Power- and Performance- Aware Architectures

Ramon Canal Corretger

Departament d’Arquitectura de Computadors
Universitat Politècnica de Catalunya

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor en Informàtica

Power- and Performance- Aware Architectures

Ramon Canal Corretger

Departament d’Arquitectura de Computadors
Universitat Politècnica de Catalunya

Advisors:

Antonio González Colás
Universitat Politècnica de Catalunya
James E. Smith
University of Wisconsin-Madison

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor en Informàtica

Preface

The scaling of silicon technology has been ongoing for over forty years. We are on the way to commercializing devices having a minimum feature size less than one-tenth of a micron. The push for miniaturization comes from the demand for higher functionality and higher performance at a lower cost. As a result, successively higher levels of integration have been driving up the power consumption of chips. Today, heat removal and power distribution are at the forefront of the problems faced by chip designers.

In recent years portability has become important. Historically, portable applications were characterized by low throughput requirements such as for a wristwatch. This is no longer true. Among the new portable applications are hand-held multimedia terminals with video display and capture, audio reproduction and capture, voice recognition, and handwriting recognition capabilities. These capabilities call for a tremendous amount of computational capacity. This computational capacity has to be realized with very low power requirements in order for the battery to have a satisfactory life span. This thesis is an attempt to provide techniques for low-power microarchitectures with high-computational capacity.

The first research of this thesis targets the reduction of the complexity and energy-requirements of the issue logic. An important effort has been deployed to reduce the energy requirements and the power dissipation of processors through novel mechanisms based on value compression. Several ultra-low power processor designs that use value compression are presented, as well as, two compile-time that show how the compiler can help in reducing the energy consumption.

Acknowledgements

"The essence of all beautiful art, all great art, is gratitude"
Friederich Nietzsche

De petit sempre pensava en quin punt me'n cansaria d'estudiar i aprendre coses noves. Després de ja uns quants anys he vist que aquest camí mai s'acaba. Ni després d'haver escrit aquesta tesi m'ha quedat la sensació d'haver arribat al final del camí. M'agradaria agrair a tothom que m'ha acompanyat aquest temps i que m'ha ajudat a créixer: la seva confiança, el seu suport i la seva ma estesa disposada a ajudar-me en els entrebancs.

M'agradaria començar pels meus directors de tesi –l'Antonio i el Jim- amb els quals he tingut la sort de compartir tot aquest treball. Ells m'han ensenyat el que és la recerca i el que és l'arquitectura de computadors. Als meus pares i germans també els hi vull agrair tot el suport durant aquest temps, casa sempre ha estat un oasis de tranquil·litat per aquells moments per pensar i reflexionar. Als meus cosins i tiets, la millor companyia els dies de festa.

També vull agrair a tot el Grup de Colònies a Borredà les estones compartides tots aquests anys, en especial, a la Queralt i al Ramon Ma. També dins d'aquest món, gràcies, a la gent de la Fundació Mn. Víctor Sallent, de Coordinació Catalana de Colònies Casals i Clubs d'Esplai (especialment a l'Anna i a la Romina) i últimament de la FIMCAP pels moments viscuts. A tota la "colla" d'amics de Berga i en concret a les Neus, l'Anna, la Sílvia i el Ricard, companys infatigables de tertúlies al voltant d'un cafè.

Vull agrair també a tots els becaris, professors i altres membres del Departament d'Arquitectura de Computadors de la UPC i de l'Intel-UPC Barcelona Research Center, la seva ajuda durant aquest temps i especialment al Josep-Llorenç, Fernando, Pedro, Joan-Manel, Enric, Jaume A., Jaume V., Josep Maria, Àlex A., Jordi, Pepe, Suso, Carlos M., Carles A., Montse, Carles N., Manel, Àlex R., Daniel J, Daniel O, Jesús, Ayose, Oliver, Xavi, Fran, Carmelo i a la resta de PBCs.

En resum, gràcies a tots els que m'heu acompanyat no només durant el període d'aquesta tesis sinó des de que vaig aparèixer en aquest món.

Als que mirareu aquest treball

des de la distància

Index

“Teachers open the door, but you must enter by yourself”
Chinese proverb

Chapter 1

Introduction

Energy consumption and power dissipation have become a key constraint in the design of processors. In the embedded segment, battery life is an issue, so the processor energy consumption has to be minimal. In the high-performance segment, the power dissipation is a limiting factor since the cooling mechanisms are becoming more expensive or even reaching its limits.

Although significant research is targeted to produce longer-life batteries and better cooling systems, contributions from other areas are becoming critical. This chapter presents the fundamentals of energy consumption -and power dissipation- and the power analysis of a conventional processor.

The invention of the transistor was a giant leap forward for low-power electronics that has remained unequaled to date, even by the virtual torrent of developments it forbore. The operation of a vacuum tube required several hundred volts and few watts of power. In comparison, the transistor required only miliwatts of power. Since the invention of the transistor, decades ago, through the years leading to the 21st century, power dissipation, though not entirely ignored, was of little concern. The greater emphasis was on performance and miniaturization. Applications powered by batteries –pocket calculators, hearing aids and, most importantly, wristwatches- drove low-power electronics. In all such applications, it is important to prolong the battery life as much as possible. And now, with the growing trend towards portable computing and wireless communication, power dissipation has become one of the most critical factors in the continued development of the microelectronics technology. There are two reasons for this:

1. To continue to improve the performance of the circuits and to integrate more functions into each chip, feature size has to continue to shrink. As a result, the magnitude of power per unit area is growing and the accompanying problem of heat removal and cooling is worsening. Examples are the general-purpose microprocessors used in desktop computers and servers. Even with the scaling down of the supply voltage, power dissipation has not come down. Figure 1 shows the power density for several comercial processors. As it is shown in the figure, the trend is to increase the power density to levels where the cooling mechanisms are unlikely to be effective enough.

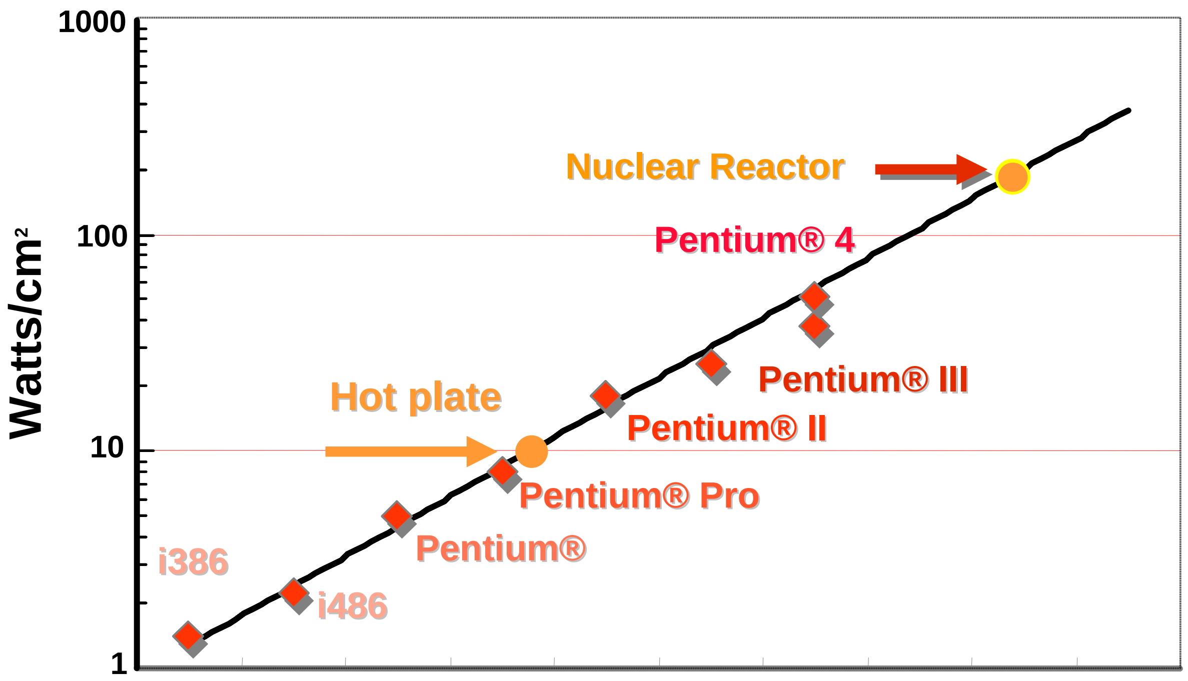


Figure 1: Power density for the Intel-32 family [6]

2. Portable battery-powered applications of the past were characterized by low computational requirements. The last few years have seen the emergence of portable applications that require a greater amount of processing power. Two vanguards of this processing model are the notebook computer and the personal digital assistant (PDA). People are beginning to expect to have access to the same computing power, information resources, and communication abilities when they are travelling as they do when they are at their desk. A representative of what the very near future holds is the portable multimedia terminal. Such terminals will accept voice input as well as hand-written (with a special pen on a touch-sensitive surface) input. Unfortunately, with the technology available today, effective speech or hand-writing recognition requires significant amounts of space and power.

As a result, today, it is widely accepted that power efficiency is a design goal at par in importance with miniaturization and performance. In spite of this acceptance, the practice of low-power design methodologies is being adopted at a slow pace due to the widespread changes called for by these methodologies. Minimizing energy consumption and power dissipation calls for conscious effort at each abstraction level and at each phase of the design process.

1.1 Sources of Power Dissipation

There are three sources of power dissipation in a digital complementary metal-oxide-semiconductor (CMOS) circuit. Logic transitions are the first source. As the transistors in a digital CMOS circuit transition back and forth between the two logic levels, the parasitic capacitances are charged and discharged. Current flows through the channel resistance of the transistors, and electrical energy is converted into heat and dissipated away (see Figure 2). This component of power dissipation is proportional to the supply voltage, transistor voltage swing, and the average switched capacitance per cycle. As the voltage swing in most cases is simply equal to the supply voltage, the dissipation due to transitions varies overall as the square of the supply voltage and linearly with the capacitance. Short-circuit currents that flow directly from supply to ground when the *n*-subnetwork and the *p*-subnetwork of a CMOS gate both conduct simultaneously are the second source of power dissipation (see Figure 3). With the input(s) to the gate stable at either logic level, only one of the two subnetworks conduct and no short-circuit currents flow. But when the output of a gate is changing in response to a change in the input, both subnetworks conduct simultaneously for a brief interval. The duration of the interval depends on the input and the output transition (rise or fall) times and so does the short-circuit dissipation. Both the above sources of power dissipation in CMOS circuits are related to transitions at gate outputs and are therefore collectively referred to as dynamic dissipation. In contrast, the third and the last source of dissipation is the leakage current that flows when the input(s) to, and therefore the outputs of, a gate are not changing (see Figure 4). This is called static dissipation. In recent technology, the relative magnitude of leakage current was low and was neglected. But as the supply voltage is scaled down to reduce dynamic power, MOS field-effect transistors (MOSFETs) with low threshold voltages have to be used. The lower the threshold voltage, the lower the degree to which MOSFETs in the logic gates are turned off and the higher is the standby leakage current.

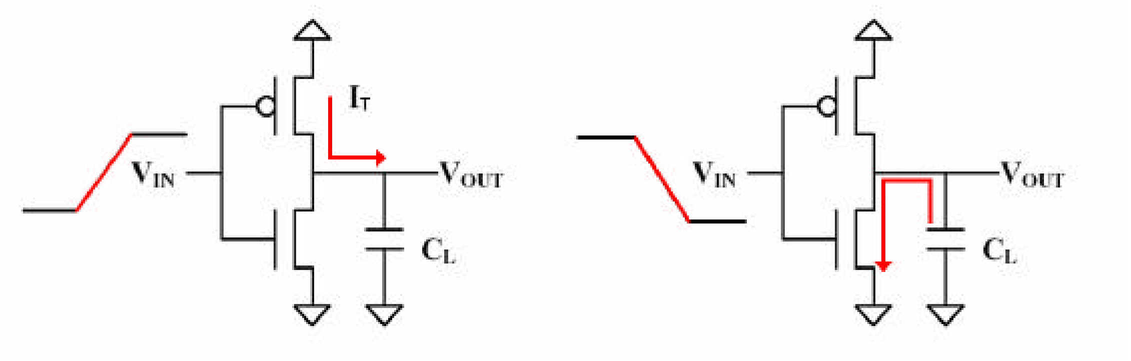


Figure 2: Circuit transition currents (left: charge, right: discharge)

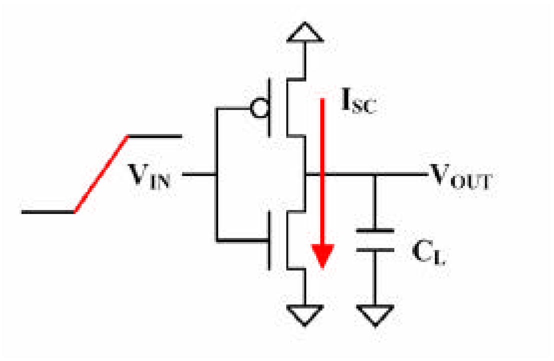


Figure 3: Short-circuit currents

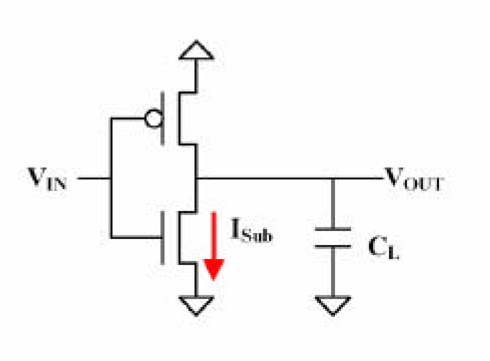


Figure 4: Leakage currents

1.2 Designing for Low Power

The power dissipation attributable to the three sources described above can be influenced at different levels of the overall design process.

Since the dominant component of power dissipation in CMOS circuits (due to logic transitions) varies as the square of the supply voltage, significant savings in power dissipation can be obtained from operation at a reduced supply voltage. If the supply voltage is reduced while the threshold voltages stay the same, reduced noise margins result and gates become slower. To improve these parameters, the threshold voltages need to be made smaller too. However, the subthreshold leakage current increases exponentially when the threshold voltage is reduced. The higher static dissipation may offset the reduction in the dynamic component of the dissipation. Hence, the devices need to be designed to have threshold voltages that maximize the net reduction in the dissipation and the delay can be compensated by other means –in order to retrain the system level throughput at the desired level.

One way of influencing the delay of a CMOS circuit is by changing the channel-width-to-channel-length ratio of the devices in the circuit. The power-delay product for an inverter driving another inverter through an interconnect of a certain length varies with the width-to-length ratio of the devices. If the interconnect capacitance is not significant, the power-delay product initially decreases and then increases when the width-to-length ratio is increased and the supply voltage is reduced to keep the delay constant. Hence, there exists a combination of the supply voltage and the width-to-length ratio that is optimal from the power-delay product consideration.

Circuit level choices also impact the power dissipation of CMOS circuits. Usually a number of approaches and topologies are available for implementing various logic and arithmetic functions. Choices between static versus dynamic style, pass-gate versus normal CMOS realization versus asynchronous circuits have to be made.

At the logic level, automatic tools can be used to locally transform the circuit and select realizations for its pieces from a precharacterized library so as to reduce transitions and parasitic capacitance at circuit nodes and therefore circuit power dissipation. At a higher level, various structural choices exist for realizing any given logic functions; for example, for an adder one can select among ripple-carry, carry-look-ahead, or carry-select realizations.

In synchronous circuits, even when the outputs computed by a block of combinatorial logic are not required, the block keeps computing its outputs from observed inputs every clock-cycle. In order to save power, entire execution units comprising combinatorial logic and their state registers can be put in stand-by mode by disabling the clock and/or powering down the unit. Special circuitry is required to detect and power-down unused units and power them up again when they are needed to be used.

The rate of increase in the total amount of memory per chip as well as rate of increase in the memory requirement of new applications has more than kept pace with the rate of reduction in power dissipation per bit of memory. As a result, in spite of the tremendous reductions in power dissipation obtained from each new process generation of memory to the next, in many applications, an important portion of power dissipation occurs in the memory.

Overall, a design for low power has implications at all the design levels. In this thesis, we target the microarchitecture level. At this level, several proposals are made and evaluated that reduce the dynamic energy consumption (most dominant in current technologies).

1.3 Power Breakdown

Figure 5 shows the power breakdown for the Alpha 21264 processor (as reported by Gowan et al. [29]). The clock network, the issue queue and the caches account, in this order, for most of the power dissipated. On one hand, the clock network is an issue mainly related to the layout. On the other hand, the rest of the structures are subject to architecture optimizations that can reduce significantly the energy consumption and the power dissipation as it is shown in the next chapters.

