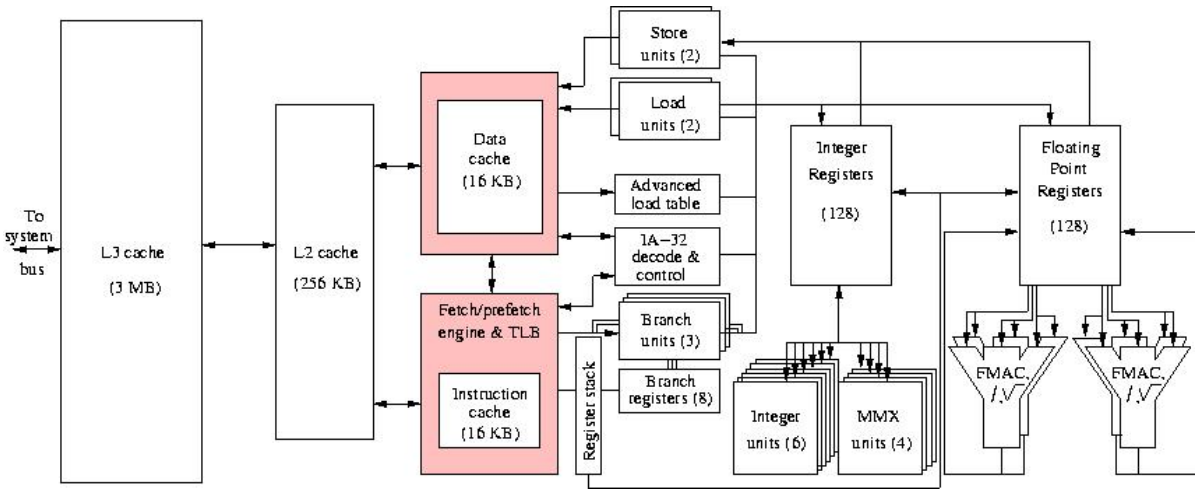


Figure 6.7: Madison Block diagram

With just one more unit not certain of its purpose, deduction from other floor plans concluded that this unit in red question marks in **??** should be the decoder. Considering its placement between the L1 I$ and the IA-64 pipeline, it would make sense.

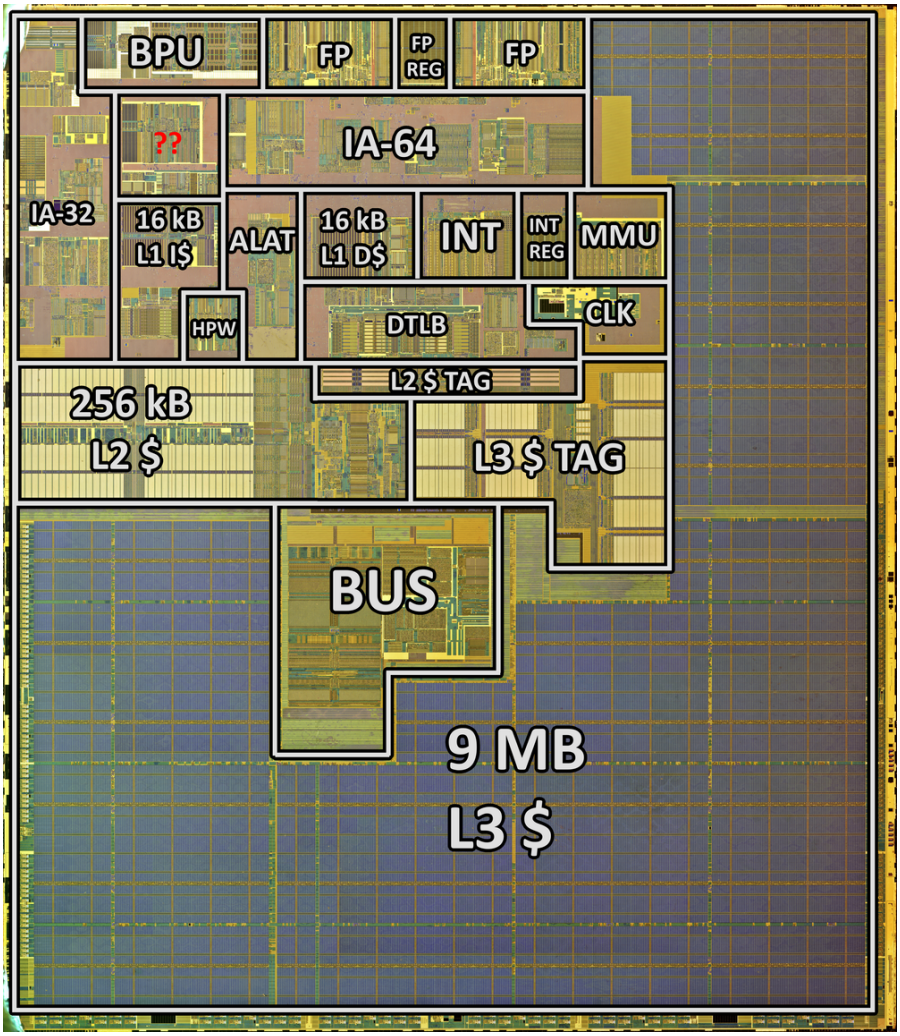
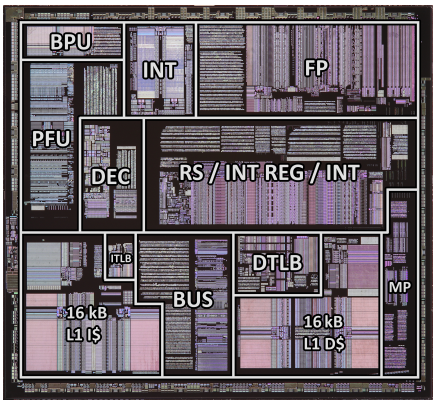


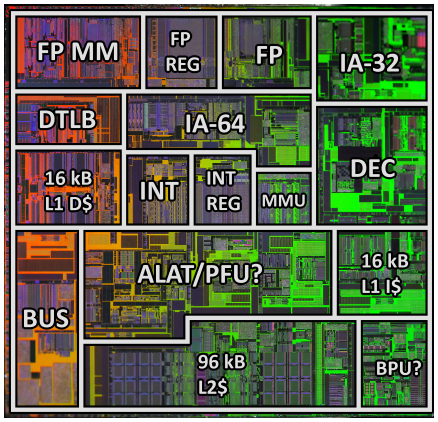
Figure 6.8: Madison core with deduced FP and caches with one unknown area