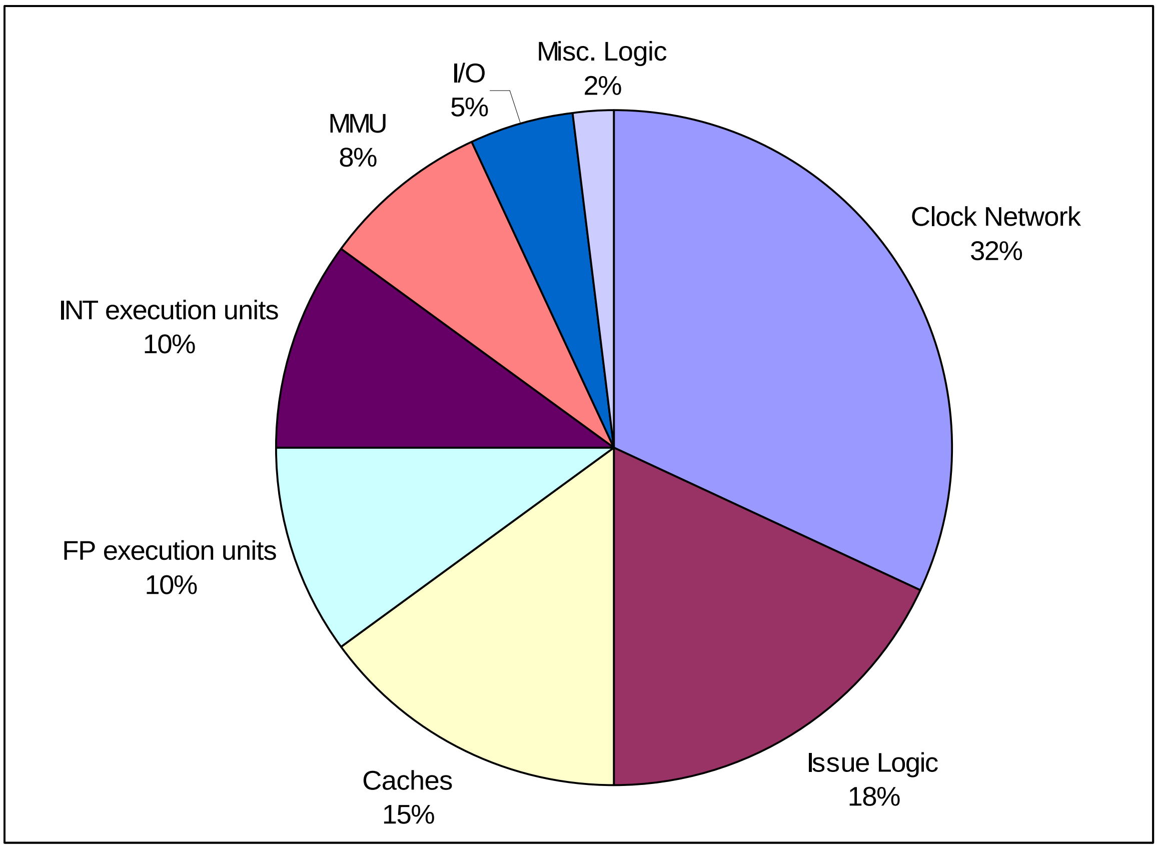


Figure 5: Alpha 21264 processor power breakdown [29]

1.4 Main Contributions

1.4.1 Contributions to the Issue Logic Design

As shown in Figure 5, the issue logic is one of the main energy consumers in a conventional microarchitecture. At the same time, the issue logic is a key part for exploiting large amounts of ILP (Instruction Level Parallelism), its energy budget, as well as its cycle time is critical in most superscalar processors. In order to tackle this issue, novel issue logic designs have been proposed in this thesis. These proposals are detailed in Chapter 2 and can be divided into two families:

- *Dependence-tracking schemes.* These schemes keep track of the producer-consumer relationships between instructions. Since the direct dependences are stored, the scheme eliminates most of the need for the associative wake-up logic –the part of the issue logic responsible for checking what instructions have their operands available, and thus, decide whether the instruction is ready for execution.
- *Prescheduling schemes.* Since most of the functional units have a fixed execution latency (all but the memory); at dispatch time, the mechanism schedules the instructions for execution according to the estimated availability cycle of its source operands and functional unit. Due to the deterministic latencies of the units and schedule time of the instructions, the cycle where the output value of an instruction will be available, can be already computed. As a consequence, these schemes eliminate most of the logic needed by the issue logic since the instructions will be only considered for issue just once (at the time they have been scheduled).

Three papers have been published related to this topic. The IEEE Micro paper entitled “Power- and Complexity- Aware Issue Logics” [1] is a survey of the techniques presented so far to reduce the complexity and reduce the energy requirements of the issue logic. The techniques presented in this work have been published in the International Conference on Supercomputing (2000 and 2001) [13][14].

1.4.2 Contributions to the Ultra-Low Power Processors

Chapter 3 shows how value compression can be an effective way to reduce energy requirements. Both for ultra-low power processors and high-performance ones, the values that flow through the pipeline determine the energy consumption of the structures in the datapath (mainly the register file, the data cache and the functional units). Since the energy consumption of these components depends directly on the values manipulated, we analyzed and proposed several value compression schemes that reduce the energy consumption of the processor by reducing the number of bits needed to be manipulated for each data value. In Chapter 4, several datapaths are redesigned by considering value compression. From this setup, different microarchitectures that exhibit different energy-performance tradeoffs are proposed and analyzed.

- A new microarchitecture targeted to embedded processors is proposed. In this architecture the datapath is redesigned in order to compute just the bytes that are significant.
- Several alternatives for a different set of energy/performance trade-offs are proposed. While the alternatives are still targeted to embedded processors, these show a compromise between the energy budget and the performance reduction.

The work in this chapter has been published in the paper entitled "Very Low Power Pipelines using Significance Compression" presented in the 33rd International Symposium on Microarchitecture (MICRO-33) [17].

1.4.3 Contributions to the high-performance processors

Chapter 5 shows how value compression can also reduce significantly the energy consumed by conventional high-performance processors. We show how hardware only schemes are able to reduce significantly the energy budget and, at the same time, we show how compile-time techniques can be an important support for reducing the energy requirements apart or on top of hardware-only value compression. In short, the main contributions are:

- A value-aware processor design. Where the structures are modified in order to take advantage of value compression and finally reduce its energy consumption.
- Two compile-time techniques that propagate the range of the values at compile-time and then re-encode the instructions to use narrower execution widths. In this case, the value compression is "communicated" to the processor through the use of narrower opcodes.
- A cooperative hardware-software approach that takes the advantages the two previous proposals. Both hardware and software approaches have strengths and weaknesses so a cooperative approach can yield the best energy reduction (and energy-performance tradeoff), while taking advantage of both schemes.

The proposals in this topic have been published in the 2004 International Symposium on Code Generation and Optimization [16] and a technical report [15] that describes all the findings on the compressibility of data values and its potential for energy reduction and it proposes the 64-bit value-aware processor design.

“Everything should be made as simple as possible, but not simpler”
Albert Einstein

Chapter 2

Complexity-Aware Issue Logics

The issue logic of dynamically scheduled superscalar processors is one of their most complex and power-dense parts. In this chapter, we present alternative issue-logic designs that are much simpler than the traditional ones while they retain most of its ability to exploit ILP. These alternative schemes are based on the observation that most values produced by a program are used by very few instructions, and the latencies of most operations are deterministic.

2.1 Introduction

Out-of-order issue is common in today's high-performance microprocessor due to its higher potential to exploit instruction-level parallelism (ILP) for some applications whose behavior is hard to predict at compile time (e.g. non-numeric applications such as SpecInt).

However, the hardware structures required by an out-of-order issue scheme are rather complex, which translates to significant delays that may challenge the cycle time [54] and the energy budget [27]. The complexity and energy consumption of the issue logic depends on a number of microarchitectural factors, mainly the size of the issue queue and the issue width [52][54][62]. These two parameters have experienced a continuous increase, and future projections suggest that this trend will continue in the near future. Therefore, the delay and energy consumption of the issue logic is expected to be even more critical in the future.

The main complexity of the issue logic comes from the associative search of the wake-up mechanism [62]. The wake-up uses long wires to broadcast the tags (and data, sometimes) to the non-ready instructions, and a large number of comparators that compare each broadcast tag with every source operand's tag.

Besides, the issue logic is not easy to pipeline [54] since this may prevent the back-to-back execution of some dependent instructions. Therefore, the issue logic may significantly impact the clock-cycle time. At the same time, the issue logic becomes a significant consumer of energy [27]. Recently, Gowan, Biro and Jackson [29] (Figure 5) reported that in an Alpha 21264 operating at the maximum operating frequency the issue logic accounts for the 18% of the total energy consumption. This percentage was even higher than the consumption of caches, which accounted just for 15% of the total.

To address this problem, various techniques have been proposed elsewhere. These techniques attempt to partition the central issue logic by means of a clustered architecture where the assignment of instructions to clusters is performed by either considering each instruction in turn [18] or managing larger instruction units such as trace cache lines [59] or loop iterations [42]. Another approach is to rely on the compiler to schedule the instructions, VLIW [25] architectures being an example of this approach.

We take a different approach in this thesis to tackle this problem. This approach is based on the observed properties of typical dynamic instruction streams. We first note that a vast majority of the register values generated by a program are read at most once. For instance, only about one out of four values generated by the Spec95 benchmarks is read more than once [22]. This feature suggests that the wake-up function could be implemented through a simple table that is indexed by the register identifier, avoiding the associative search. The second observation is that the latencies of most instructions are known (except for memory unit operations) and thus the time when a source operand will be available can be deterministically computed in most cases. This suggests an issue logic that schedules instructions based on the availability time of their operands.

We present different implementations of the issue logic based on the above two concepts and show that these schemes can significantly reduce the issue logic complexity with a minor impact on the IPC rate.

2.2 Dependence Tracking Schemes

2.2.1 N-use Issue Scheme

The N-use scheme is designed upon the observation that most register values are read very few times. For instance, only 22% of the values generated by the SpecInt95 and 25% of the FP register values produced by SpecFP95 are read more than once [22]. The N-use issue scheme is based on a table (we refer to it as N-use table) that, for each physical register, stores the first N instructions that read it in sequential order. We refer to the parameter N as the associativity degree of the issue logic. The scheme works as follows:

After being decoded, each instruction is inserted in the issue logic –dispatched- in a different way depending on the availability of its source operands:

- If all its operands are available, it is dispatched to a queue of ready instructions.
- If for each non-ready source operands there is a free slot in the N-use table in the corresponding operand, the instruction is dispatched to the table entries corresponding to the non-ready operands.
- If for any of the non-ready source operands there is no free slot in the N-use table entry corresponding to the source operand, the dispatch of instructions is stalled until the operand becomes ready. Alternatively, the issue logic can be extended with an issue queue where such instructions are dispatched and later issued either in-order or out-of-order. This queue could be small since it is used by few instructions.

Figure 6 shows a block diagram of the N-use issue scheme. This scheme consists of two main hardware parts. The first one is a ready queue, which contains instructions that have all their operands available. This queue issues the instructions to the functional units in-order. Instructions are dispatched to this queue if they meet the conditions in the above paragraph (1). The second component (N-use table) is a table that contains the first N instructions that read each physical register that is not available. The table will have N times the number of physical registers (N x #Physical Regs). Instructions are dispatched to this queue when they meet the conditions in the above paragraph (2). Since an instruction can have up to two source registers, it can be stored into two different entries, if both operands are not available. Each entry of the N-use table has an additional field that may point to another entry of the table. When an instruction is placed in two entries, each entry's pointer is set to point to the other entry. The pointer size is: $\lceil \log_2 N \times physicalregs \rceil$.

When the execution of an instruction completes, its physical register identifier is used to index the N-use table. If an instruction is found in the corresponding entry, then the pointer field is analyzed. If it does not point to any other entry (i.e., the pointer's value is NIL), the instruction is forwarded to the ready queue, since this indicates that this register was the only one that was not available for that instruction. Otherwise, the pointer's value is used to access the other entry of the N-use table where the same instruction resides, and the pointer of that other entry is set to point to NIL. When the physical register corresponding to the entry is available, the instruction will be forwarded to the ready queue since the pointer that the processor will find when it accesses this entry will be NIL.

In the basic N-use scheme described above, if a decoded instruction has a source operand that is not ready and it does not correspond to the first N uses of such value, the decode and dispatch of instructions is stalled until this operand becomes ready. Then, it is dispatched to the ready queue. Alternatively, the basic scheme could be extended for increased performance with an extra buffer (I- buffer) shown in Figure 6. Basically, it consists of a buffer (I-buffer) where the instructions that have non-ready operands that are not in the first N uses of them are dispatched. The instructions from this I-buffer are issued to the functional units after their operands are available. We have investigated two different organizations for the I-buffer, with very different hardware cost and complexity: in-order and an out-of-order issue policies.

Note that both the basic scheme and the extended scheme where the I-buffer uses an in-order issue policy do not require any associative search for the issue logic, which is a significant simplification with respect to a conventional out-of-order issue mechanism.

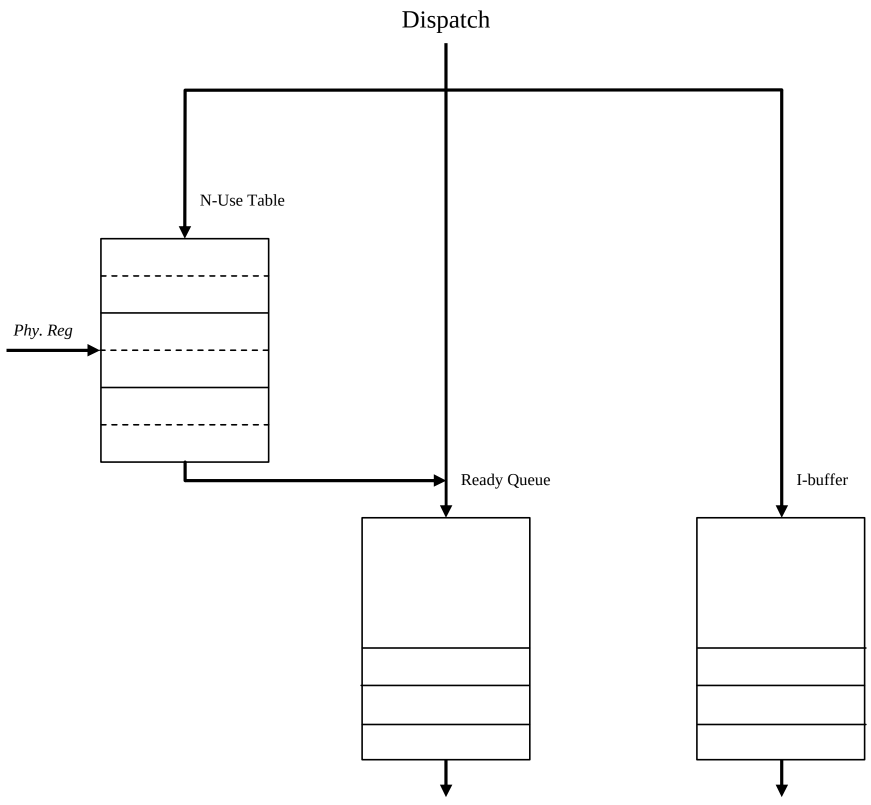
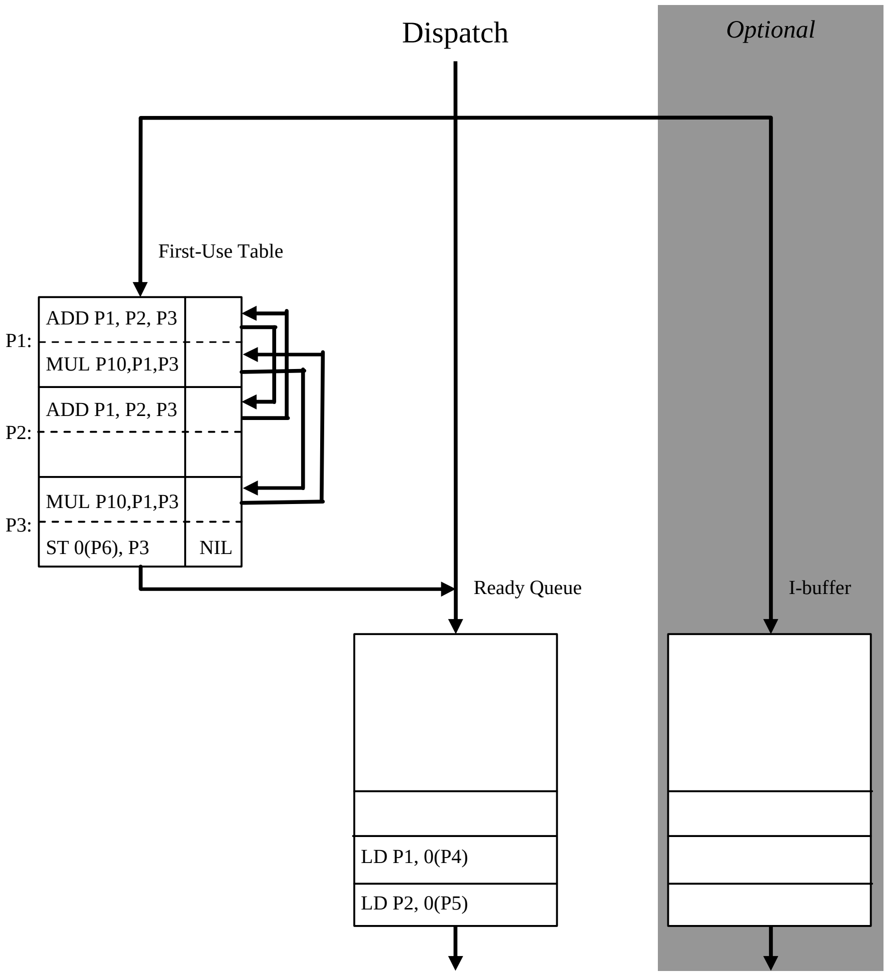


Figure 6: N-use issue logic design

In Figure 7, we can see an example of the use of the 2-use scheme for a sample code. We assume that all five instructions can be dispatched in the current cycle and that registers P4, P5 and P6 are available at this cycle. Since the two loads have their operands ready they are sent to the Ready queue. When dispatching the ADD instruction, the processor detects that this operation does not have its operands ready and both correspond to the first use of them. Thus, this instruction is steered to the N-use table into the entries corresponding to its source registers. Furthermore, each entry is made to point to the other one. As far as the multiply is concerned, it is the first use of P3 and the second of P1. Since this implementation allows 2 instructions (uses) per register in the N-use table this instruction is kept in the 2-use table. Regarding the store instruction, the operand P3 is not available and it is the second use of P3, thus the instruction is stored in the corresponding N-use table entry. The pointer is set to NIL since P3 is the only unavailable operand.



Sample code:

LD R1, 0(R4)
LD R2, 0(R5)
ADD R1, R2, R3
MUL R10, R1, R3
ST 0(R6), R3

Code after renaming:

LD P1, 0(P4)
LD P2, 0(P5)
ADD P1, P2, P3
MUL P10, P1, P3
ST 0(P6), P3

Figure 7: Example of the 2-use scheme

In order to handle instruction squashing, for example in the case of a branch miss-prediction, each entry in the N-use table will hold a tag that indicates the position of that instruction in the ROB (ReOrder Buffer). When flushing the instructions, the tag of the misspredicted branch is sent to the N-use table. If an entry of the N-Use table holds an instruction, the tag of that will be compared to the one of the misspredicted branch. In case that the instruction has been fetched after the branch, that entry will be set to invalid and thus squashed.