6.3.2. Out of Order vs EPIC core complexity

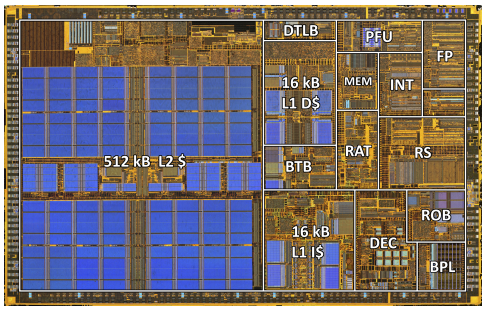
Inferred from the data collected from existing floor plans, block diagrams and other available data sheets, the following floor plan approximations were made:



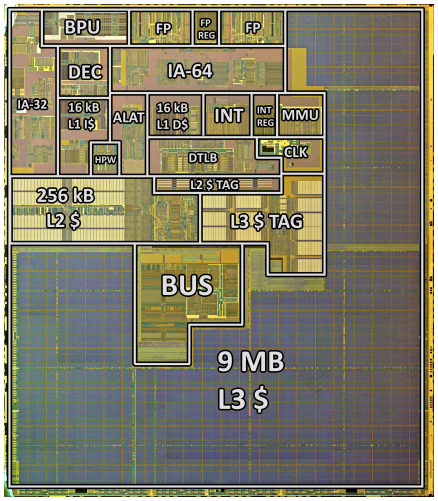
(a) Intel P54C (Branded as Pentium) die shot with floor plan annotations



(b) Intel Merced (Branded as Itanium) die shot with floor plan annotations



(c) Intel P6 (Branded as Pentium Pro and Pentium II) die shot with floor plan annotations



(d) Intel Madison 9M (Branded as Itanium 2) die shot with floor plan annotations

Figure 6.9: Collection of example images with floorplans

From figure 6.9, it would seem that despite architectural changes, they all seem to have roughly the same relative hardware floorplan estate for ILP in hardware. In Itanium, the ILP is mostly in hazard detection and scoreboarding, which is denoted by IA-64 logic blocks. In Pentium x86, the ILP sections is denoted as RS, and in P6, the Out of Order (OoO) engine also encludes Re-order buffer (ROB) and Register alias table (RAT).

Originally, Itanium was set out to exploit ILP in software to omit the need for hardware complexity in ILP extraction. However, in the comparisons, it would seem that the ILP-related circuitry, such as scoreboarding and hazard detection in Itanium is still similar to Pentium's reservation stations and hazard detection.

Interestingly, Itanium's relative size for its Integer units is similar to Pentium, despite it being double or thrice the size (4 execution units on original Itanium, 6 on Itanium 2) and the 64-bit width. This is likely due to the less complex instructions, such as lack of integer Multiply and Divide in these units, with these only implemented in the FPU's.

Both Itanium and Pentium have 2x64 or 4x32 parallel floating point units. Itanium as two separately piped 64-bit (1x64 and 2x32) SIMD, and Pentium as a single 128-bit (2x64 and 4x32) SIMD in its SSE extension, which explains why the floating point units have similar relative sizes and performance.

Another factor to keep in mind is the IA-32 decoding circuitry in both Itanium systems, which are the hardware accelerated x86 execution circuitry. It takes a significant amount of surface area, likely due

to x86's complexity. This area is ignored in the comparison, as it only functions as a compatability layer, and not as contributing towards IA-64 hardware complexity.

6.3.3. Conclusion

While the relative size of these units does not exactly convey the complexity of the hardware, it does indicate that IA-64's ILP management circuitry is still similar in complexity as Pentium, which was expected to be significantly smaller.

6.3.4. Transistor counts of Itanium Cores

Using the specifications from table 6.1, it is also deducible that Intel's L3/L2 cells consist of 8 transistors using some basic reasoning:

$$M + C \times X = T$$

where:

- M represents the microarchitecture transistor count without L2 cache,
- C is the L3+L3 tags+L2 Cache size in bits,
- X is the Cache SRAM transistor count per bit,
- T is the Total transistor count including L3 cache.

For two models with a similar M (e.g. same microarchitecture), one can fill in C and T accordingly and deduce X, the amount of transistors per L2 Cache SRAM cell.

$$M + C_1 \times X = T_1$$

$$M + C_2 \times X = T_2$$

Which can be rewritten as

$$C_1 \times X - T_1 = C_2 \times X - T_2$$

Since we know M is equal in both cases. Then we extract X:

$$X = \frac{T_2 - T_1}{C_2 - C_1}$$

Now, since we have data for multiple Intel models with similar cache sizes:

- Intel Coppermine P6 and Intel Tualatin P6 with with 256 kB (=2097152 bits), 28 million transistors and 512 kB (=4194304 bits), 44 million transistors respectively.
- Intel McKinley and Intel Madison 9M with 3 MB + 64 kB + 256 kB (=26,560,000 bits), 221 million transistors and 9 MB + 192 kB + 256 kB (=75,584,000 bits), 592 million transistors respectively.

Note: L3 tag size is $9MB/128Blines = 73728$ *entries,*

Filling in these values, we get

$$X = \frac{592e6 - 221e6}{75584000 - 26560000}$$

$$X = \frac{44e6 - 28e6}{4194304 - 2097152}$$

Which equates to $X \approx 7.56$ or $X \approx 7.63$, which we will assume to be about 7.5. This leaves just one more unknown, the M, which we can derive from the same

$$M + C \times X = T$$

By rewriting as

$$M = T - C \times X$$

Which indeed makes sense, that we are calculating the total amount of transistors excluding the L2 Cache transistor count approximation. Using the formula we get:

For P6 (Coppermine and Tualatin)

$$M = 28e6 - 2097152 \times 7.5$$

$$M = 44e6 - 4194304 \times 7.5$$

$M \approx$ 12 million transistors

For Itanium 2 (McKinley/Madison 9M):

$$M = 221e6 - 26560000 \times 7.5$$

$$M = 592e6 - 75584000 \times 7.5$$

$M \approx$ 24 million transistors

For Itanium (Merced):

$$M = 295e6 - 33554432 \times 7.5$$

$M \approx$ 43 million transistors

So it seems that Itanium microarchitectures (chip - L3 - L3 tags - L2 cache approximation) were significantly larger than P6 chips. While this could be attributed to Itanium's chips being newer and bigger, but newer Intel chips in the Pentium 4 line from 2000 gives similar results: Intel Willamentte 42000000-256 *1024*8*7.5 = 26 million transistors Additionally, it can hardly be attributed to Itanium being 64-bit vs x86's 32-bit, as the x86-line Pentium's all included 128-bit SSE instructions. x86 is famous for its complexity, and that also extends to its hardware such as complex align/decode and complex x87 support as examples.