2.3 Prescheduling Schemes

Another approach to the issue logic is based on the fact that the latency of most instructions is known when they are decoded. Thus, we could dynamically schedule the instructions following a similar approach to that implemented by a static scheduler. In other words, the order in which instructions will be executed is determined at the decode stage. The only problem this scheme has to face is the varying latency of the memory accesses. We consider two main approaches for memory accesses. The first one is to assume that the latency is not known and thus, any instruction depending on a memory access will have to wait to be scheduled till the latency of the access is known. The second alternative is to assume that memory accesses have a given latency and thus, the instructions depending on the memory access can be scheduled according to the assumed latency.

2.3.1 Distance Scheme

The Distance Scheme implements the first alternative described above. In this scheme, the instructions are scheduled after decode according to its producers' latencies. If any producer's latency is unknown at this time, the scheme provides an extra hardware that will hold the instructions until the cycle when their operands will be available is known.

Figure 8 shows a block diagram of the Distance issue logic. This scheme consists of three main blocks. First, we have a table that for each physical register contains the cycle when its value will be produced, if it is known. This table is called the register-availability table.

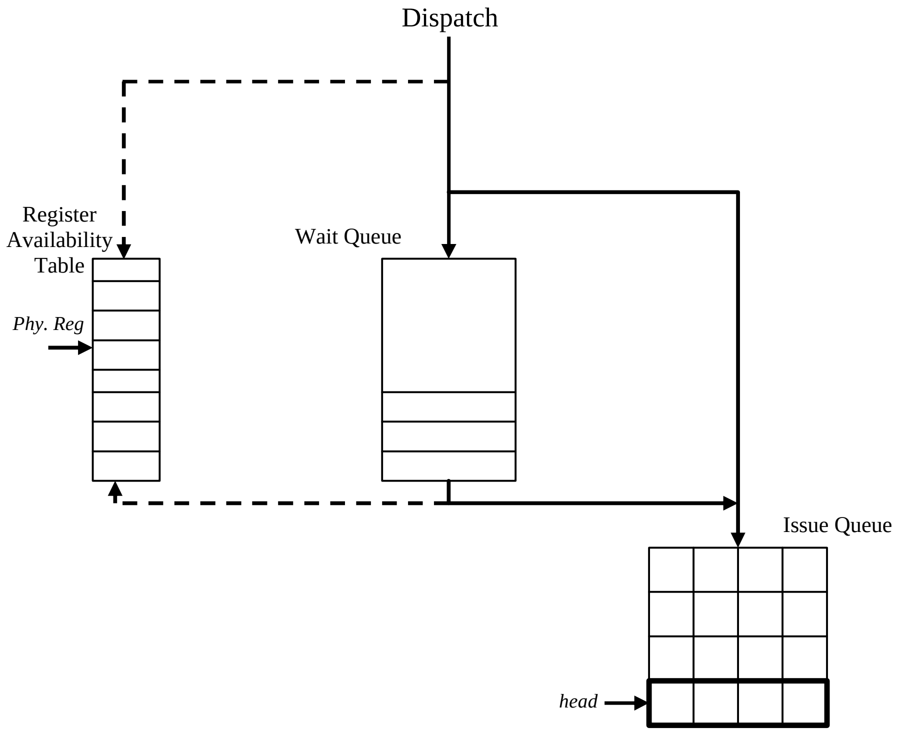


Figure 8: Distance scheme issue-logic

The second block is the Wait queue, which holds the instructions that do not know when one (or both) of their operands will be produced. When an instruction in the Wait queue knows the availability time of all its source operands, it is removed from the queue and placed in the Issue queue. Besides, the time when its destination register will be available is computed and it is used to update the register availability table. The identifier of its destination register along with its availability time is then broadcast to the Wait queue. Actually, the instructions placed in this queue will be those depending directly or indirectly on a load which has not been completed yet, as it is explained in short.

The third block is the Issue queue. Instructions are issued always from this queue. For each instruction in the queue, this block contains information regarding the cycle when the instruction will be issued. Conceptually, it can be regarded as a circular buffer where for each entry there are as many instruction slots as the issue width and each entry corresponds to one cycle. Thus, this queue contains the instructions in the order that they will be executed and separated by a distance that ensures that dependencies will be obeyed if at every cycle the processor issues the instructions in the head entry. Thus, the issue logic for this queue is very simple: at each cycle the instructions in the head of the queue are issued and the head pointer is increased by one. In Section 2.4.3, the depth of the Issue queue is empirically determined. Instructions are placed on the issue queue only when the time when its source operands will be available is known. The location in the issue queue is computed as follows. First, the maximum of the availability time of its source operands (*MaxSource*) is calculated and then, the difference between this value and the current cycle indicates the displacement with respect to the head pointer. The instruction is placed on the first free slot starting at that cycle.

Once an instruction is placed on the issue queue, the time when its output register will be available is computed as *MaxSource* plus the latency of the instruction plus the additional delay due to conflicts in the issue queue with previously scheduled instructions. This value is used to update the register availability table.

Loads from memory are handled in the following way. They are dispatched as any other instruction but the entry in the register-availability table of the output register is set to unknown. Instructions that depend on the load will be held in the Wait queue since the availability time of their operands is unknown. When the load performs the write-back stage, it will update the entry in the register-availability table with the current cycle and it will broadcast the destination register of the load and the current cycle to the Wait queue. This may wake up some instructions that use the value produced by the load. Similarly, these instructions will update the entry of its output register in the register-availability table and wake up other instructions in the Wait queue, and so on.

2.3.2 Deterministic Latency Scheme

The other alternative presented in the prescheduling family is the Deterministic Latency Scheme. In this case, the memory accesses are assumed to have a fixed latency (either hit or miss time in the first level cache). Instructions are all scheduled in the Issue queue according to the predicted latencies.

Nevertheless, the issue will be stalled when the instruction depending on the memory access has to issue and the memory access has not finished yet. Alternatively, there could be a queue where the dependant instructions are kept when they are not ready at the expected issue time.

Figure 9 shows a block diagram of the Deterministic Latency issue logic. This scheme consists of three main blocks. First, we have a table where for each physical register it contains the cycle when its value will be produced. This table is called the register-availability table (as in the Distance scheme).

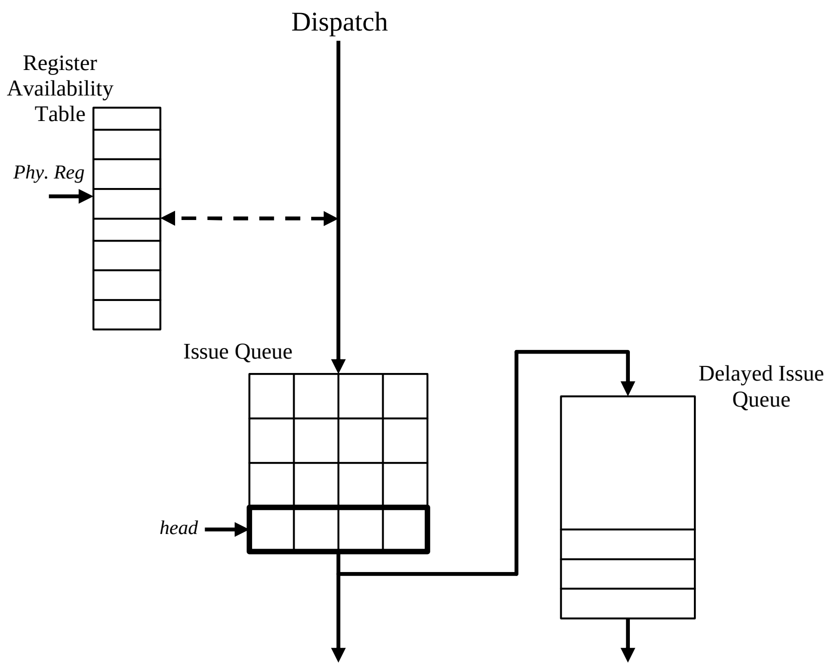


Figure 9: Deterministic Latency issue logic

The second block is the Delayed Issue queue, which holds the instructions that were scheduled to be issued too early, before their operands were ready. This queue has a complexity similar to that of a traditional instruction issue queue since every time an instruction finishes its execution, its physical destination register is broadcast to all entries of this queue. Any entry with a matching source operand takes note of its readiness and will be issued the next cycle that there is an available issue slot. The third block is the Issue queue, which has the same structure and functionality as described in the previous section for the Distance scheme.

Loads from memory are handled in the following way. They are dispatched as any other instruction and they are assumed to have the latency of a hit in the first level cache (alternatively, they could have the latency of a second level cache hit). Instructions that depend on the load read the output time and are scheduled according to it. Nevertheless, it can happen that an instruction depending on a memory access is not ready to execute when it is at the head of the issue queue since the memory access has not finished.

For example, an instruction depending on a load may assume that the memory value will be available 1 cycle after the load is issued. If the load misses in the first level cache, the data will not be ready at that time so the instruction will have to be taken apart (kept in the Delayed Issue queue) or the issue will have to be stalled. In this chapter, we study both alternatives and investigate the trade-off between the size of the delayed issue queue and performance. Note that since stores do not produce any output register the scheduling is not affected by the store latency.

Instructions are issued first from the Delayed queue, and then from the Issue queue. Any instruction that cannot be issued from the Issue queue head -either because its operands are not ready or because the instructions of the Delayed queue took its issue slot- will be kept in the Delayed queue. If the Delayed queue is full and there are some instructions in the Issue queue that should move into the Delayed queue, the issue will be stalled till the "delayed" instructions fit in the Delayed Issue queue.

The technique used for instruction squashing is similar to that presented for the N-Use scheme in Section 2.2.1. In this case, the tags will be held in the Issue queue. No work needs to be done in the Register-Availability Table since during the squashing the physical registers will be deallocated (and thus not referenced). Next time the physical register is assigned (as an output register of a new operation) the availability time will be set by the new instruction.

2.4 Performance Evaluation

2.4.1 Experimental Framework

We have used a cycle-based timing simulator based on the SimpleScalar tool set [10] for performance evaluation. We extended the simulator to include register renaming through a physical register file and the issue mechanisms described in this chapter. See Table 1 for the main architectural parameters of the machine. We used the programs from the Spec 95 suite. All the benchmarks were compiled with the Compaq-Alpha C compiler with the -O5 optimization flag. For each benchmark, 100 million instructions were run after skipping the first 100 million. Performance results are reported as the harmonic mean for the whole benchmark suite. For comparison purposes, an out-of-order and an in-order scheme have been also evaluated. The next subsections present performance figures for all these schemes.

Table 1: Machine parameters

Parameter	Configuration
Fetch Width	4 instructions
I-cache	64KB, 2-way set-associative. 32-byte lines, 1-cycle hit time, 6-cycle miss penalty.
Branch Predictor	Combined predictor of 1K entries with a Gshare with 64K 2-bit counters, 16 bit global history, and a bimodal predictor of 2K entries with 2-bit counters.
Decode/Rename width	4 instructions
Max. in-flight instructions (RUU)	64
Retire width	4 instructions
Functional units	3 intALU + 1 int mul/div3 fpALU + 1 fp mul/div
Issue mechanism	4 instructions Out-of-order Window based
D-cache L1	64KB, 2-way set-associative. 32-byte lines, 1-cycle hit time, 6-cycle miss penalty 3 R/W ports
I/D-cache L2	256 KB, 4-way set associative, 64-byte lines, 6-cycle hit time. 16 bytes bus bandwidth to main memory, 16 cycles first chunk, 2 cycles interchunk
Physical registers	96

2.4.2 N-Use Scheme

Figure 10 shows the IPC of the basic N-use mechanism (i.e. without an I-buffer) in comparison to an in-order issue and an out-of-order issue. We can see that the first-use scheme performs better than an in-order issue mechanism but significantly worse than an out-of-order issue scheme. Actually, what happens is that the out-of-order scheme can find instructions ready to execute further ahead in the code sequence than the N-use scheme since the latter stalls the dispatch much more frequently. The first-use scheme performs –in terms of IPC- 38% better than the in-order scheme but it slows down the out-of-order machine by a 40%. For bigger associativities of the N-use table, the performance is very close to that of an out-of-order mechanism since with this configuration the issue is stalled less frequently due to the bigger capacity of the N-use table.

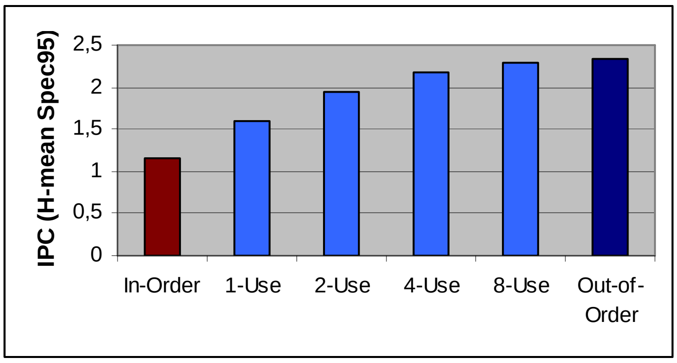


Figure 10: Performance of the N-use scheme (without I-buffer)

When the N-use scheme is extended with a small out-of-order I-buffer (see Figure 6) the performance of this scheme significantly increases. Figure 11 shows the evolution of the IPC when the size of the I-buffer varies. We can see that for sizes of the I-buffer of 8 elements or more, the N-use scheme performs almost at the same level as the out-of-order issue mechanism. When reducing the I-buffer size further, the overall effect depends on the associativity of the N-use table. The higher the associativity, the minor the effect of the I-buffer size is. For a 4-entry I-buffer, the performance degradation is very low whereas without a I-buffer, the degradation varies from 2.5% and 32% depending on the associativity level.

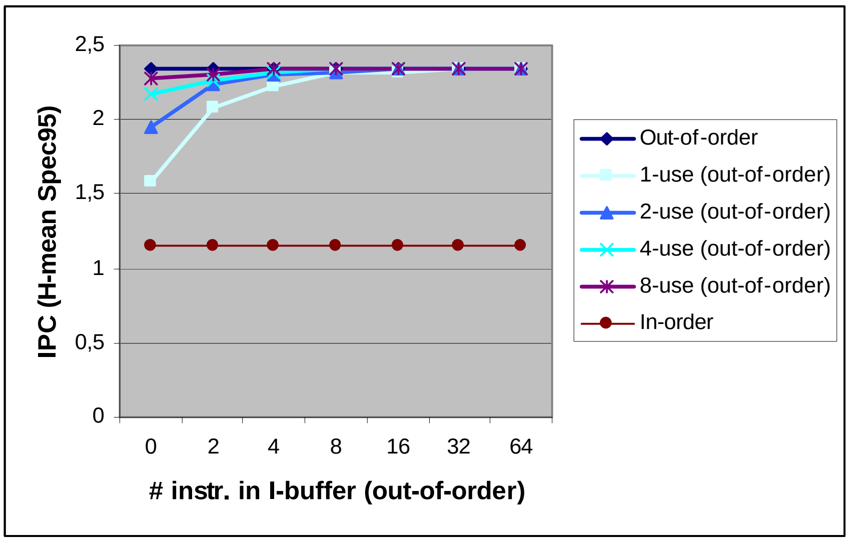


Figure 11: Evolution of the IPC for the N-use scheme for different sizes of an out-of-order I-buffer

In particular, the 2-use scheme with a 2-entry I-buffer achieves an IPC comparable (~4% slow-down) to that of an out-of-order scheme and it reduces the associative look-up from 64 to 2 entries (32 times smaller). This restricted associative search will certainly result in a shorter issue delay which may in turn influence the clock-cycle time.

Alternatively, we could get rid of any associative search logic by implementing an in-order issue for instructions in the I-buffer. The performance of this alternative is shown in Figure 12.

We can see in Figure 12 that this scheme implies a decrease in IPC with respect to the out-of-order I-buffer configuration (see Figure 11) for the first-use scheme whereas it is rather low for higher degrees of associativity. It is interesting to analyze the impact of the size of the in-order I-buffer on performance. A bigger I-buffer reduces the stalls in the dispatch. However, since instructions from the I-buffer are issued in order, once an instruction is placed on this buffer it must wait until all previous instructions have been issued. However, sometimes it is better to stall the dispatch for a few cycles and then issue the instruction to the N-use table, from where it can issue out of order. This trade-off explains why the IPC increases when the I-buffer size increases, but beyond a certain size (8 entries), the benefits of a larger I-buffer are more than offset by its drawbacks, which results in a slight decrease in performance. This effect is minimized by the associativity of the N-use table. For smaller values of N the effect is more visible. Overall, the performance of the N-use scheme is quite close to that of an out-of-order scheme for a small I-buffer and associativities higher than 1. For associativity 1 the performance is somewhat lower (about 12% lower on average) for any I-buffer size.

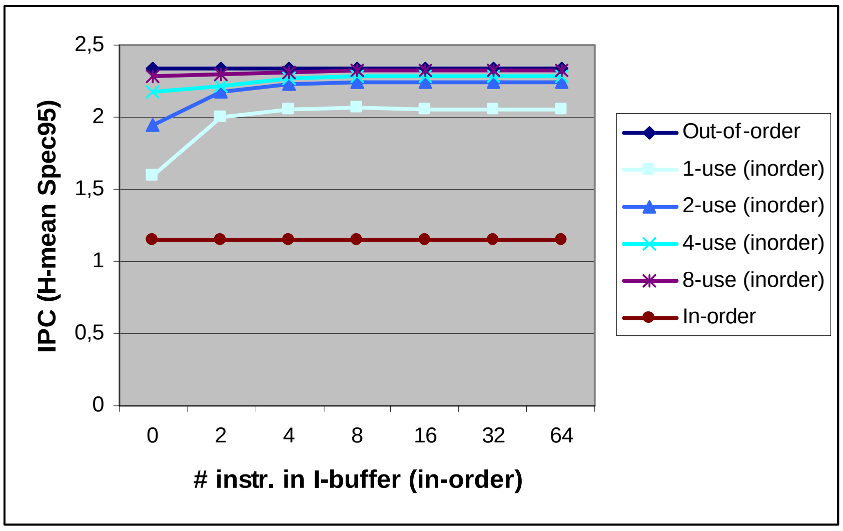


Figure 12: Performance of the N-use scheme for different sizes of an in-order I-buffer

2.4.3 Distance Scheme

Figure 13 shows the IPC of a basic implementation of the Distance scheme in comparison with an in-order issue approach and an out-of-order issue mechanism. This basic implementation does not require any associative search since it assumes a zero-entry Wait queue (see Figure 8). We can see that the new mechanism performs better than an in-order machine (85% speed-up) and not very far from an out-of-order machine (10% slow-down).