6.4. Conclusion

As what often happens in an exploratory study, when aiming to answer the question of hardware issues relating to development delays, there did not seem to be any clear data showing the one way or the other. However, what did result from this study was that the overall hardware complexity of Itanium seemed comparable with the x86 lines in terms of functional units and their sizes. Both approaches, being CISC and EPIC, seem to result in quite similar structures and complexities in terms of hardware.

7

Differentiation Strategy Framework Analysis

As the final study of this thesis, the remaining two subquestion is tackled:

> **R3.9**. Does the Differentiation Strategy framework shape Itanium's market placement?

Using the differentiation strategy framework, this chapter aims to answer the key question to shed light on the main question of "Why did Itanium fail?"

7.1. Background on Differentiation Business Strategy

Differentiation business strategy involves making a product or service stand out from its competitors in the market. The framework describes how differentiating products or services from competitors, it increases your competitive advantage in the market [22].

This framework is relevant to this case study about Itanium, as differentiation is critical given the competitive landscape of the processor market. Understanding why and how Intel attempted to differentiate Itanium provides insights into its strategic decisions and market performance.

An alternative strategy could include Blue Ocean. The blue ocean strategy involves attempting to create a novel and uncontested market space. If successful, this strategy hypothesizes that this new market is completely dominated by the creator by default, either remaining dominant or at least having an edge other competitors before they get the chance to step in. While some aspects of the Itanium case study seem to line up with the Blue Ocean strategy, one key factor make the entire strategy inapplicable: The market was not created by Intel or HP. The high-performance workstation and server segment already existed, and Itanium's purpose was to compete in this existing demand, against other companies like AMD and IBM. Additionally, Intel remained a competitor in this segment with its traditional CISC CPU's, which further shows that this market was not untapped potential.

7.2. Methodology

The methodology for this analysis involves:

1. Identifying unique value propositions of Itanium.
2. Analyzing the target market for Itanium.
3. Assessing the competitive response from other market players.
4. Evaluate the effectiveness of the differentiation strategy.

7.2.1. Identify Unique Value Propositions

As discussed in the Sub section 2.6.1, Itanium was set out to have certain unique value propositions:

- 64-bit for scalability (allow more memory, high performance with large numbers);
- Higher performance than traditional CPU's leveraging EPIC;
- Specific features tailored at the High-performance and data center segment.

According to Intel's marketing, Itanium's instruction set, called IA-64 was clearly stipulated to the public that that was going to be the 64-bit architecture, and that IA-32 (x86-32) would remain 32-bit. All the benefits that come with 64-bit computing was practically reserved for IA-64 at that time.

Due to a certain fear that existed that OoO execution would not scale well [23] in hardware, the next best possible way to still achieve better performance is to exploit ILP in software and simplifying the hardware. At the time of development this approach proved quite promising in lab conditions.

Specific features tailored at high-performance and data center entails Reliability, Availability and Serviceability (RAS), security, multi-processor scalability and error correction.

7.2.2. Target Market Analysis

The target market for Itanium included enterprise servers, high-performance computing, and business-critical applications, sectors that demand high reliability, scalability, and processing power. This is exactly what Itanium was designed for, so there is no surprise that Itanium checks all the boxes in this aspect. As concluded in Chapter 4: Performance Comparative Analysis, the performance was comparable to x86 in general workloads, but in floating point workloads it seemed to outperform competitors. Given Itanium's high prices, but marginally better performance, the cost / performance ratio may have been unbalanced slightly. As for the specific features, such as security, reliability and scalability, Itanium included all of these features already before it was released. At the time, virtualization features did not exist yet. The long-term support seemed positive at Itanium's release.

Table 7.1: Step 1: Alignment of Itanium Features with Server/Enterprise Needs

Server/Enterprise Needs 2001	Itanium's Applicability
64-bit for scalability (memory, large numbers)	Great
Performance	Good
Compatibility with Existing Software	Terrible
Cost-Effectiveness	Medium
Advanced Security Features	Great
High Reliability	Great
Multi-processor Scalability	Great
Support for Virtualization	Poor
Energy Efficiency	Moderate
Long-term Support	Good

Table 7.1 indicates Itanium's initially overall great applicability for this segment. Out of the 10 measures, it seems to have met 7 of them with a "Good" or higher. The only three main issues are compatibility, cost-effectiveness and (relative) performance. Intel expected performance to improve over generations of Itanium. As soon as performance needs were met, then they would scale down towards the workstation and desktop market, which has a higher need for cost-effectiveness, forcing Intel to price their products competitively.

7.2.3. Assess Competitive Response

However, since Itanium does not exist in a vacuum, the differentiation framework necessitates a comparison with the competition. One of such competition, is Intel's own Pentium line:

With Pentium added to table 7.2, it contextualizes Itanium a bit in terms of its cost effectiveness and performance. Pentium did not have 64-bit support, but did have great performance due to its OoO execution. Pentium did not have any specific data center features, such as RAS or scalability in mind, however since Pentium processorss were already being used in those segments, those features might have already developed at least a moderate amount.

Table 7.2: Step 2: Competitive alignment between Itanium and Pentium for Server/Enterprise Needs

Server/Enterprise Needs 2001	Itanium's Applicability	Pentium's Applicability
64-bit for scalability (memory, large numbers)	Great	Poor
Performance	Good	Great
Compatibility with Existing Software	Terrible	Great
Cost-Effectiveness	Poor	Great
Advanced Security Features	Great	Poor
High Reliability	Great	Moderate
Multi-processor Scalability	Great	Moderate
Support for Virtualization	Poor	Poor
Energy Efficiency	Moderate	Good
Long-term Support	Good	Good

At that point in time, 2001, Itanium still holds an edge over Pentium applicability. With better features tailored at data centers such as support for multi-processor scalability, RAS reliability features and above all, 64-bit support it still holds a qualitative edge over Pentium, while Pentium holds a quantitative edge over Itanium in performance and energy efficiency for Server/Enterprise workloads.