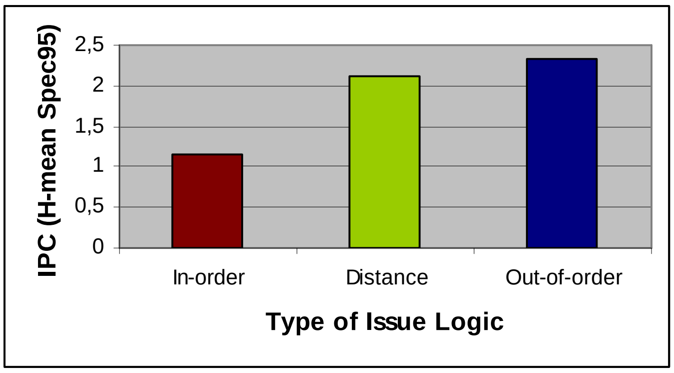


Figure 13: Performance of the basic Distance scheme (without Wait queue)

When a full implementation of the Distance scheme is considered, the IPC is significantly improved. Figure 14 shows the evolution of the IPC when the size of the Wait queue varies. We can see that from 4 to 64 elements in the associative part of the mechanism (Wait queue) the scheme performs almost at the same level (~4% slow-down) as the out-of-order approach.

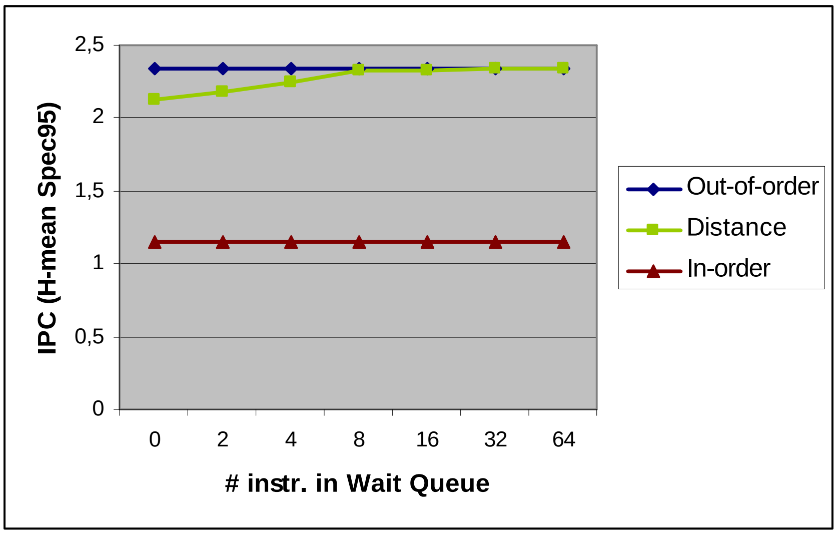


Figure 14: Performance of the Distance scheme for different Wait queue sizes

We have empirically determined that the maximum depth that the Issue queue of the Distance scheme requires in order not to cause any stall is 4 entries of 4 instructions each and, on average, it is around 2 entries (16 instructions to be issued in this or the following cycle) for the SpecInt95 benchmarks and a little bit larger (8 entries) for FP benchmarks in our experimental framework (see Section 2.4.1 for the details).

2.4.4 Deterministic Latency Scheme

Figure 15 shows the IPC of a basic implementation of the Deterministic Latency scheme when memory instructions are scheduled assuming a 1-cycle -DL1- and 6-cycle -DL6- memory access latency. An in-order issue approach and an out-of-order issue mechanism are also shown for comparison. This basic implementation does not require any associative search since it assumes a zero-entry Delayed Issue queue (see Figure 9). We can see that the DL mechanism performs much better than an in-order machine (65% and 84% speed-up for the DL1 and DL6, respectively) and not very far from an out-of-order machine (10% slow-down for the DL6).

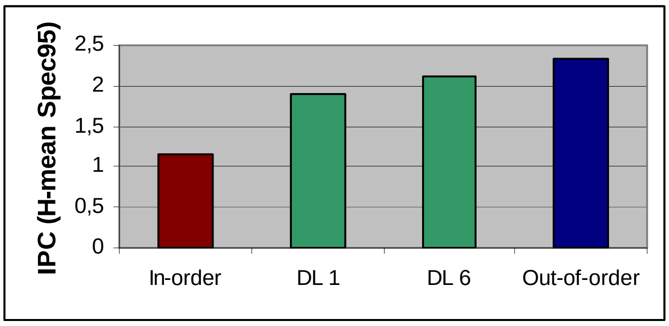


Figure 15: Performance of the basic Deterministic Latency scheme (without a Delayed Issue queue)

When a full implementation of the Deterministic Latency scheme is considered, the IPC is significantly improved. Figure 16 shows the evolution of the IPC when the size of the Delayed queue varies. The size of the queue hardly affects the performance of the DL6 scheme. Although the performance grows up when the size increases, there is a point where the extra cycles spent for every access (this scheme assumes that memory accesses take 6 cycles) limit the maximum performance achievable. As far as the DL1 is concerned, we can see that with 16 to 64 elements in the associative part of the mechanism (Delayed Issue queue) the scheme performs almost at the same level (~4% slow-down) as the out-of-order approach. For smaller sizes, the performance goes down significantly.

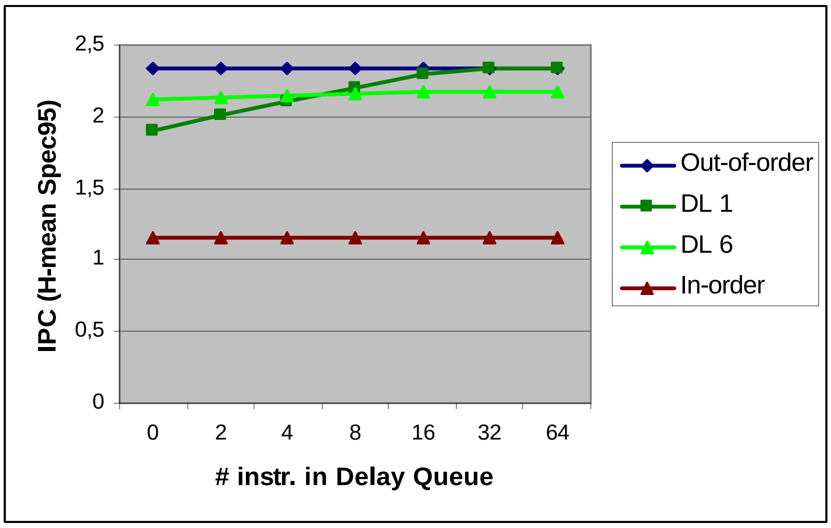


Figure 16: Performance of the Deterministic Latency scheme for different Delayed Issue queue sizes

We have empirically determined that the maximum depth that the Issue queue of the Deterministic Latency scheme requires in order not to cause any stall is 60 entries of 4 instructions each and, on average, it is around 18 entries for the Spec benchmarks for the 1-cycle memory access latency configuration (45 for the DL 6). Note also that for integer codes the size of the Issue queue is smaller due to the shorter latencies of the functional units.

2.4.5 N-Use vs. Latency

Figure 17 shows the performance of both the N-use and the Deterministic Latency schemes when varying the size of the associative hardware. We can see that when there is no associative buffer, the Deterministic Latency scheme and the 4-Use scheme perform at the same level. Besides if a small associative buffer (2 or 4 entries) is feasible, the N-use scheme performs better. And, in this case, the performance achieved is very close to that of an out-of-order mechanism.

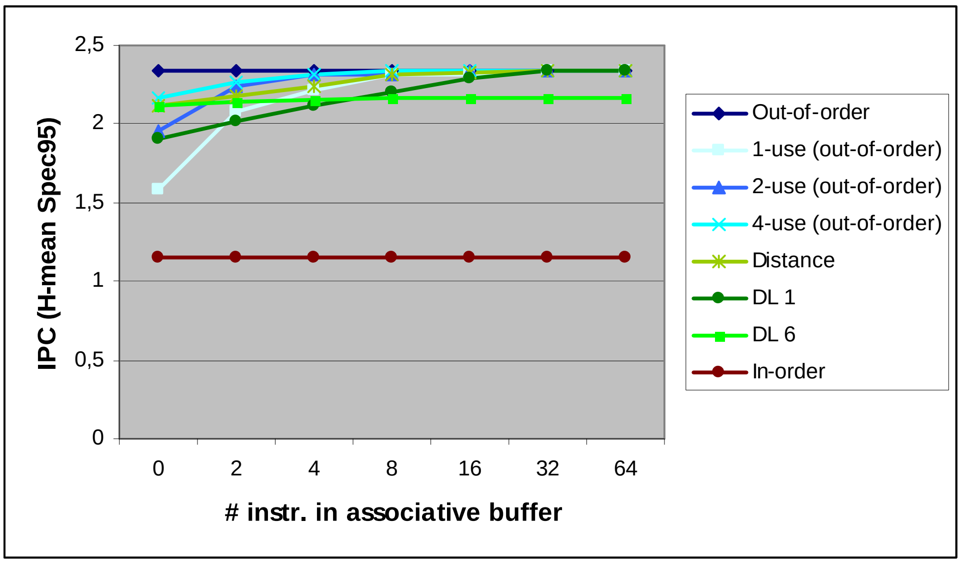


Figure 17: Performance of the three proposed schemes

Thus, we can conclude that if we can afford a small associative buffer (around 2-4 entries), the N-use scheme is very competitive in terms of IPC when compared to a fully out-of-order mechanism and it has the potential advantage of reducing the cycle time. If we want to completely avoid any associative search, both the Deterministic Latency scheme without the Delayed queue and the N-Use scheme with a moderate associativity in the N-use table with an in-order I-buffer have about the same IPC, which is significantly higher than that of an in-order scheme (~84%) and not far from the performance of an out-of-order mechanism (~10%).

2.4.6 Comparison with the FIFO-Based Scheme

In this Section, we will compare the proposed mechanisms to the FIFO-based scheme proposed by Palacharla *et al.*[54]. In Palacharla's scheme instructions are dispatched to several FIFO queues depending on the following facts:

- If the last instruction of a queue produces one of the registers of the instruction being dispatched, this instruction is sent to that queue.
- Otherwise it is sent to an empty FIFO queue. If there are no queues available, the dispatch is stalled until this instruction has an empty FIFO.

Figure 18 shows the performance of the FIFO scheme in comparison to the N-use scheme and the Deterministic Latency scheme. For the FIFO-based scheme, the X-axis corresponds to the number of FIFO queues (note that the zero position means zero entries in the extra buffer for the DL and N-use schemes and one single FIFO for the FIFO-based mechanism). The number of FIFO queues determines the degree of associativity of the issue logic since instructions of different queues are issued in any order whereas those in the same queue are issued in-order. The number of entries in the FIFO queues for each configuration is determined by the number of entries in the instruction window (64) divided by the number of queues. We can see that the performance of all the schemes is similar for a degree of associativity of 16 or more (except for the Deterministic Latency that assumes 6 cycles for the memory accesses). When reducing the associativity the FIFO scheme reduces dramatically its performance due to lack of empty FIFO queues when dispatching.

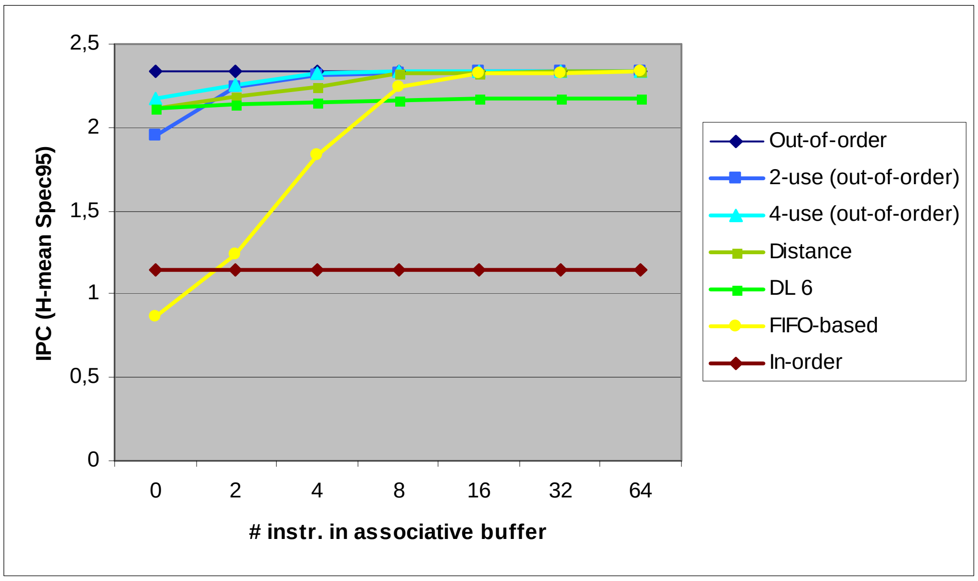


Figure 18: Performance of the FIFO scheme together with the N-use and the Deterministic Latency

In conclusion, the N-use scheme and the Deterministic Latency scheme have a much better performance than the FIFO-based scheme for low degrees of associativity.

2.4.7 Energy Savings

Energy consumption statistics have been taken from a modified version of the Wattch framework [8]. The tool was modified to implement the techniques presented in this chaper. Processor parameters –as used through section 2.4- can be found in Table 1 in page 19. Energy savings are above 8% for most of the configurations. Figure 19 shows the energy savings of a processor (total processor power) with the proposed mechanisms over an out-of-order architecture (base architecture). The configurations with smaller structures (the ones with a small out-of-order queue) can further reduce the energy spent and they go beyond the 10% savings mark.

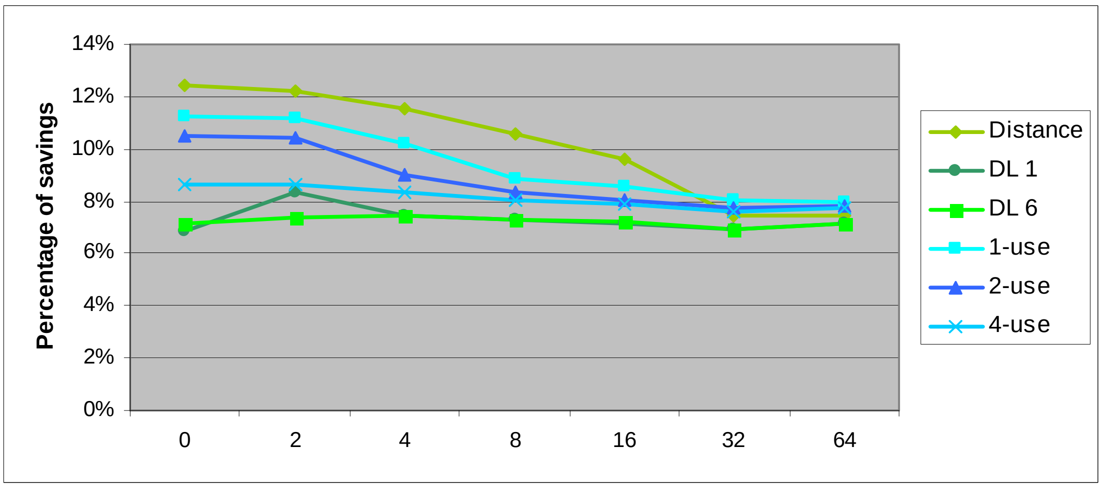


Figure 19: Evolution of the energy savings for the proposed mechanisms

Some of the schemes studied degrade performance. In this sense, energy-efficiency has been studied and it is shown in Figure 20 as the benefit on the energy-delay^2 product. Although all the mechanisms reduce the energy requirements (as it is shown in Figure 19), when considering energy-efficiency most configurations are more energy-efficient than the out-of-order scheme. Since performance is degraded in many of the schemes this turns out in energy-efficiency levels below that of an out-of-order architecture as it is the case for the configurations without extra buffer. Since the energy-delay^2 product gives more importance to delay than to energy reduction, the performance of the schemes is very important when considering its energy efficiency. In this sense, just the schemes that perform close to the out-of-order architecture (the ones that have big extra buffers) can be more efficient since their performance slow down is minimal and while keeping most of the energy-savings of these schemes.

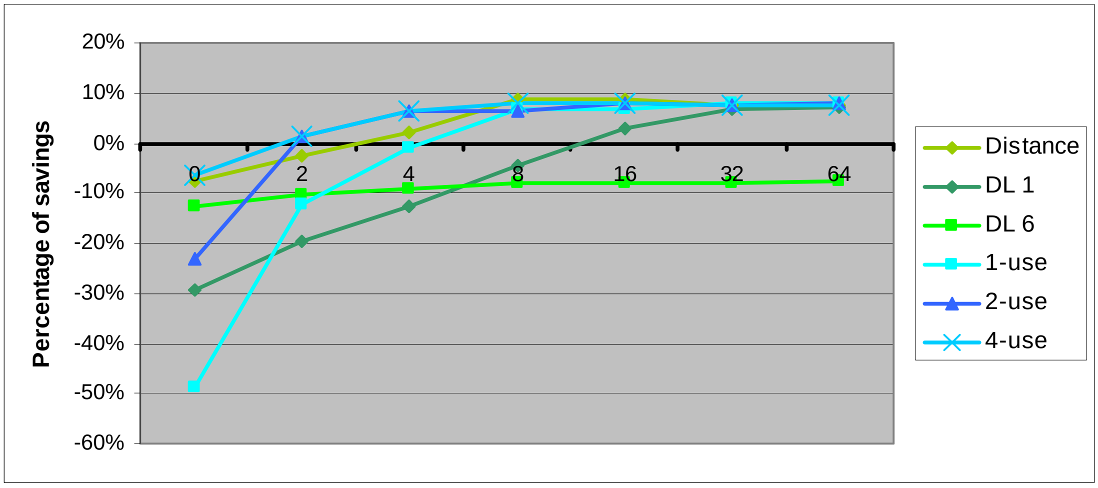


Figure 20: Evolution of the energy-delay^2 product for the proposed mechanisms

2.5 Related Work

S. Weiss and J.E. Smith [73] designed an issue logic similar to the basic first-use mechanism explained in Section 2.2.1. In their work, they did not implement an I-buffer and the first use check was done through a tag mechanism. As shown in Section 2.4.2 the mechanism suffers significant performance degradation when the I-buffer is not present.