However, AMD releases its 64-bit extension for x86 in 2003: AMD64. Quickly after, Intel follows with EM64T in 2004. Already at this point it is visible what is happening. Slowly, but surely, all edges that

Table 7.3: Step 3: Alignment of Itanium, Pentium, and AMD64 Opteron Features with Server/Enterprise Needs

Server/Enterprise Needs 2004	Itanium	Pentium	AMD64 Opteron
64-bit for scalability (memory, large numbers)	Great	Great	Great
Performance	Moderate	Great	Good
Compatibility with Existing Software	Terrible	Great	Great
Cost-Effectiveness	Poor	Great	Great
Advanced Security Features	Great	Moderate	Good
High Reliability	Great	Great	Good
Multi-processor Scalability	Great	Moderate	Good
Support for Virtualization	Great	Great	Great
Energy Efficiency	Moderate	Good	Good
Long-term Support	Moderate	Great	Great

Itanium had over x86 are slowly being implemented in x86 in the form of extensions.

7.2.4. Evaluate the Effectiveness of the Differentiation Strategy

At that point in time, 2004, Itanium has no more unique value propositions over x86, but x86 does have a big one over Itanium, namely compatibility with existing software, and long-term support. This study show how this framework can indicate the reason for lack of market success of Itanium.

7.3. Conclusion

This study has applied the differentiation strategy framework to analyze the commercial trajectory of Intel's Itanium processor. Through a detailed comparison with its contemporaries, particularly AMD's Opteron (AMD64) and Intel's own Pentium series, it becomes evident how Itanium's value propositions diminished over time.

To reiterate the original questions of this study: "Why did such a large business investment still lead to such a niche product?" and "How can you predict and prevent such missteps in advance?" This study seems to reveal that simply the choice in competitive edge, being 64-bit and qualitative features, did not have a long-standing triumph over a well-established and highly flexible CPU architecture that can easily absorb these qualitative features into its architecture. Itanium remained to have a competitive edge in niche workloads, such as number crunching and scientific calculations, where the code is

predictable and parallel, and can be specifically tailored to its needs. There it has a value proposition in terms of performance. However in the case of mainstream, traditional RISC and CISC systems seem to have a considerable edge due to the dynamic nature of mainstream code and existing code infrastructure. As referred to in Chapter 4.

8

Conclusions and recommendations

This section aims to present a final answer to the main question. To answer the main question: ”Why did Itanium fail?” First the sub-questions are answered based on the conclusions of the different studies. To keep this section self-contained, it starts off by summarizing the different studies.

8.1. Summary

The thesis is structured in three different study categories: Market analysis, Technical analysis and business strategy analysis.

The market analysis consisted of an automated sentiment analysis. This analysis was done by first collecting thousands of articles and papers, and scoring their sentiment towards Itanium using a Large Language Model. The results of this section was that the most frequently mentioned sentiment labels for Itanium was Performance, Software support, Price and Development delays, From this study it seemed that especially performance was mentioned a lot, most notably, both negative and as positive feedback. This may have alluded to some audience targeting issues.

The technical analysis consisted of two parts: The performance comparative analysis and the Compiled binary Static analysis of IA-64. The performance comparative analysis compared the Itanium chips with existing architectures according to the SPEC benchmarks. It concluded that Itanium only fell slightly behind x86, having some issues with fully expressing OoO techniques in software which OoO Superscalar is able to make full use of in hardware.

The compiled binary static analysis of IA-64 showed some flaws in Itanium’s encoding, such as the amount of NOPs, which may cause some memory (cache, bandwidth, issue splitting). The theoretical upper limit of 2.3 Instruction Per Cycle (IPC) strongly suggests that the compiler is not able to extract enough parallelism, causing under-utilization of the computation units.

Itanium’s hardware complexity was comparable with contemporary x86 designs in terms of functional units and their sizes. The transistor count approximations suggested comparable sizes between Itanium and Pentium, with Itanium slightly over double the size of Pentium, which according to Moore’s law is indeed approximately the time difference in release dates.

The business strategy analysis applied the differential strategy framework and applied it to this Itanium case study. The differential framework did seem to indicate a reason why Itanium might have failed, namely due to its lack of clear value propositions compared to contemporaries. As discussed in the technical analyses, the performance of Itanium was not stellar nor detrimental. But since it lacked an aspect where it clearly excelled, either quantitative or qualitative, it had no ground to compete.

The same can be done with the next sub-question R4: “Is there a single key business strategic decision that hindered Itanium’s wide market adoption?”. It can be broken down into the following sub-hypotheses: Public awareness, target audience alignment, public reputation,

8.2. Answering the research questions

To conclude the main question: “Why did Itanium fail?”, a piece-wise approach follows which subquestions are answered first to piece together the final answer.

R1. Can Itanium's originally intended objectives be identified from existing literature?

This question was already answered in the background and literature study, There were many indicators of Intel’s intention for Itanium to replace IA-32. Some of the features in Itanium seemed to point at the CPU being targeted at data centers specifically, so that narrative would also mean it inherits the goals of being a good data center chip: A focus on scaling (large integers, more memory) through 64-bit, implementing data center critical features such as Reliability, Availability, Serviceability (RAS). Additionally, to outcompete its competitors, execution time is an often used metric for performance. This means that Itanium had to compete on the field of performance. It was set out to do so using its EPIC architecture.

R2. Which metrics can quantitatively and qualitatively assess Itanium's success or failure?

Given Itanium’s goals as listed in the previous answer to **R1**, one could make a list of these features and their measurements:

- 64-bit (qualitative)
- Better performance (execution time)
- Data center features (such as RAS, ECC, scalability) (qualitative)

Failing in these can allude to a clear seeming failure. However, Itanium did not seem to fail any of these, it had 64-bit, it had decent performance, and had all the features perfectly suited for data center use cases. Clearly there must be another reason that lead to Itanium’s failure.

R3. Is there a single key technological factor that hindered Itanium's market adoption?

This section exists out of nine sub-questions. Each sub-question will first be answered, then afterwards conclude whether any technological or business strategic factors hindered Itaniums market success.

R3.1. How does EPIC compare with Superscalar in theoretical performance comparison?

In dynamic workloads, Superscalar OoO generally outperforms Itanium. In very specific workloads with lots of static ILP, such as Matrix multiplications, Itanium may have outperformed x86 due to the sheer amount of parallel execution units. However in practice, x86 has been extended to have SIMD, which would perform similarly to Itanium in these workloads. Additionally, modern CPU’s have wide-OoO systems [3] [17], where the amount of parallel instruction streams is equal or even wider than Itanium.

R3.2. How does Itanium's SPEC2000 performance compare with other architectures?

While in theory, Itanium does have niche applications where it could outperform x86. However, these workloads have also been saturated with alternative solutions such as GPGPU programming, and other accelerators. While performance sometimes did fall short (especially on legacy benchmarks), it had the potential to be on-par.

R3.3. How do IA-64's program sizes compare with competitors?

Many other CPU’s and technologies were developed throughout the lifetime of Itanium. These could have an impact on Itanium, even indirectly. In terms of technology, their performance was not significantly better, the compiler might have been an issue, but the debate whether Itanium could have prevailed if more time and effort was put into the compiler is a challenging one to answer. Al in al, it does seem like contemporary technologies had an integral role in Itanium’s failure.

R3.4. How significant is the proportion of NOPs in typical IA-64 binaries to performance?

With performance being average according to the performance comparative analysis, but prices high like according to the background study, it did seem like Itanium was not cost-efficient compared to its

competition.

R3.5. How much Instruction Level Parallelism is expressed in IA-64 compiled binaries?
In the market analysis, the lack of software support was often mentioned. This could allude to some form of its shortcomings in compatibility with existing x86 infrastructure.

R3.6. Do EPIC architectures reduce hardware complexity in CPU floor plans?
The hardware complexity comparative analysis showed that the complexity of Itanium's hardware in relation to contemporary x86 Pentium hardware was quite comparable. There did not seem to be any specific issues in the hardware complexity that might have caused this according to this study. On peculiarity was that Itanium was seemingly designed to keep hardware less complex than superscalar systems. However, this study seemed to show that Itanium's hardware was just as complex.

R3.7. How aware was the public of Itanium's existence and purpose?
As mentioned in the market analysis, Itanium's existence was acknowledged, but the purpose was not well explored or conveyed before it reached the market. There was confusement about what Itanium's target audience was, clear from the mixed performance results. Itanium did not perform well in general purpose environments, but still did target those segments.

R3.8. What does Itanium's popularity spikes imply about its public reception?
The public's initial response to Itanium was mixed, but clearly did gain traction up to 2005. The big set-back for Itanium's reputation was caused by Intel refocusing Itanium's goals to target a different market segment, instead of replace IA-32 completely. The public may have taken this as Intel stepping back from their commitment towards Itanium to reach its ambitious goals.

R3.9. Does the Differentiation Strategy framework shape Itanium's market placement?
According to the differential strategy framework, Itanium's downfall have been caused by x86's fast adoption to the new data center requirements, causing Itanium to no longer be differentiable competitively in terms of qualitative features. However as mentioned in Chapter 4: Performance Comparative Analysis, there was still some niche grounds for Itanium to outperform x86. Itanium only stuck to those niche markets from that point forward.

R3. Conclusion
To conclude **R3.**, there were many technological and business strategic factors that influenced Itanium's downfall. However, this research concludes that the downfall was not attributable to a single technological flaw. Each of the listed flaws have a surmountable solution. The business strategic framework shows how Itanium could have been difficult to market in competition with the rapid developments in x86.

8.3. Conclusion
Itanium was supposed to bring 64-bit, ILP and data center features, but the existing x86-family was eventually able to achieve this, so the extra investment was not well motivated.

8.3.1. Hypothetical success
The only ways Itanium could have succeeded according to this study is if Itanium would have been able to differentiate itself over one or more features: ILP, 64-bit or data center features. If hypothetically IA-64 remained to be the only 64-bit architecture, it would have found widespread attention, and the compilers would have developed more maturely. Or if - hypothetically - ILP was indeed too difficult to extract from OoO hardware, then it could also have gained widespread adoption.

8.3.2. Realistic hindsight lessons learned
If Itanium would have been targeted at niche markets first (such as supercomputing or mission critical), then it could prove its benefits, while building a software platform and initiate a reason for compiler research. In this market, Itanium could be extremely successful in its Blue Ocean [21] situation, it could have gotten positive reviews and a positive snowballing effect. This would probably still not reach widespread adoption, but it would be considered a success and turn a profit in its niche.

8.4. Contributions

This thesis has made several distinct contributions to the academic community and commercial innovators.

8.4.1. Methodology contributions Automated Sentiment analysis

This thesis explored the use of an automated sentiment analysis as a novel approach to quantitatively analyze a topic which traditionally has been a qualitative process. This introduces a structured and reproducible approach to assess meta-analyses and systematic public reviews and perceptions.

With the recent considerable improvements and popularization of Large Language Models, new methodologies are developed every day in the automation of different tasks. One example of such tasks, is the sentiment analysis. The automation of this task has seen significant improvements using these improved LLM's.

Using this new approach did have some complications in e.g. prompt engineering Large Language Model to follow the intended steps, to help with the reasoning process for better performance.

This study methodology could be useful, is to gather statistics about the public's opinions for principle CPU architects and as automated feedback analysis for better marketing and business strategic decisions. For example, this study showed quantitatively show how users reacted to the positive and negative aspects of Itanium execution performance, which brought attention to the fact that Itanium's target audience - and thus strengths and weaknesses - were not well enough marketed.

8.4.2. Methodology contributions large static analysis

The static analysis study also used a relatively novel approach to running statistics over large amounts of compiled binaries. The more general approach is to take a program, or a suite to disassemble for statistics on this program. However, to do statistics for the overall use of a General Purpose Instruction Set Architecture this approach can bring more insights. With the large amount of data, patterns such as certain use cases e.g. vector operations, or more branches, or more logical functions, etc., can be analyzed over certain types of files or programs. The use case for such a study is quite slim. Generally only useful for ISA or CPU design.