S. Palacharla and J.E. Smith [54] presented an approach based on implementing the issue queue through several FIFOs so that just the heads of the FIFOs need to be considered for issuing. Section 2.4.6 compares this proposal to the ones in this thesis.

S. Öner and R. Gupta [52] proposed a mechanism which tried to chain the instructions according to the dependences among them. This scheme builds the data dependence graph dynamically and limits the number of instructions that each instruction can wake up. In their study they assumed that all the FUs latencies are known, which simplifies the issue logic.

A drastic approach to reduce the complexity of the issue logic is to use in-order issue, such as VLIW architectures [25]. In this case the scheduling is done at compile time. This is the approach that reduces most the issue logic complexity but, on the other hand, it is not as flexible as a dynamic scheme since not all the information is available at compile time (e.g. memory access latencies).

Michaud *et al.* [43] presented a prescheduling technique similar in concept to the proposed prescheduling techniques presented in this chapter. In their case, they did not work directly on the issue logic but on preparing (prescheduling) the instruction for that stage. A deeper pipeline was used and thus more complex sturctures could be used.

Henry *et al.* [31] presented several circuit-level techniques for reducing the complexity of the issue logic. Their approach is orthogonal to ours and thus both could be combined nicely.

More general approaches to reduce the complexity of the issue logic and, in general, of the microarchitecture are clustering and multithreading. These approaches try to reduce the complexity of the issue logic by partitioning it into several parts. The cluster approach [13][18][24][33][54] partitions the datapath whereas in the multithreading approach [66][73], each thread may have its own issue logic. Both architectural approaches are orthogonal with the research conducted in this work.

2.6 Conclusions

Out-of-order issue is a key component for high-performance in non-numeric applications. For instance, we have observed that for the SpecInt95, an out-of-order implementation achieves an IPC (instruction committed per cycle) that is about 2.5 times higher than that of a similar processor with an in-order issue scheme. This difference could be reduced by a compiler that was aware of the underlying in-order issue scheme and generated a more optimized code for this case, but we may still expect significant differences since the performance of instruction scheduling for non-numeric codes is significantly constrained by the small size of basic blocks and the large percentage of ambiguous memory references.

However, implementing a fully out-of-order issue scheme in the way that current microprocessors do, is very complex and energy-demanding, which poses significant constraints to its scalability. In this chapter we have proposed alternative implementations of an out-of-order issue scheme that are much simpler and retain most, if not all, the ability of the fully out-of-order mechanism to exploit instruction-level parallelism.

In particular, the N-use scheme does not require any associative search and provides a performance very close the out-of-order scheme for N equal to 4. The cost of this scheme is a table with as many entries as number of physical registers but this table does not require any associative search, i.e., it is indexed by the operand register identifiers). Other alternatives are the prescheduling schemes. Both the Distance scheme and the Deterministic Latency scheme provide a performance very close to the out-of-order processor and at the same time they constrain the associative searches to a small queue of 8-16 entries.

At the same time, the schemes proposed can be more energy-efficient than a conventional out-of-order issue logic while using smaller amounts of associative structures. These small structures reduce the impact on performance that the schemes introduce and while not adding significant extra power; can yield better results when considering both energy and performance.

"The most exciting phrase to hear in science, the one that heralds the most discoveries, is not 'Eureka!', but 'That's funny...'"
Isaac Asimov

Chapter 3

Value Compression Principles

In this chapter, we show that most of the values that flow through the pipeline do not need the maximum word-width to be represented. We then show how value compression can reduce the number of bits that flow through the pipeline. In the forthcoming chapters, value compression will be used to design ultra-low power architectures and to devise compile-time techniques to assist in reducing energy consumption and power dissipation.

3.1 Motivation

As the name suggests, value compression reduces the number of bits used for representing a given value. In this work, we focus on compression of non-floating point data; extensions for floating point is left for further research.

Value compression can be used in several structures that make up a processor's datapath. These include data and instruction caches, integer functional units, register files, and branch predictors. Figure 21 contains data that indicates the compressibility of data values read/written in the register file as SpecInt benchmarks are run on a 64-bit Compaq Alpha processor. This distribution shows a large potential for the value compression mechanisms because a large percentage of the values are narrow. For example, 44% have only one significant byte (values range from -128 to 127) and 52% fit in two bytes. The peak at 40bits is due to the memory addresses which are typically 5 bytes long in these binaries.

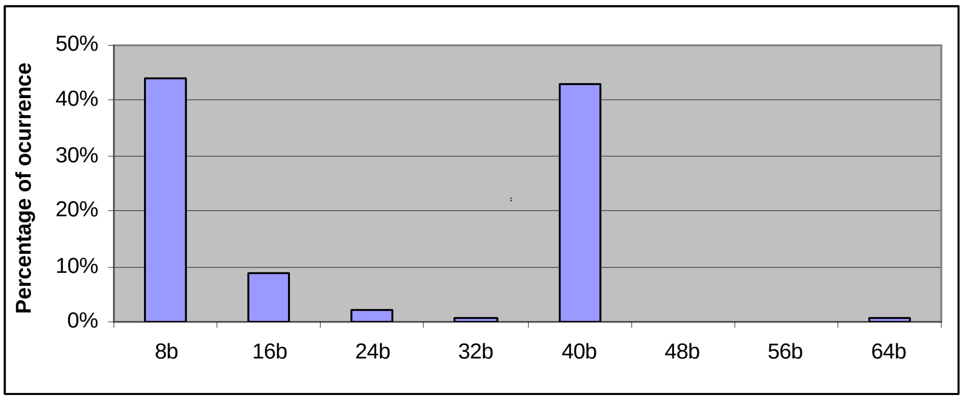


Figure 21: Register data size distribution for the SpecInt95 on a 64-bit microarchitecture

A good value compression method has to take advantage of this data distribution, and, at the same time, incur in a low overhead when compressing and decompressing. Thus, a good compression scheme should strike a balance between the compressibility of the values and the extra performance and energy costs of the mechanism.

Three basic methods of value compression have been considered in this work. The first, which we will call *size compression*, compresses values according to their size (i.e. number significant bytes in 2's complement notation) [7][39][51][52][60]. With size compression, one or more format bit(s) indicate the number of significant bytes. The second mechanism uses one format bit per byte to indicate whether the byte is zero or not [71]. This method, which we call *zero compression*, can take advantage of zero bytes in any position, not just in high order positions as with size compression. The last mechanism is the *significance compression* introduced in this chapter which is a superset of size-compression.

3.2 Significance Compression

The basic technique proposed for representing data is to tightly compress data bits that do not hold significant data. For example, a small two's complement integer (positive or negative) has only a few numerically significant low-order bits and a number of numerically insignificant higher order bits (all zeros or all ones).

In principle, one could consider significance at bit-level granularity, i.e. store and operate on exactly the numerically significant bits and no more. However, implementations are likely to be simpler and more efficient overall if a coarser granularity is used. Consequently, we primarily consider byte granularities and focus on the significant bytes rather than bits. Byte granularity is rather arbitrarily chosen, but it seems to be a good compromise of implementation complexity and activity savings. In general, one could consider non-power-of-two bit sequences and dividing words into sequences of different lengths (as shown in Chapter 5). Because the lowest order data byte is very often significant, we will always represent and operate on the low order byte. Then we will use a very small number of bits (2 or 3) to indicate the significance of the other 3 bytes (of a 32 bit word).

A simple encoding (which we named *size compression*) is to add two extra extension bits to encode the total number of bytes that are merely sign extensions. For example, the 32-bit number 00 00 00 04 (in hexadecimal) can be encoded as - - - 04 : 11. This is a mixed hexadecimal/binary notation that uses hexadecimal for significant (represented) bytes, a dash for the insignificant (non-represented) bytes, and a binary pattern after the colon for the values of the extension bits. In the above example, the only significant byte is 04 with three sign extension bytes, so the extension bits encode a binary three. This simple method also works for two’s complement negative numbers if it is assumed that the high order significant bit of the most significant data byte is extended. For example, the number FF FF F5 04 can be represented as - - F5 04: 10. I.e. it has two significant bytes, and the most significant bit of these two bytes is extended to fill out the full 32-bit number. This encoding has an overhead of two bits per 32-bit word (about 6 percent).

After inspecting commonly occurring data/address patterns, it is apparent that there are other, easily compressible values. In these cases there are some "internal" bytes that are all zeros or all ones, and these bytes are in a sense insignificant (although we abuse the meaning of "significance" somewhat). An important case occurs for memory addresses in upper memory. These addresses often have nonzero upper bits, nonzero lower bits, but zero bits in between. For example, the data segment base of our experimental framework (see section 5.2.1) is set at address 10 00 00 00, thus a variable may be located at address 10 00 00 09.

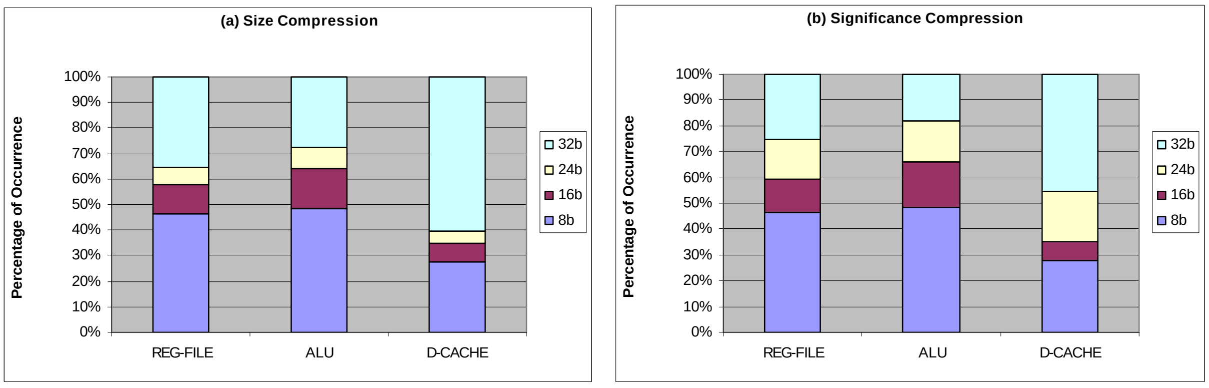
To handle these cases, we propose the *significant compression* scheme with three extension bits (approx. 9% overhead). In this scheme, the extension bits apply on a per-byte basis. Each extension bit corresponds to one of the upper three data bytes (as before, the least significant byte is always fully represented). If an extension bit is set to one, it indicates that the previous byte position is sign extended; if the extension bit is zero, it indicates the corresponding byte is significant. Consequently, the earlier example 10 00 00 09 is represented as 10 - - 09: 011 . As a more complex example, FF E7 00 04 is represented as - E7 - 04 : 101 The three-bit extension scheme allows for eight different patterns of significant/insignificant bytes (assuming the low order byte is always significant).

3.3 Value Compression for Ultra-Low Power Microarchitectures

Value compression can be applied to any microarchitecture, in this thesis we have made a separation between ultra-low power microarchitectures and high-performance ones. In this section we analyze the potential of the value compression methods for the ultra-low power segment. In this scenario, the typical applications run are media processing ones. Thus, the comparison will be done using the Mediabench benchmarks [37] and a 32-bit microarchitecture.

A simple, trace driven study was run to determine the relative frequency of occurrence of each value size. The benchmarks were run under an extended version of the *sim-fast* simulator of the Simplescalar framework [10]. Figure 22 shows the distribution of the run-time values manipulated by the register file, the ALU and the data cache. For the three compression schemes, the structure that has narrower values is the ALU since in many cases immediate values are used. On the other hand, the data cache is the structure that has wider data. The data cache has wider data since each memory instruction sends (and sometimes loads and stores) addresses and both address and data values are considered in these figures. All three compression mechanisms perform in a similar way in each structure. On one hand, significance compression achieves the smallest number of wide (32-bit) values for the register file and the ALU. On the other hand, zero compression reduces further the number of 32-bit values for the data cache. This may indicate that many addresses contain zero bytes that are not a sign extension of previous ones (otherwise significance compression would be as effective as zero compression). It is less probable that it may be due to data values since these are also considered in the register file and, in this structure, significance compression achieves better performance.

Overall, the average size for any value manipulated by the register file, the ALU and the data-cache is showed in Figure 23. The *overall average size* column is the arithmetic mean of all the accesses to the structures just mentioned (not the average of the previous columns). Overall, significance compression and zero compression achieve a similar compression rate on average. Nevertheless, significance compression needs two less extra-bits per word to indicate the format. Thus, significance compression might be a slightly better value compression method in these 32-bit architectures.



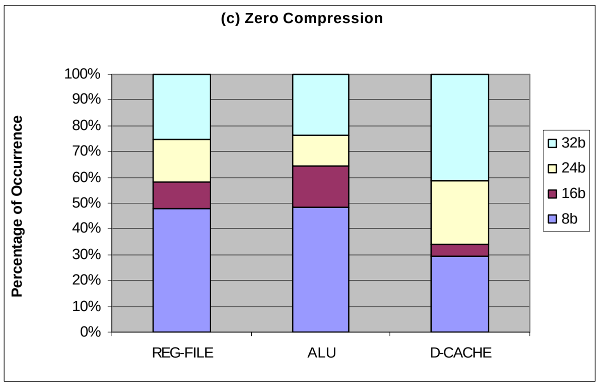


Figure 22: Distribution of run-time values for the several pipeline structures

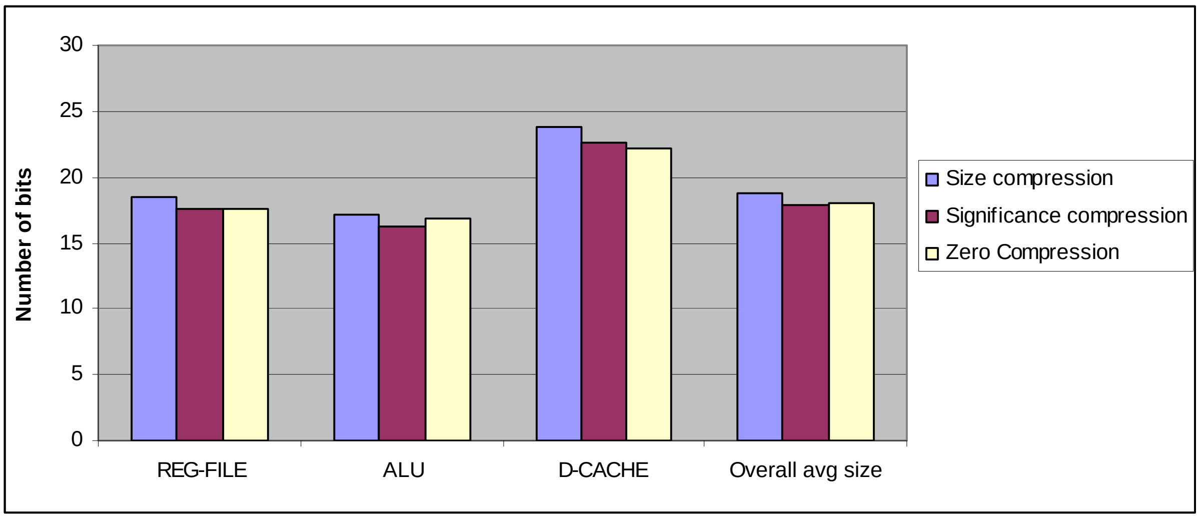


Figure 23: Average operand size for several pipeline structures

The exploration of value compression for ultra-low power microarchitectures continues in next chapter where several pipeline organizations are presented that take into account the compressability of the values by using significance compression.

3.4 Value Compression for High-Performance Microarchitectures

The previous section has showed that value compression can reduce the average number of bits needed for the values manipulated through the pipeline. In this section we analyze the potential of the same value compression methods for high-performance microarchitectures. In this scenario, the typical applications run are modeled through Spec Integer 95 and a 64-bit microarchitecture (i.e. Compaq Alpha). The study was conducted through a trace driven simulation to determine the relative frequency of occurrence of each value size using the Simplescalar framework [10].

Figure 24 shows the distribution of the run-time values for the register file, the ALU and the data-cache (as in the previous section). In any of the three compression methods, most of the data is either 8-bit wide or 40-bit wide (i.e. memory addresses). It is interesting to see that in the case of the zero compression, the number of 64-bit values is significantly higher than in the other cases. This is due to the large percentage of negative values (values that have the upper bits set to 1). Both size and significance compression can compress these negative values effectively but zero compression. Overall, the average access size for each of these structures is shown in Figure 24. In this case, significance and size compression perform better than zero compression.

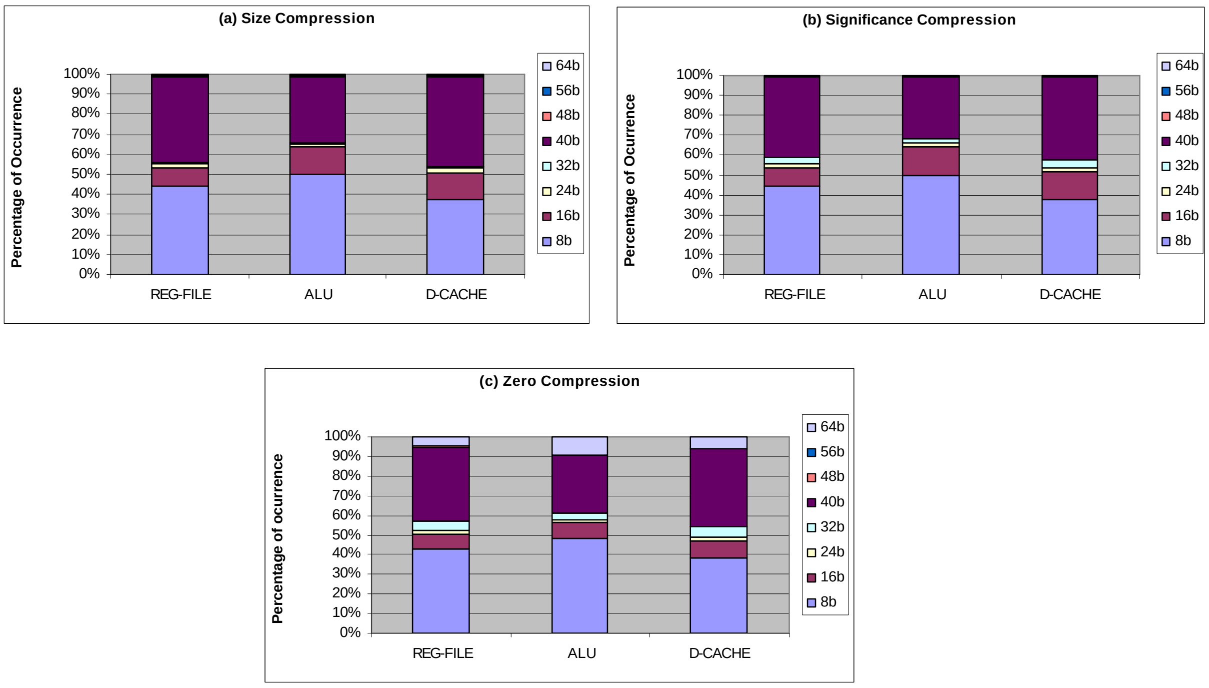


Figure 24: Distribution of run-time values for the several pipeline structures with size compression

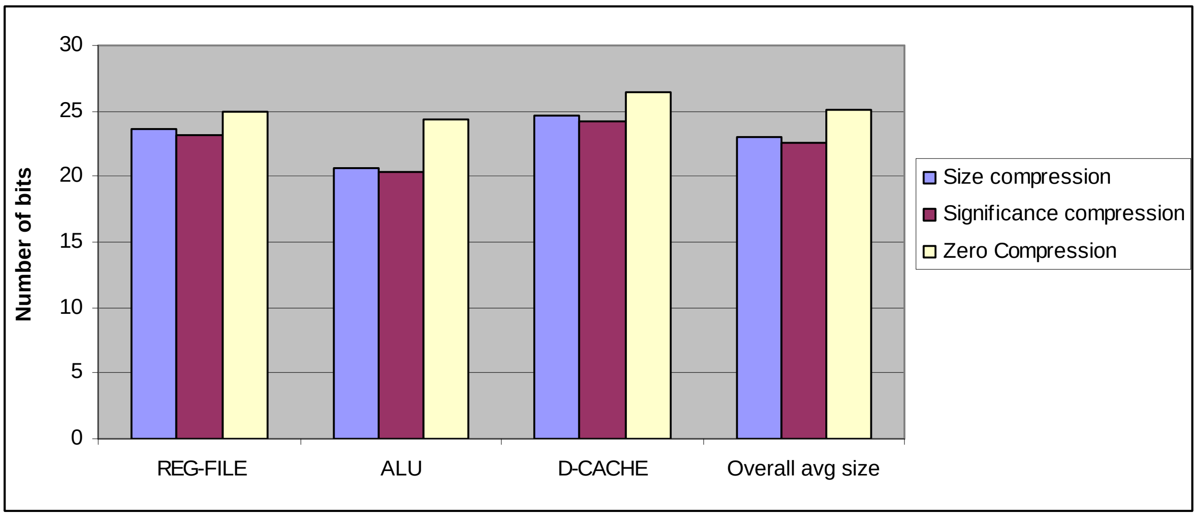


Figure 25: Average operand size for several pipeline structures

In Chapter 5, several hardware and software techniques are proposed for high-performance microarchitectures to take advantage of the compressibility of the values (as Figure 25 shows, although 64 bits are reserved for each value manipulated in the datapath, just around 23 bits are necessary –on average).

"Necessity is the mother of invention"
Plato

Chapter 4

Ultra-Low Power Microarchitectures

Power requirements are becoming an increasing headache among processor designers. In the ultra-low-power architectures, simple pipelines are implemented. We show how a significance compression method is integrated into a 5-stage pipeline, with the extension bits flowing down the pipeline to enable pipeline operations only for the significant bytes. Consequently register, logic, and cache activity (and dynamic power) are substantially reduced.

4.1 Introduction

There are many microprocessor applications, typically battery-powered embedded applications, where energy consumption is the most critical design constraint. In these applications, where performance is less of a concern, relatively simple RISC-like pipelines are often used [45][57]. As shown in Chapter 1, in current CMOS technology, most energy consumption occurs when transistor switching or memory access activity takes place. Consequently, an important energy conservation technique is to "gate off" portions of logic and memory that are not actively being used.

Recently [7] it was proposed that certain operand values could be used to gate off portions of execution units rather than basing logic gating decisions entirely on operation types. In particular, arithmetic involving short-precision operands only needs to be performed on the (relatively few) numerically significant bits. Operations involving insignificant bits (typically leading zeros or ones) can either be avoided or can be computed more simply than with the normal functional unit.

We generalize the notion of operand gating to all stages of the pipeline. The key principle is the use of a small number of extension bits appended to all data and instructions residing in the caches, registers, and functional units. In Figure 26, the extension bits are shown along the bottom of a basic pipeline. These bits correspond to portions of the datapath, and they flow through the pipeline to gate-off unneeded energy-consuming activity at each stage. New extension bit values are generated only when there is a cache line filled from main memory (although they could also be maintained in memory) and when new data values are produced via the ALU. The points where extension bits are generated are indicated in Figure 26 by circled "G"s.

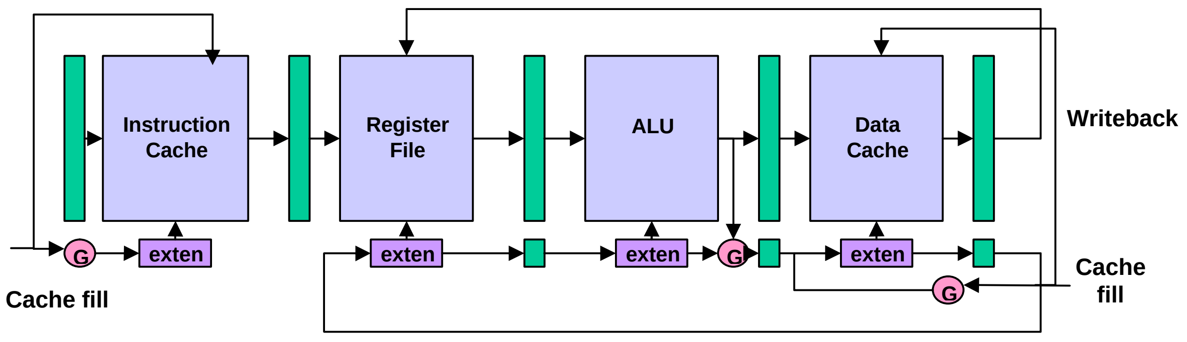


Figure 26: Basic pipeline

For the instruction caches, extension bits allow a simple form of compression targeted at reducing instruction fetch activity, rather than reducing the number of bits in the program's footprint. For other datapath elements, they enable a form of compression where memory structures actively load and store only useful (significant) operand bytes. For arithmetic and logical operations, the extension bits enable operand gating techniques similar to those proposed in [7].

Given that only significant bytes require datapath operations and storage, pipeline hardware can be simplified by using byte-serial implementations, where the datapath width may be as narrow as one byte, and a pipeline stage is used repeatedly for the required number of significant bytes. Although there are many alternative implementations with different degrees of parallelism, in all of them there is some serialization in the pipeline. In particular, low-order byte(s) and extension bits are first accessed and/or operated on; then additional bytes may be accessed and/or operated on if necessary. We describe and evaluate several pipeline implementations of this type.

4.2 Techniques for Reducing Activity Levels

In this section, we develop methods for reducing memory and logic activity for each pipeline stage. Because activity in the simple pipeline depends primarily on the data values and instructions, we first undertake a trace-driven study to determine the required activity for each of the major pipeline operations. Then, in later sections, we propose and study pipelined implementations that come close to achieving the minimum "required" activity levels.

This work is based on a simple 5-stage pipeline with in-order issue as is often used for low power embedded applications. We consider the 32-bit MIPS instruction set architecture (ISA) and focus on integer instructions and benchmarks -commonly used in the very low power domain.

4.2.1 PC Increment

Incrementing the PC is one of the operations at the very beginning of the pipeline. When incrementing the program counter, we do not literally append extension bits to the operands. One of the operands is always +1 (the PC is word resolution), so it is known to have only one significant bit. The PC, on the other hand, is held to full 30-bit precision (the two least significant bits are zero). The PC increment is performed byte-serially to reduce activity. In particular, we first increment only the low order byte. If a carry out is produced, the next byte is incremented on the next cycle, etc. If a carry out is not produced at any stage, no additional byte additions need to be done.

This method very often saves adder and PC latching activity for higher order bytes, but it can lead to some performance loss in the uncommon cases when there is a carry beyond the low order byte, and instruction fetch is temporarily stalled while additional byte additions are performed. A brief analysis sheds some light on this tradeoff. In general, one can consider a block-serial implementation where the block size is not necessarily a byte. The size of the block determines the performance and the activity savings. Performance is maximized by the biggest block size (i.e. 30 bits), but the activity savings are null. On the other hand, a smaller block may have a slightly lower performance but may produce significant activity savings.

If the block size is *N* bits, and we assume that at a random point in time all instruction addresses have the same probability, then we can calculate the probability that *i* stages (each of size *N*) are required to compute PC+1. We refer to this probability as *P(i,N)*. The value of *P(i,N)* for a 30 bit PC can be calculated as:

$$\begin{cases} P(i,N) = \dfrac{2^{N-1}}{2^{i*N}} & i = 1..\left\lceil \dfrac{30}{N} \right\rceil - 1 \\ P(\left\lceil \dfrac{30}{N} \right\rceil, N) = 2^{\left(\left\lceil \frac{30}{N} \right\rceil - 1\right)*N} & i = \left\lceil \dfrac{30}{N} \right\rceil \end{cases}$$

We can then compute the required activity (measured on average number of bits operated on) as

$$Activity(N) = \sum_{i=1}^{\left\lceil \frac{30}{N} \right\rceil} i * N * P(i,N)$$

And we can compute the average number of cycles to compute a PC as

$$Latency(N) = \sum_{i=1}^{\left\lceil \frac{30}{N} \right\rceil} i * P(i,N)$$

Table 2 shows the activity and latency statistics for values of *N* ranging from 1 to 8. Higher values of *N* are not interesting because they hardly improve performance, and activity increases significantly. Minimum activity is achieved for *N*=1, but this incurs in a 1-cycle penalty per instruction on average (as expected). *N*=5 may be a good trade-off because activity is reduced by 83%, and performance is degraded by just 3%. Finally, *N*=8 provides negligible degradation in performance with an activity reduction of 73.2%. In section 4.3, we validate these analytical figures with an empirical analysis. As indicated above, we assume a block size of 8 bits for the PC increment.

Table 2: Activity and performance estimates for PC updating

number of bits per block	Activity	Performance
1	2.00	2.00
2	2,67	1.33
3	3.43	1.14
4	4.27	1.07
5	5.16	1.03
6	6.10	1.02
7	7.06	1.01
8	8.03	1.00

4.2.2 Instruction Cache

To save instruction cache activity, instruction words are stored in a permuted form. The goal is to reduce the number of instruction bytes that have to be read, written, and latched. This objective is different from the more common instruction compression techniques [34][36][68][69][70] that attempt to store more instructions in a given amount of memory. In our case, each instruction is still allocated a full word in the instruction cache. However, not all bits have to be read/written/latched each time an instruction is placed in the cache or is fetched.

Permutation methods of this type are likely to be specific to the ISA, and we consider methods that work well for the MIPS ISA. While the exact methods may not extend entirely to other ISAs, similar methods are likely to be applicable – at least for RISC ISAs. Hence, the method given here is an example of the basic principle: instruction words can be modified (other than through classical compression) and extended with extension bits to reduce cache activity.

Although we considered a number of methods, two basic schemes seem to work well for the MIPS ISA and probably provide a significant majority of the benefit that can be achieved. First of all, we observe that the MIPS ISA very often uses one of two formats[1] [58]:

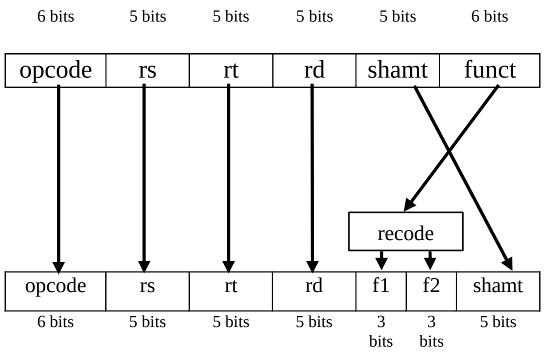
- R-format: A 6-bit opcode, three 5-bit register fields, a shift amount field, and a 6-bit function code.
- I-format: A 6-bit opcode, two 5-bit register fields, and a 16-bit immediate value.

In the R-format, the number of significant instruction bits can frequently be reduced to three bytes by recoding the six-bit function field so that the most common eight cases use three bits of the field with zeros in the other three bits. For these eight common cases, only three instruction bytes must be fetched and latched. In the other less common cases, all four instruction bytes must be fetched. Shifts that use the

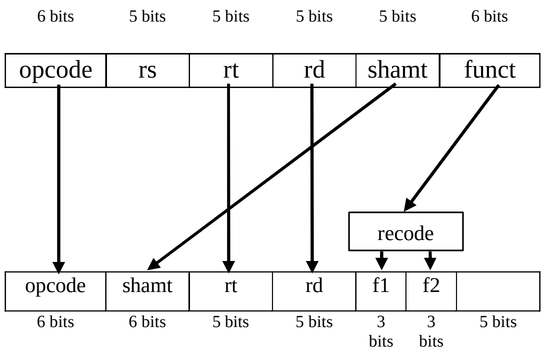
[1] There is a third format (J-format), but it only accounts for 2.2% of the executed instructions in the Mediabench.

shift amount field do not use the first register field (rs), so the fields can be permuted by moving the shift amount (shamt) into what is normally the rs field.

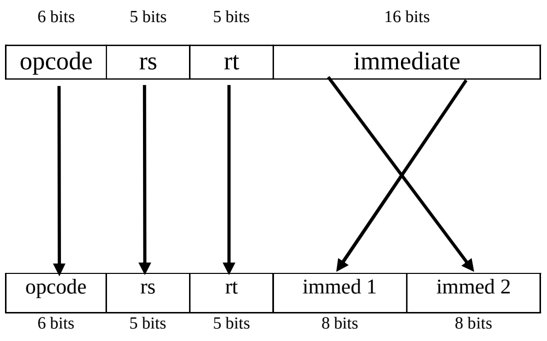
The permutation for R-format consists of shuffling bits in a minor way and re-encoding the function bits. Figure 27a and Figure 27b show the permutations for the R-format instructions. The function field is split into two 3-bit fields, f1 and f2, as noted above. To determine which function re-encoding to use, we first traced the Mediabench benchmarks and counted the dynamic frequency of each of the function codes. The results are in Table 3. Consequently, the most common eight function codes are recoded to 6-bit encodings, where the last three bits are all zeros (and do not have to be fetched). All the other function code patterns are mapped to the remaining six bit patterns. From the table we see that 86.7 % of all the R-format instructions require three bytes when modified in this manner.



a) First permutation for R-format instructions



b) Second permutation for R-format instructions



c) Permutation for I-format instructions

Figure 27: Permutations for the different instruction formats

For the I-format, we simply note that often eight or fewer immediate bits are actually significant, and in these cases three instruction bytes are again adequate. Figure 27c shows the permutation for the I-format instructions. For I-format instructions we also traced the benchmarks and determined the sizes of the immediate values. It was found that 59.1% of all instructions use immediate values and 80% of these immediates require only eight bits.

Although there are a few cases where it can be done, we do not attempt to reduce the number of fetched instruction bytes to fewer than three. Consequently, we add a single "extension" bit to the instruction word portion of the instruction cache. This bit indicates whether three or four bytes should be fetched and latched. Note that only one bit is used and it serves multiple purposes depending on the actual 6-bit opcode. For typical R-format opcodes it indicates that the low order three function bits (field f2) are zeros. For the shift amount R-format opcodes, it also moves the shamt field, and for I-format opcodes it indicates an 8-bit immediate.

Overall, a total of 36.85% of instructions are R-format that use the function field, 4.05% are R-format but the function field is not used, 56.9% are I-format and 2.2% are J-format. Combining this with the immediate and function code frequency statistics, the average number of bytes fetched and latched per instruction is 3.17 bytes (3.29 if we include the extension bit). This represents a savings of about 20% (at an overhead of 3% for the extra bit per word). There is also additional overhead during instruction cache fill for permuting/modifying the instruction bits, but this is a relatively small amount of additional activity, assuming a reasonable instruction cache miss rate.

Table 3: Dynamic frequency of the function codes

Opcode	Percentage	Cumulative
ADDU	36.01	36.01
SLL	16.20	52.21
SRA	9.09	61.30
SLT	8.19	69.49
SUBU	8.17	77.66
SLTU	3.28	80.94
XOR	3.07	84.01
MFLO	2.74	86.75
Other	2.47 .. 0.04	100

Finally, note that the order of the rearranged instruction bytes is chosen so that the bytes needed earlier in the pipeline are toward the most significant end. This enables better performance for implementations (to be given later) that read instruction bytes serially. For example, after an implementation fetches the first two bytes, there is enough information to perform the initial opcode decode and register read operations. The other bytes give the immediate bits, a result register field, and/or ALU function bits that are not needed until later in the pipeline.

4.2.3 Register File Access

For the register file, extension bits as described in Section 3.1 are used. When the register file is accessed, first the low order data byte and the extension bits are read. Depending on the values of the extension bits, additional register bytes may be read during subsequent clock cycle(s). In a study of the Mediabench benchmarks described below, we determined that the extension bits result in large register file activity savings. On average, the number of bits that are read is reduced by 47%.