8.4.3. Hardware enthusiasts community contribution

As a hardware enthusiast myself, I am proud to also give something (ever so slightly) back to the community in regards to the speculated CPU die floor plan annotations. I have studied many of these myself out of curiosity, but making my own and contributing back to the community could hopefully leave a positive impact on another such enthusiasts.

8.5. Future work and recommendations

8.5.1. Effective limitations of EPIC

While this thesis has alluded to some use cases where EPIC is fundamentally limited compared to superscalar designs, it could provide a more definitive answer to the performance question to do a comprehensive study on EPIC's strengths and limitations.

8.5.2. Performance benchmarks on effective Itanium hardware

This thesis used online benchmarks, emulation and theoretical upper limits to calculate the performance characteristics of Itanium code, and concludes that Itanium may have performed well in niche use cases. A more definitive answer to where Itanium - and as extension: EPIC - has better performance could bring more clarity in what Itanium's target audience should have been.

References

[1] Donald Alpert and Dror Avnon. “Architecture of the Pentium Microprocessor”. In: *Micro, IEEE* 13 (July 1993), pp. 11–21. DOI: 10.1109/40.216745.

[2] Artificial Analysis. *Gemini 1.5 Flash*. https://artificialanalysis.ai/models/gemini-1-5-flash. Accessed: June 2024. 2023.

[3] Arm Limited. *Arm Cortex-X1 Core Technical Reference Manual*. 2020. URL: https://developer.arm.com/documentation/101433/latest/.

[4] Babbage. “iAPX432: Gordon Moore, Risk and Intel’s Super-CISC failure”. In: *The Chip Letter* (Apr. 2, 2023). Accessed: June 2024. URL: https://thechipletter.substack.com/p/iapx432-gordon-moore-risk-and-intels.

[5] Denis Barthou et al. “Loop Optimization using Hierarchical Compilation and Kernel Decomposition”. In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO’07)*. San Jose, CA, USA: IEEE, 2007, pp. 170–184. DOI: 10.1109/CGO.2007.32.

[6] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. “Power, performance and energy efficiency of high-performance processors: A comparative study”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 9.4 (2013), pp. 1–23.

[7] David Blustein. “The Psychology of Working: A New Perspective for Career Development, Counseling, and Public Policy”. In: *The Psychology of Working: A New Perspective for Career Development, Counseling, and Public Policy* (Jan. 2006), pp. 1–360. DOI: 10.4324/9780203935477.

[8] Chlamchowder. *Intel Microarchitectures - Merced*. Accessed: June 2024. 2021. URL: https://en.wikichip.org/wiki/intel/microarchitectures/merced.

[9] Clamchowder. *Intel’s Netburst: Failure is a Foundation for Success*. Accessed: June 2024. 2022. URL: https://chipsandcheese.com/2022/06/17/intels-netburst-failure-is-a-foundation-for-success/.

[10] Intel Corporation. *i860 64-bit Microprocessor Programmer’s Reference Manual*. 1989.

[11] Intel Corporation. *Intel® ItaniumTM Architecture Software Developer’s Manual Volume 2: System Architecture*. Intel Corporation, Dec. 2001.

[12] Intel Corporation. *Intel®Itanium™Processor Reference Manual for Software Optimization*. Document Number: 245473-003. Nov. 2001. URL: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/itanium-processor-reference-manual-software-optimization.pdf.

[13] James C. Dehnert et al. “The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges”. In: *Micro, IEEE* (2000).

[14] European Space Agency. *The SPARC Architecture Manual: Version 8*. Accessed: June 2024. Gaisler Research. 2022. URL: https://www.gaisler.com/doc/sparcv8.pdf.

[15] Joseph A. Fisher. “Very long instruction word architectures and the ELI-512”. In: *Proceedings of the 10th Annual International Symposium on Computer Architecture* (1983), pp. 140–150.

[16] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2012.

[17] Apple Inc. *Apple M1 System on a Chip*. Tech. rep. 2020. URL: https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/.

[18] Intel. *Intel revenue from 2014 to 2023, by segment (in billion U.S. dollars)*. Graph. Accessed: June 2024. Jan. 2024. URL: https://www-statista-com.tudelft.idm.oclc.org/statistics/495928/net-revenue-of-intel-by-segment/.

[19] Intel. *Introduction to the iAPX 432 architecture*. 1981, p. 73.

[20] *ISO/IEC 9899:2018: Programming languages – C*. Geneva, Switzerland: International Organization for Standardization, 2018. URL: https://www.iso.org/standard/68564.html.

[21] W. Chan Kim and Renée Mauborgne. *Blue Ocean Strategy: How to Create Uncontested Market Space and Make the Competition Irrelevant*. Boston, MA: Harvard Business Review Press, 2005.

[22] Peep Laja. *What is a Differentiation Strategy, and How Does It Work?* Accessed: June 2024. 2023. URL: https://cxl.com/blog/differentiation-strategy/.

[23] John Markoff. "Inside Intel, The Future is Riding on the Merced Chip". In: *The New York Times, republished by The Jerusalem Post* (Apr. 1998). URL: https://www.nytimes.com/1998/04/05/technology/inside-intel-the-future-is-riding-on-the-merced-chip.html.

[24] Cameron McNairy and Don Soltis. "Itanium 2 Processor Microarchitecture". In: *IEEE Micro* 23.2 (2003), pp. 44–55.

[25] Mixeur. *x86 CPUs Guide*. Accessed: 2024-06-25. 2024. URL: https://www.cpu-world.com/.

[26] James Niccolai. "Intel shifts gears on Itanium, raising questions about the server chip's future". In: *PCWorld* (Feb. 2013). Accessed: June 2024. URL: https://www.pcworld.com/article/456882/intel-shifts-gears-on-itanium-raising-questions-about-the-server-chips-future.html.

[27] Timo Schick et al. "Teaching Large Language Models to Self-Debug". In: *arXiv preprint arXiv:2304.06832* (2023).

[28] Standard Performance Evaluation Corporation. *SPEC CPU2000 Benchmark Results*. Accessed: June 2024. 2020. URL: https://www.spec.org/cpu2000/results/.

[29] The Register. *Intel processors and prices*. Accessed: June 2024. Dec. 2001. URL: https://www.theregister.com/2001/12/03/intel_processors_and_prices/.

[30] Roger Uy. "Powerhouses of the future: Comparing the RISC and CISC processor designs". In: *The LaSallian* (2020). URL: https://thelasallian.com/2020/12/15/powerhouses-of-the-future-comparing-the-risc-and-cisc-processor-designs/.