4.2.4 ALU Operations

ALU operations are performed using only the numerically significant register bytes and the extension bits as input operands. The ALU produces significant result bytes as well as the extension bits that go with them.

ALU operations are performed in a byte-serial fashion. Because additions/subtractions, memory instructions, and branches all require an addition, and they collectively account for 70.7% of the executed instructions in our benchmarks, this operation is the most critical one to be implemented efficiently. For each byte position, there are three major cases, depending on which of the operands have significant byte(s) in the position being added.

- Case 1: Both bytes are significant. In this case, the byte addition must be performed.
- Case 2: Only one of the operands has a significant byte. If the non-significant byte is zeros (ones) and the carry-in from the preceding byte is zero (one), the result btye will be equal to the significant byte. If the non-signficant byte is zeros (ones) and the carry-in is one (zero), the result byte is the significant byte plus one (minus one). In all these cases one could use simplified logic; however, we do not include these potential optimizations in activity statistics.
- Case 3: Neither of the operands has a significant byte in the position being added. Consider the addition of two bytes, $C_i=A_i+B_i$, where A_i and B_i are both sign extensions of their preceding bytes, A_{i-1} and B_{i-1}. There is a general rule with some exceptions. The general rule is that the result byte C_i is not significant, and the result is computed simply by setting the extension bits of the result because C_i will also be a sign extension of C_{i-1}. In the exceptional cases, the ALU must generate a full byte value. Table 4 lists all exceptions to the general rule.

Table 4: Cases in which byte C_i has to be generated

Values of A_{i-1} and B_{i-1} (the order is not significant)	**Extra conditions**
00xxxxxx 01xxxxxx	5th bit produces carry
01xxxxxx 01xxxxxx	-
11xxxxxx 10xxxxxx	5th bit produces carry
10xxxxxx 10xxxxxx	-
00xxxxxx 11xxxxxx	5th bit produces carry
01xxxxxx 10xxxxxx	5th bit produces carry

To understand the exceptions to the general rule of Case 3, consider the example where A_{i-1} = 00000001, and B_{i-1}=01111111; A_i and B_i are both sign extensions (i.e. they are equal to zero). Then the addition of A_i+B_i will obviously be zero, but because byte C_{i-1} has a one in its most significant bit, C_i is not the sign extension of C_{i-1}. In this case, the processor has to generate the full byte value, although the addition is not actually necessary.

Finally, note that in some cases a result byte may not be significant although the two source operand bytes are significant (e.g. 3 + -3 = 0). To handle these cases, there is simple logic that examines each result byte and generates extension bits accordingly. This logic basically checks whether all bits of a byte and the most significant bit of the previous byte have the same value. It then sets the extension bits for the result accordingly.

Another common ALU operation that can be optimized is the comparison, which represents the 6.9% of the instructions in our benchmarks. In this case, the normal byte processing order can be reversed, with the computation starting at the most significant byte and finishing with the least significant one. However, as soon as the two compared bytes are different, no more bytes must be computed, even for comparisons of the type greater-than or less-than.

This reversal of access order can be implemented with different levels of complexity depending on the particular processor design. For instance, for the byte-serial implementation described in section 4.5, the reversal is easily implemented since this design has a single byte-wide register file and a byte-wide ALU. In other cases such as the byte-parallel skewed implementation described in section 4.7, the order reversal is more complex and may require an additional register port for avoiding structural hazards.

Finally, bit-wise logical operations, which represent 4.2% of the instructions in our benchmarks, can also be byte-pipelined. In this case, whenever two bytes are sign extensions, the result will also be a sign extension. Note that other optimizations are feasible when just one of the operands is a sign extension, but we have not considered them. For instance, A *AND* 0 = 0, A *AND* -1 = A, etc. As shown below in Section 4.3, extension bits result in an average reduction of the ALU activity of 33% for Mediabench.

4.2.5 Data Cache Operations

The data cache holds data in a manner similar to the register file. That is, extension bits are appended to each data word and only the bytes containing significant data are read and written. The address bytes may also be formed sequentially, beginning with the low order byte. This means that the cache index will be computed before the tag bits, and that the tag bits may be formed as part of multiple byte additions.

Consequently, the tag comparison may be done in two sections as the tag bits become available. If the lower order tag bits do not match the cache tags, then an "early miss" can be signaled, and the higher order address and cache tag bits do not have to be formed and compared, resulting in reduced activity. However, because the miss rate is relatively low (2.7% on average), the activity saving is insignificant.

There is similar activity for cache writes; the extension bits for the store data are read from the register file and written alongside the data. For cache fills, the extension bits must be generated as data is brought from memory (although the extension bit concept could also be maintained in main memory). We show below in Section 4.3 that the above techniques reduce the activity on the data cache by 31% for the data array and 1% for the tag array.

4.2.6 Register Write Back

During the register write-back stage, only bytes holding significant values have to be written into the register file. The extension bits also have to be stored. For ALU data, the bits are generated as described above in Section 3.1. For memory data, the extension bits read from the data cache are used. We show below in Section 4.3 that extension bits result in an average reduction of 42% in register file write activity.

4.2.7 Pipeline Latches

Significant energy is consumed in pipeline latches [70]. The extension bits are used for gating the pipeline latches in the normal way [41][46]. Only the PC bytes that change require latch activity. Based on extension bits, only significant register, ALU and cache bits need to be latched. Hence, activity savings in the datapath elements is reflected directly in activity savings in the pipeline latches immediately following the datapath elements.

Latch activity depends on the particular implementation. The lowest latch activity is achieved by the implementations with fewer pipe stages. This is the case for instance of the byte-serial implementation described in section 4.5. In this case, we show in section 4.3 that the latch activity can be reduced on average by 42%.

4.3 Activity Reduction

To determine the activity savings for the techniques described above, we performed a trace driven simulation of the Mediabench [37]. Only byte activity indicated by the extension bits was performed. Table 5 provides the overall results for byte granularity and for comparison, Table 6 contains average results for halfword granularity significance compression. The tables show percent activity savings.

The byte-serial PC increment operation saves 73% activity, because the great majority of the time, only the least significant byte is changed, as predicted by the analysis in Section 4.2.1. I-cache activity saving is 18%, and is quite uniform across all benchmarks. On average 47% of the Register read activity is saved, with individual benchmarks saving from 34% to 72%. ALU activity saving averages 33% (ranging from 15% to 68%) and data cache activity saves an average of 30% (ranging from 1% to 57%). The data cache activity is measured for data fills, reads and writes. The average saving on the data bank is 31% (ranging from 1% to 57%) whereas the saving for the tag bank is negligible. Register writeback saving is on average 42% (ranging from 30% to 69%). Finally, for implementations where the number of stages is not increased beyond the basic 5-stage pipeline, the latch activity is reduced by 42% on average and between 30% and 67% for individual benchmarks.

Table 5: Activity reduction (%) for datapath operations (8 bit)

Benchmark	Fetch	RF read	RF writes	ALU	D-cache data	D-cache tag	PC increment	Latches
cjpeg	17.25	45.95	41.72	30.26	39.61	0.15	73.33	40.43
djpeg	19.43	41.07	37.10	22.74	39.85	0.12	73.33	36.63
epicdec	19.11	45.05	35.21	30.03	20.33	0.06	73.33	39.73
epicenc	20.55	42.45	46.65	41.96	0.92	0.01	73.33	46.73
g721dec	19.32	50.51	47.27	39.60	43.86	1.94	73.33	45.38
g721enc	19.30	50.66	46.91	39.68	44.51	0.98	73.33	45.41
gs	13.59	47.73	47.21	35.52	39.88	1.30	73.33	42.44
gsmdec	17.39	45.69	40.26	33.14	38.10	0.23	73.33	41.03
gsmenc	19.02	45.13	35.35	31.69	12.95	0.69	73.33	39.92
mesamipmap	16.16	37.49	30.52	19.35	12.36	3.55	73.33	32.91
mesaosdemo	15.60	33.73	26.94	15.19	19.00	0.51	73.33	30.21
mesatexgen	16.76	40.48	36.69	23.18	12.30	1.91	73.33	35.62
mpeg2dec	16.95	43.74	38.88	28.33	36.14	0.38	73.33	38.87
mpeg2enc	20.86	47.01	48.67	32.17	30.19	0.03	73.33	42.61
pgpdec	19.01	38.85	29.75	20.49	25.04	2.79	73.19	34.33
pgpenc	18.05	39.68	34.24	25.26	30.59	0.02	73.31	35.98
rasta	16.56	43.91	38.27	27.23	12.15	2.76	73.33	38.02
rawcaudio	20.05	71.73	69.20	67.61	57.37	0.00	73.33	67.39
rawdaudio	20.05	71.73	69.20	67.61	57.37	0.00	73.33	65.41
AVG	18.16	46.98	42.11	33.21	30.66	0.92	73.33	42.06

Table 6: Activity reduction (%) datapath operations (16 bit)

Benchmark	Fetch	RF read	RF writes	ALU	D-cache data	D-cache tag	PC increment	Latches
AVG	18.16	35.85	30.31	22.11	23.38	0	46.67	34.93

The 16-bit serial savings remain substantial (Table 6), but are somewhat less than the byte serial activity savings, as expected. The primary advantage of the 16-bit granularity is in performance, as will be shown in short.

Holding and maintaining the extension bits adds an overhead of 9% when three bits are used, and the PC increment and instruction fetch stages have much less overhead.

The bottom line is that the net overall activity savings (and therefore the overall energy savings) can be substantial. Major savings are possible in each of the pipeline stages. Finally, note that these results are for a 32-bit architecture; if a 64-bit ISA were to be used (as in [7]), the savings will likely be much greater.

4.4 Experimental Framework

We developed a simulator for several proposed pipeline implementations using some components of the SimpleScalar toolset, primarily the instruction interpreter and the TLB and cache simulators. In all cases we assumed an in-order issue processor, with the following microarchitecture parameters:

- First level split instruction and data cache: 8 KB, direct-mapped, 32-byte line, 1-cycle hit time.
- Second level unified cache: 64 KB, 4-assoc., 32-byte line, 6-cycle hit time, 30-cycle miss time.
- Instruction TLB: 16 entries, 4-way set-associative, 1-cycle hit time,30-cycle miss time.
- Data TLB: 32 entries, 4-way set-associative, 1-cycle hit time, 30-cycle miss time.

The processor does not perform any type of branch prediction, thus every branch stalls the fetch stage until the branch is resolved in the ALU stage. This is in keeping with some very low power embedded processors, although the trend is toward implementing branch prediction.

We used the Mediabench benchmark suite [37], which was compiled with the gcc compiler with "-O3 -finline-funtions -funroll-loops" optimization flags into a MIPS-like ISA. As a baseline for comparison we use a conventional 32-bit wide processor, with 5 pipeline stages: Instruction Fetch, Decode and Register Read, Execute, Memory, and Write Back.

As explained in Chapter 1, dynamic power consumption is directly proportional to the switching activity of the different components. Thus, reducing the switching activity will result in a reduction in energy consumption at the same level.

4.5 Byte-Serial Implementation

Having established potential activity reductions that can be achieved (and therefore energy reductions), we now consider implementations that attempt to achieve these levels while providing good performance.

Implementations will differ in total hardware resources although they may not necessarily differ in circuit activity.

First, we consider a simple byte-serial implementation that has a one byte wide data path. If more than one data/address byte is needed at a given stage, then that pipeline stage will be used sequentially for multiple cycles. While later sequential data bytes are being processed, however, earlier bytes can proceed up the pipeline. For example, if it is necessary to read 3 bytes from the register file, first the low order byte is read and passed on to the EX stage, then while the next byte is being accessed, the EX unit can perform on the first data byte and pass it to the data cache stage.

Figure 28 shows the byte-serial implementation. In this microarchitecture there is a single register file bank, a single ALU, and a single data cache bank, all one-byte wide. Inter-stage latches are provided to store values on a byte basis and only the significant bytes are required to be latched. In addition, the extension bits must flow through the pipeline and a three bit latch is provided between some stages for this purpose. The ALU stage includes a special unit that operates on extension bits as described in Section 4.2.4. There is one byte-wide PC increment unit that operates serially and three instruction cache banks, which are accessed in the first stage along with the extension bit. Then, if the extension bit indicates that it is needed, the instruction remains in this stage for one more cycle while one of the banks is accessed again. Using a three byte wide instruction cache stage is a departure from the strictly byte serial implementation. This decision was made to avoid excessive stalls while reading instructions; otherwise, every instruction would incur at least two stall cycles because the average number of bytes per instruction is 3.17.

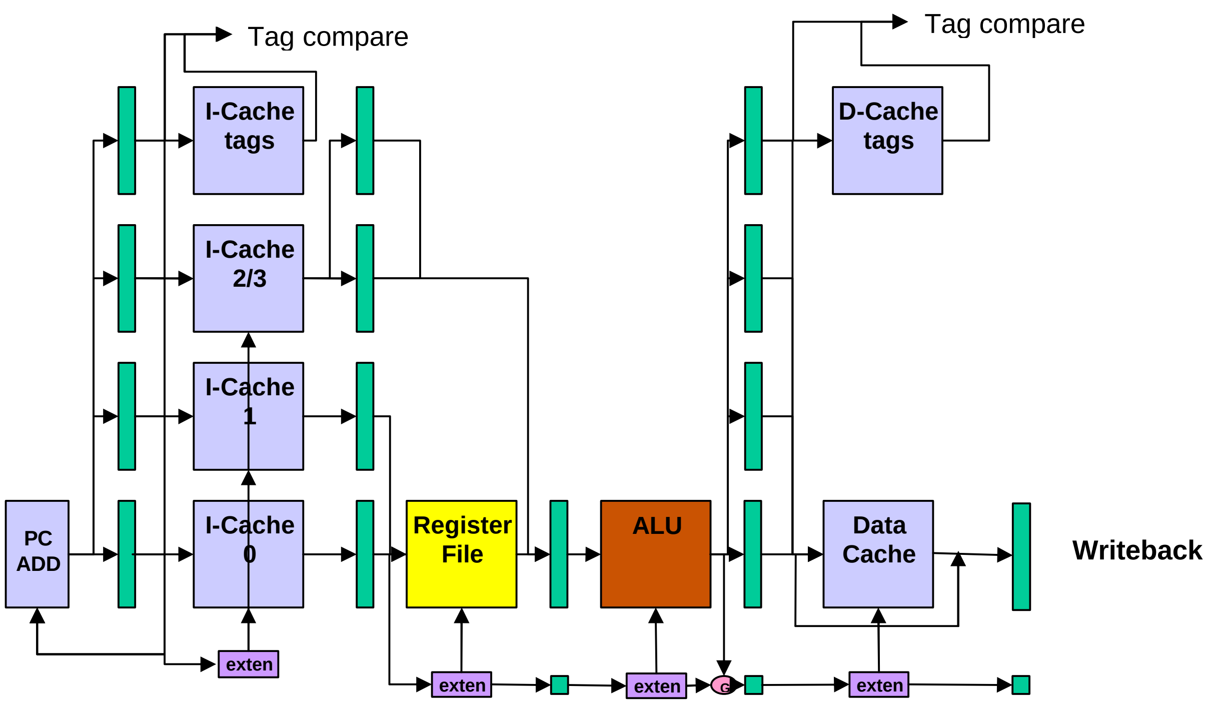


Figure 28: Byte-serial implementation

Figure 29 shows the performance of the byte-serial implementation, expressed as cycles per instruction (CPI). For comparison, the CPI of a baseline 32-bit wide implementation is also shown. For most programs, the performance of the byte-serial implementation is significantly lower than that of the 32-bit processor. CPI is increased by 79% on average, although activity (and energy) is reduced by 30-40% for most of the pipeline functions (Table 5).

If the pipeline is widened to 16-bits, the average CPI becomes 1.96, which is just 29% higher than that of the byte-wide implementation, but the activity savings are lower (around 20-30% for most of the pipeline functions). Note that the relative performance of the pipelined schemes is quite uniform across all the benchmarks.

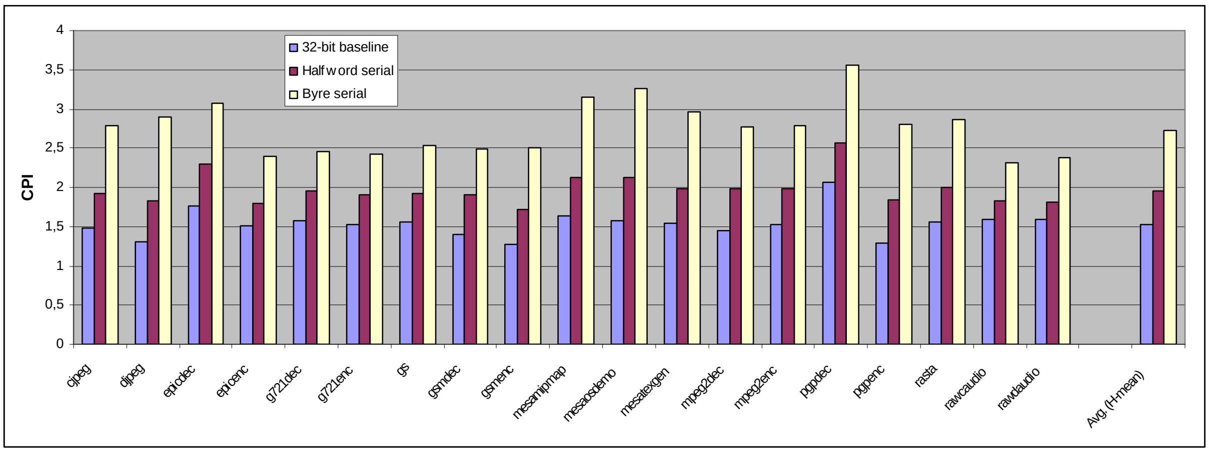


Figure 29: Performance of the byte-serial architecture

4.6 Semi-Parallel Implementations

The byte-serial implementation achieves significant activity reduction, but at the cost of substantial performance losses with respect to the baseline 32-bit pipeline. For some applications, energy savings may be much more important than performance, and this will represent a good design point. There may be other applications, however, where performance is more important, and performance losses should be reduced. We now consider methods that retain low activity levels, but use additional hardware to improve performance.