[31] S. Vassiliadis, B. Blaner, and R. J. Eickemeyer. "SCISM: A scalable compound instruction set machine". In: *IBM Journal of Research and Development* 38.1 (1994), pp. 59–78.

[32] S. Vassiliadis, B. Blaner, and R. J. Eickemeyer. "SCISM: A scalable compound instruction set machine". In: *IBM Journal of Research and Development* 38.1 (1994), pp. 59–78. DOI: 10.1147/rd.381.0059.

[33] Xuezhi Wang et al. "Self-Consistency Improves Chain of Thought Reasoning in Language Models". In: *arXiv preprint arXiv:2203.11171* (2022).

[34] Jason Wei et al. "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models". In: *arXiv preprint arXiv:2201.11903* (2022).

A

Source Code Example

This appendix contains the code of the static analysis tool which recursively runs over objdump-generated disassemblies in a directory and performs (in this case) an instruction and argument counting analysis. This code can easily be modified to crunch numbers and format it to custom preferences.

The code consists of four parts: A directory walker, a callback which parses out the assembly from the text files, a argument parser (which can classify addressing modes such as registers, memory addresses, immediates, floating point registers, etc.) and a thread-safe statistics merger, which only flushes the data to the global structure when the file is done instead of at all times.

```
1  package transscription;
2
3  import java.io.BufferedReader;
4  import java.io.File;
5  import java.io.FileNotFoundException;
6  import java.io.FileReader;
7  import java.io.IOException;
8  import java.io.PrintWriter;
9  import java.nio.file.Files;
10 import java.nio.file.Path;
11 import java.nio.file.Paths;
12 import java.util.ArrayList;
13 import java.util.Comparator;
14 import java.util.HashMap;
15 import java.util.Map;
16 import java.util.Map.Entry;
17
18 public class DirectParsingArgs {
19
20
21         static PrintWriter out;
22         static long globalTimer = System.currentTimeMillis();
23         static HashMap<String, Long> globalHashes = new HashMap<String, Long>();
24
25
26         static String arg(String line, final int start, final int end) {
27                 sw:switch(line.charAt(start)) {
28                 case 'r': {
29                         if((start + 2 == end && line.charAt(start+1) >= '0' && line.charAt(
                                start+1) <= '9') ||
30                                         (start + 3 == end && line.charAt(start+1) >= '1' &&
                                             line.charAt(start+1) <= '9'
31                                          && line.charAt(start+2) >= '0' && line.charAt(start
                                              +2) <= '9') ||
32                                         (start + 4 == end && line.charAt(start+1) == '1'
33                                          && line.charAt(start+2) >= '0' && line.charAt(start
                                              +2) <= '2'
34                                          && line.charAt(start+3) >= '0' && line.charAt(start
                                              +3) <= '9')) {
```

```
                                return line.substring(start, end);
                        }
                }break;
                case 'b': {
                        if(start + 2 == end && line.charAt(start+1) >= '0' && line.charAt(
                            start+1) <= '7') {
                                return line.substring(start, end);
                        }
                }break;
                case 'p': {
                        if((start + 2 == end && line.charAt(start+1) >= '0' && line.charAt(
                            start+1) <= '9') ||
                                        (start + 3 == end && line.charAt(start+1) >= '1' &&
                                            line.charAt(start+1) <= '6'
                                         && line.charAt(start+2) >= '0' && line.charAt(start
                                             +2) <= '9')) {
                                return line.substring(start, end);
                        }
                }break;
                case 'f': {
                        if((start + 2 == end && line.charAt(start+1) >= '0' && line.charAt(
                            start+1) <= '9') ||
                                        (start + 3 == end && line.charAt(start+1) >= '1' &&
                                            line.charAt(start+1) <= '9'
                                         && line.charAt(start+2) >= '0' && line.charAt(start
                                             +2) <= '9') ||
                                        (start + 4 == end && line.charAt(start+1) == '1'
                                         && line.charAt(start+2) >= '0' && line.charAt(start
                                             +2) <= '2'
                                         && line.charAt(start+3) >= '0' && line.charAt(start
                                             +3) <= '9')) {
                                return line.substring(start, end);
                        }
                }break;
                case '-': {
                        return "imm";
                }
                case '0': {
                        if((start + 1 == end) || (start + 2 < end && line.charAt(start+1) ==
                            'x')) {
                                return "imm";
                        }
                }
                case '1':case '2':case '3':case '4':case '5':case '6':case '7':case '8':case
                    '9':{
                        for(int i = start+1; i < end-1; i++) {
                                if(line.charAt(i) < '0' || line.charAt(i) > '9')
                                        break sw;
                        }
                        return "imm";
                }
                case '[': {
                        return line.substring(start, end);
                }
                case 'a': {
                        if((start + 3 == end && line.charAt(start+1) == 'r' && line.charAt(
                            start+2) >= '0' && line.charAt(start+2) <= '9') ||
                                        (start + 4 == end && line.charAt(start+1) == 'r' &&
                                            line.charAt(start+2) >= '1' && line.charAt(start
                                            +2) <= '9'
                                         && line.charAt(start+3) >= '0' && line.charAt(start
                                             +3) <= '9') ||
                                        (start + 5 == end && line.charAt(start+1) == 'r' &&
                                            line.charAt(start+2) == '1'
                                         && line.charAt(start+3) >= '0' && line.charAt(start
                                             +3) <= '2'
                                         && line.charAt(start+4) >= '0' && line.charAt(start
                                             +4) <= '9')) {
                                return line.substring(start, end);
                        }
                }break;
```

```
88                  case 'c': {
89                          if((start + 3 == end && line.charAt(start+1) == 'r' && line.charAt(
                                start+2) >= '0' && line.charAt(start+2) <= '9') ||
90                                          (start + 4 == end && line.charAt(start+1) == 'r' &&
                                                line.charAt(start+2) >= '1' && line.charAt(start
                                                +2) <= '9'
91                                           && line.charAt(start+3) >= '0' && line.charAt(start
                                                +3) <= '9') ||
92                                          (start + 5 == end && line.charAt(start+1) == 'r' &&
                                                line.charAt(start+2) == '1'
93                                           && line.charAt(start+3) >= '0' && line.charAt(start
                                                +3) <= '2'
94                                           && line.charAt(start+4) >= '0' && line.charAt(start
                                                +4) <= '9')) {
95                                  return line.substring(start, end);
96                          }
97                  }break;
98                  }
99
100                 boolean nothing = false;
101                 for(int i = start; i < end-1; i++) {
102                         char c = line.charAt(i);
103                         if((c < '0' || c > '9') && (c < 'a' || c > 'f')) {
104                                 nothing = true;
105                                 break;
106                         }
107                 }
108                 if(!nothing) {
109                         return "target";
110                 }
111
112                 for(int i = start; i < end-1; i++) {
113                         char c = line.charAt(i);
114                         if((c < '0' || c > '9') && (c < 'a' || c > 'f')) {
115                                 nothing = true;
116                                 break;
117                         }
118                 }
119                 return line.substring(start, end);
120
121         }
122
123         static void processSegment(HashMap<String, Long> localHashMap) {
124                 synchronized (globalHashes) {
125                         for (Map.Entry<String, Long> entry : localHashMap.entrySet()) {
126                                 globalHashes.merge(entry.getKey(), entry.getValue(), Long::
                                        sum);
127                         }
128
129
130                         if(globalTimer < System.currentTimeMillis()) {
131                                 globalTimer = System.currentTimeMillis() + 100000;
132
133                                 try {
134                                         out = new PrintWriter(new File("stuff.txt"));
135                                         for(Map.Entry<String, Long> entry : globalHashes.
                                                entrySet()) {
136                                                 out.println(entry.getValue() +"\t"+ entry.
                                                        getKey().replace('␣', '\t'));
137                                         }
138                                         out.flush();
139                                         out.close();
140                                 } catch (FileNotFoundException e) {
141                                         e.printStackTrace();
142                                 }
143                         }
144
145                         if(globalHashes.size() > 0x100_0000) {
146                                 ArrayList<Entry<String, Long>> bla = new ArrayList<>();
147                                 bla.addAll(globalHashes.entrySet());
148                                 bla.sort(new Comparator<Entry<String, Long>>() {
```

```
149
150                                     @Override
151                                     public int compare(Entry<String, Long> o1, Entry<
                                            String, Long> o2) {
152                                         return Long.compare(o2.getValue(),o1.getValue
                                                ());
153                                     }
154                                 });
155                                 int size = globalHashes.size();
156                                 for(int i = 0x100000; i < size; i++)
157                                     globalHashes.remove(bla.get(i).getKey());
158
159                         }
160                 }
161         }
162         static ArrayList<Long> sizes = new ArrayList<Long>();
163         static void dis(Path path) {
164                 HashMap<String, Long> localHashes = new HashMap<String, Long>();
165                 try (BufferedReader br = new BufferedReader(new FileReader(path.toFile()), 0
                        x10000)) {
166
167                         br.readLine();
168                         String l = br.readLine();
169                         if(!l.endsWith("file␣format␣elf64-ia64-little") && !l.endsWith("file␣
                                format␣pei-ia64")) {
170                                 return;
171                         }
172
173                         while(br.ready()) {
174                                 String line = br.readLine();
175                                 int len = line.length();
176                                 int addrEnd = line.indexOf(':');
177                                 if(addrEnd != -1 && addrEnd != len-1 && addrEnd != len-14 &&
                                        line.charAt(addrEnd+32) == '␣') {
178
179                                         int i;
180                                         for(i = addrEnd+33; i < len; i++) {
181                                                 if(line.charAt(i) == ';') {
182                                                         i = len;
183                                                         break;
184                                                 }
185                                                 if(line.charAt(i) == '␣')
186                                                         break;
187                                         }
188                                         boolean endspace = false;
189                                         i++;
190                                         int start;
191                                         for(start = i; i < len; i++) {
192                                                 if(line.charAt(i) == ',' || line.charAt(i) ==
                                                        '=') {
193
194                                                         String newString = arg(line, start, i
                                                                );
195                                                         localHashes.put(newString,
                                                                localHashes.getOrDefault(
                                                                newString, 0l)+1);
196
197                                                         start = i+1;
198                                                 }
199                                                 if(line.charAt(i) == '␣' || line.charAt(i) ==
                                                        ';') {
200                                                         String newString = arg(line, start, i
                                                                );
201                                                         localHashes.put(newString,
                                                                localHashes.getOrDefault(
                                                                newString, 0l)+1);
202
203                                                         start = i+1;
204                                                         endspace = line.charAt(i) == '␣';
205                                                         break;
206                                                 }
```

```
207                                     }
208                                     if(start < i) {
209                                             if(endspace)
210                                             {
211                                                     String newString = "target";
212                                                     localHashes.put(newString,
                                                            localHashes.getOrDefault(
                                                            newString, 0l)+1);}
213                                             else {
214                                                     String newString = arg(line, start, i
                                                            );
215                                                     localHashes.put(newString,
                                                            localHashes.getOrDefault(
                                                            newString, 0l)+1);
216                                             }
217                                     }
218
219                                 }
220                         }
221
222
223                 } catch (IOException e) {
224                 } catch (NullPointerException e) {
225                 }
226                 processSegment(localHashes);
227         }
228
229
230         public static void main(String[] args) throws IOException {
231                 globalTimer = System.currentTimeMillis() + 10000;
232                 long startTime= System.currentTimeMillis();
233                 Files.walk(Paths.get("path/to/files"))
234                         .parallel()
235                         .filter(Files::isRegularFile)
236                         .forEach(DirectParsingArgs::dis);
237
238
239                 System.out.println("Took␣" + (System.currentTimeMillis()-startTime)/1000 + "s
                        ");
240
241                 out = new PrintWriter(new File("outputFile.txt"));
242                 ArrayList<Entry<String, Long>> arr = new ArrayList<>();
243                 arr.addAll(globalHashes.entrySet());
244                 arr.sort(new Comparator<Entry<String, Long>>() {
245                         @Override
246                         public int compare(Entry<String, Long> o1, Entry<String, Long> o2) {
247                                 return -Long.compare(o1.getValue(),o2.getValue());
248                         }
249                 });
250                 for(Map.Entry<String, Long> entry : arr) {
251                         out.println(entry.getValue() +"\t"+ entry.getKey().replace('␣', '\t')
                                );
252                 }
253                 out.flush();
254                 out.close();
255
256         }
257 }
```