The principle is to improve performance by adding additional byte-wide datapath elements at the various pipeline stages. For example, the register file can be constructed of two byte-wide files (rather than one) and produce a full data word in 2 cycles instead of 4. Similarly, multiple byte-wide ALUs can be used to improve throughput in the execute stage of the pipe.

Adding these units does not necessarily increase circuit and memory access activity, however, because not all the units have to be enabled every cycle. For example, if a data item has only one significant byte, then a register access can be performed for one byte of a two byte wide register file, while the other byte is disabled. Similarly, if the source operands of an addition only have two significant bytes, these bytes will be operated in two of the ALUs while the others will be disabled.

Finally, the numbers of byte-wide units in each of pipeline stages do not have to be the same. That is, the number of byte units or memories can be established to permit balanced processing bandwidths among the pipe stages.

To determine how many parallel units and memories should be used, we first undertook a bottleneck study of the byte-serial implementation to see where the major stalls occur. We observed that in the byte-serial architecture the ALU is the most important bottleneck, 72% of the stalls were caused by structural hazards in the EX stage. Thus, increasing the bandwidth of the ALU stage is the most effective approach to increase performance. To quantify how much bandwidth is required in each stage, we did the following simple analysis.

Consider each of the major pipeline stages. First, the study in Section 4.2.2 shows that an instruction requires about 3.2 bytes to be fetched on average. The ALU operates on an average of 2.7 bytes, but since the maximum CPI is 1.5 (32-bit baseline processor), the activity of the ALU will not be higher than 2.7/1.5 = 1.8 bytes/cycle on average. Next, around one third of instructions access memory, and each access is 2.8 bytes wide on average. Thus, less than one byte per cycle is accessed on average. Based on this study, we determined that a good balance is achieved with an instruction cache three bytes wide, a register file and ALU two bytes wide, and data cache one byte wide.

An implementation for this configuration is shown in Figure 30 and is referred to as byte semi-parallel. The instruction cache essentially contains three byte-wide banks and works as in the byte-serial implementation.

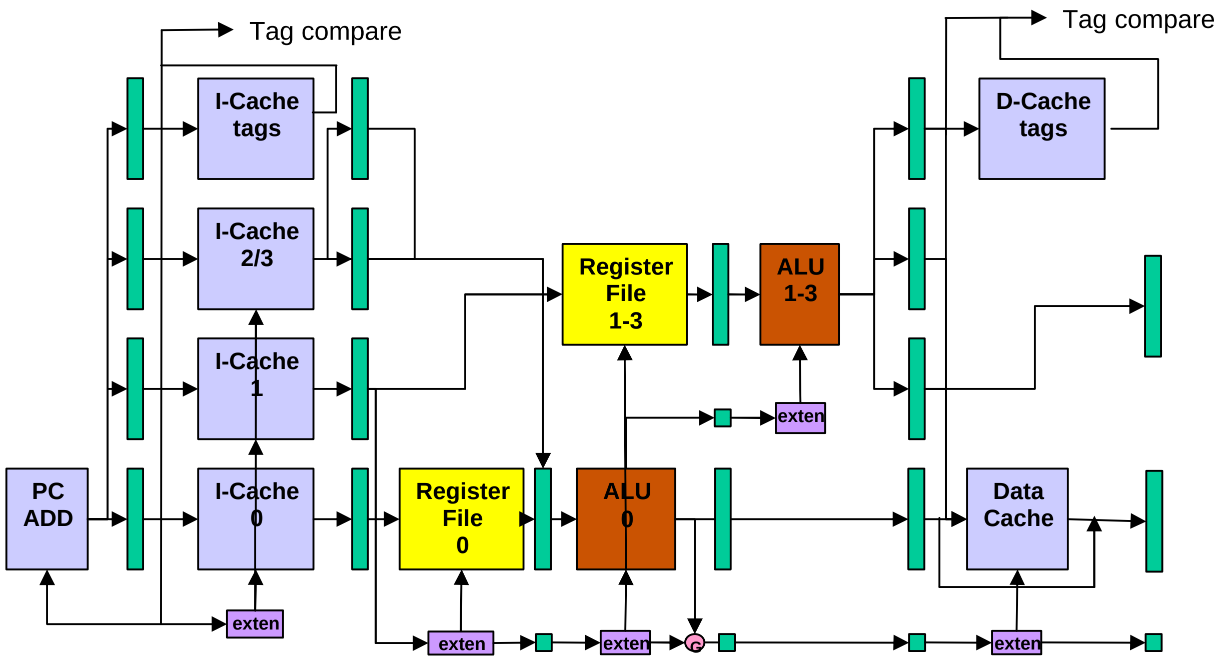


Figure 30: Byte semi-parallel implementation

The register access stage is skewed with the low order byte being accessed first together with the extension bits. In the next stage the low order byte is operated on, and at the same time another register byte is read if needed according to the extension bits. If there is more than one additional byte the instruction uses this stage for multiple cycles. The next stage performs the ALU operation on the additional bytes and is used for as many cycles as the previous stage. The following stage performs the data cache access (if needed). It first reads/writes the low order byte, the tags, and the extension bits and, according to the latter, the instruction uses this stage sequentially for multiple cycles until all data are read/written. Finally, the last stage writes the result into the register file. It first writes the low order byte, the extension bits and one additional byte if needed. If more than one additional byte must be written, the instruction uses this stage for multiple cycles.

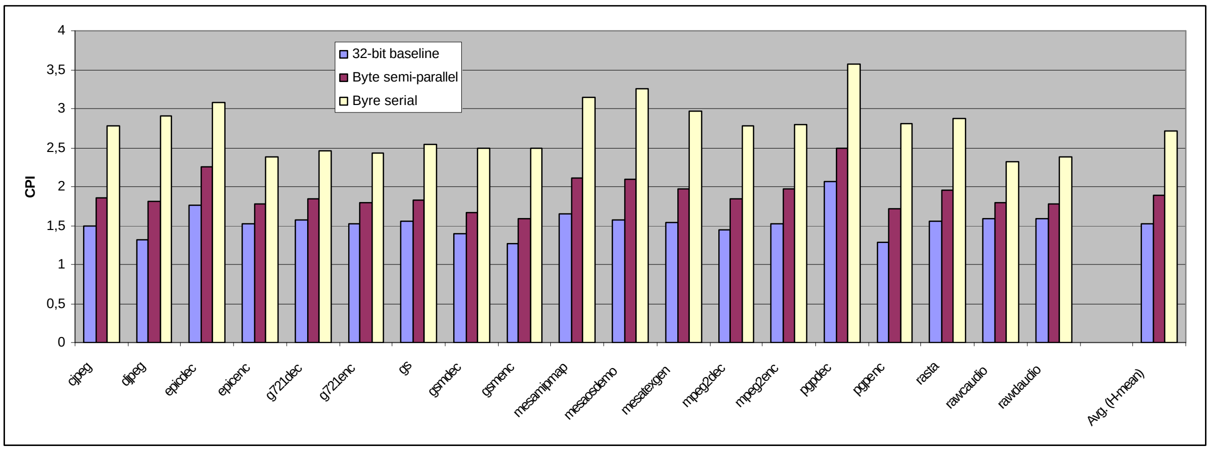


Figure 31: Performance of the byte semi-parallel microarchitecture

Figure 31 shows the CPI of this microarchitecture along with that of the 32-bit baseline processor and the byte-serial implementation. On average, the CPI is 24% higher than the 32-bit baseline processor. We observe that the performance is much closer to the 32-bit implementation than the byte-serial implementation while all the activity savings are retained except for a few additional latches.

4.7 Fully Parallel Implementations

The microarchitecture above still loses some performance – bottlenecks cannot be perfectly balanced all the time because of bursty behavior that most programs exhibit. So, we consider pipelines with maximum (4 bytes) parallelism at each stage, and use operand gating to enable only those datapath bytes that are needed. This requires a skewing of stages in a similar way to the semi-parallel implementation described in the previous section. A block diagram of a portion of the microarchitecture, which is referred to as byte-parallel skewed, is depicted in Figure 32.

This pipeline is optimized for the long data case, i.e. where the pipeline keeps flowing even if each operand is a full 4 bytes. No stage is used more than once (except for the PC computation in very few cases). Although the activity of the functional units is the same as that of the byte-pipelined and semi-parallel implementation, the longer pipeline of the byte-parallel skewed implementation implies more latch activity and more backward bypasses.

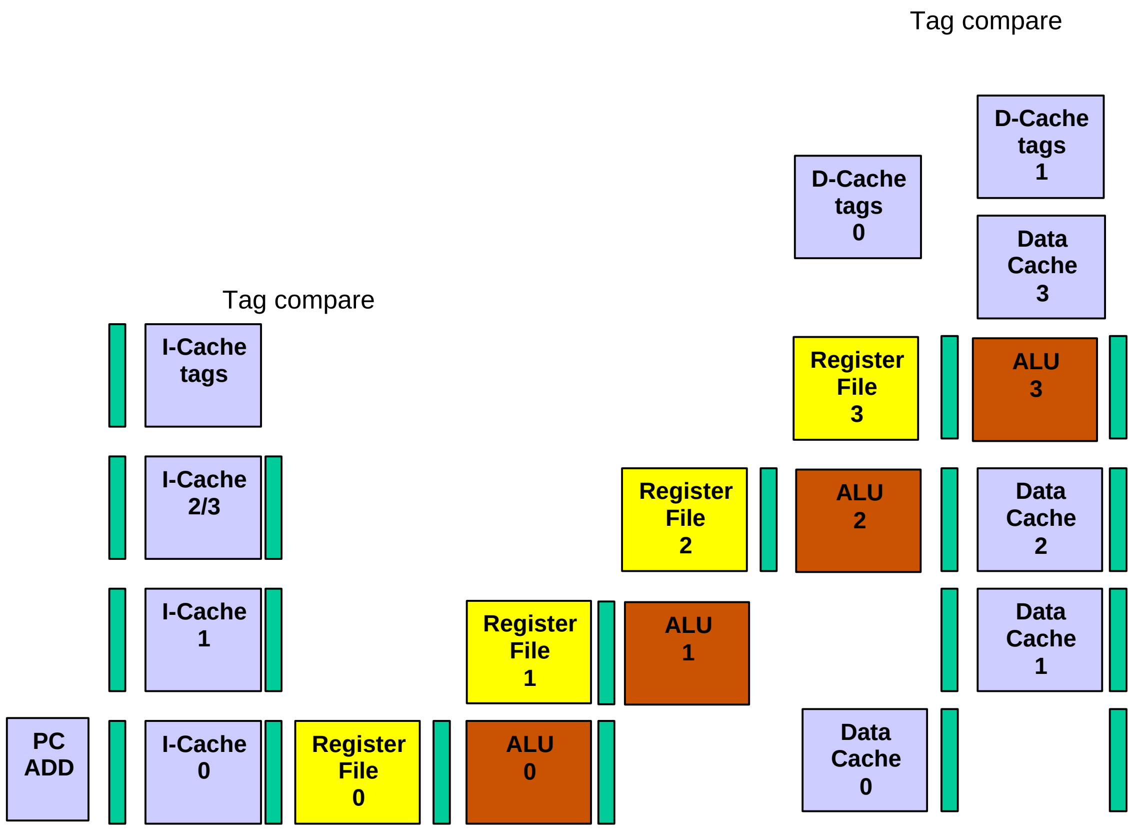


Figure 32: Byte-parallel skewed microarchitecture

The performance of this microarchitecture is shown in Figure 33. We can observe that the CPI is very close to that of the 32-bit baseline processor for all programs. On average, it is just 3% higher.

Another alternative is a "compressed" parallel pipeline implementation (see Figure 34). In this case, the pipeline consists of the original 5 stages. Each instruction spends one cycle in the Ifetch stage to read 3 bytes and an additional one if a fourth byte is needed. Then it moves on to the second stage where it reads the low order byte and the extension bits. If more bytes are needed, the instruction spends one more cycle in the same stage to read all of them in parallel. Then the instruction moves on to the ALU stage where it executes in a single cycle, using only the functional units that operate on significant bytes. Then it moves on to the memory stage where it reads first the low order byte and the extension bits, and if needed, it spends an additional cycle to read all the remaining bytes. If it is a store, all the significant bytes along with the extension bits are written in a single cycle. Finally, all significant bytes and the extension bits are written into the register file in a single cycle.

This design works well for short data because the pipeline length is kept minimal and this reduces the branch penalty and the number of backward bypasses. Furthermore, functional unit and latch activity is kept minimal (equal to the byte-serial implementation). However, full-width (32-bit) data operations suffer stalls in some stages, which result in performance losses when compared with the full parallel implementation. Performance is shown in Figure 35. The CPI increase compared with the 32-bit baseline processor is 6% on average, which is quite close to the performance of the byte parallel skewed configuration.

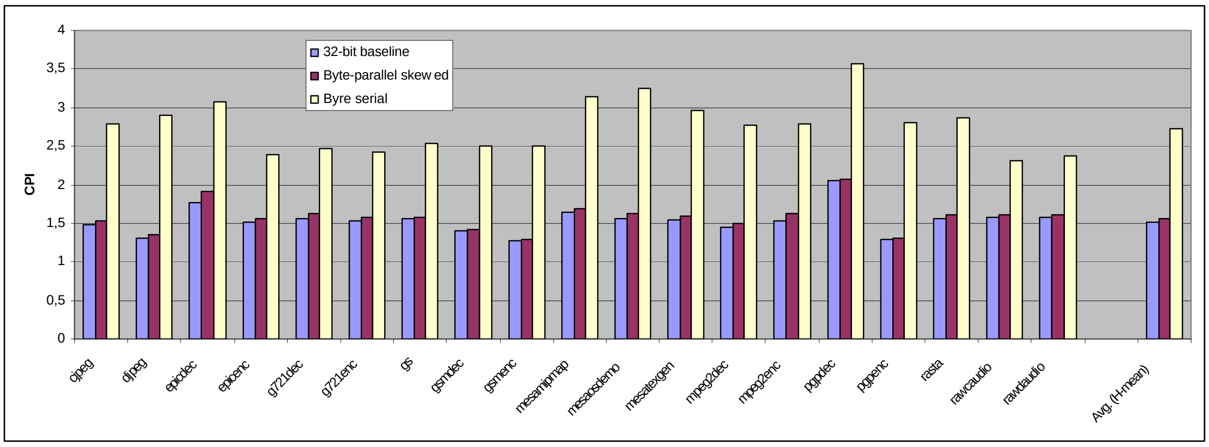


Figure 33: Performance of the byte-parallel skewed microarchitecure

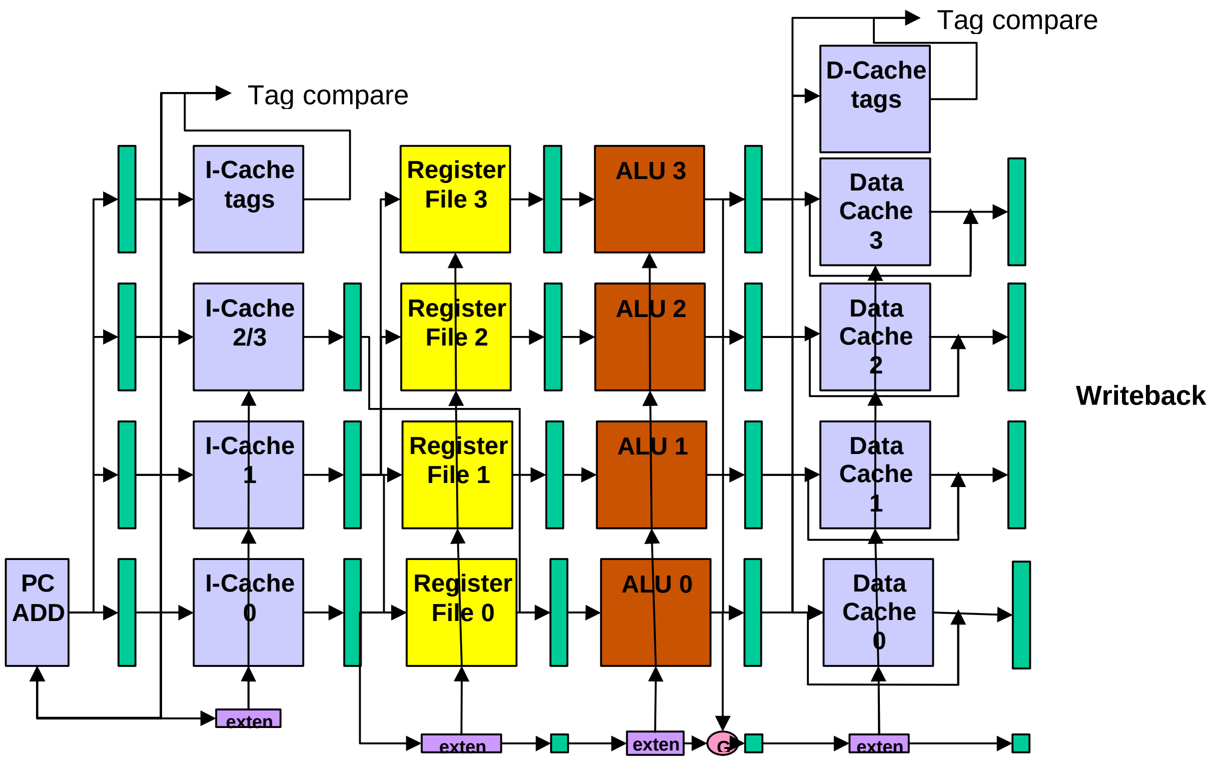


Figure 34: Byte-parallel compressed pipeline

We can get the best of both (performance wise) by putting forwarding paths into the byte-parallel skewed pipeline. In this way, when a short operand is encountered, it can skip the stages where no operation is performed. This reduces the latch activity to the same level as that of the byte-serial implementation, and at the same time the effective pipeline length is shortened, which reduces the branch penalty. However, the number of backward bypasses is the same as that of the byte-parallel skewed implementation.

The performance of this architecture is also shown in Figure 35. Now performance is very close to the baseline 32-bit processor (the CPI is only 2% higher on average) while the activity is reduced around 30-40% for most of the stages. A disadvantage, however, is that this design has rather complicated control and many data paths (for forwarding).

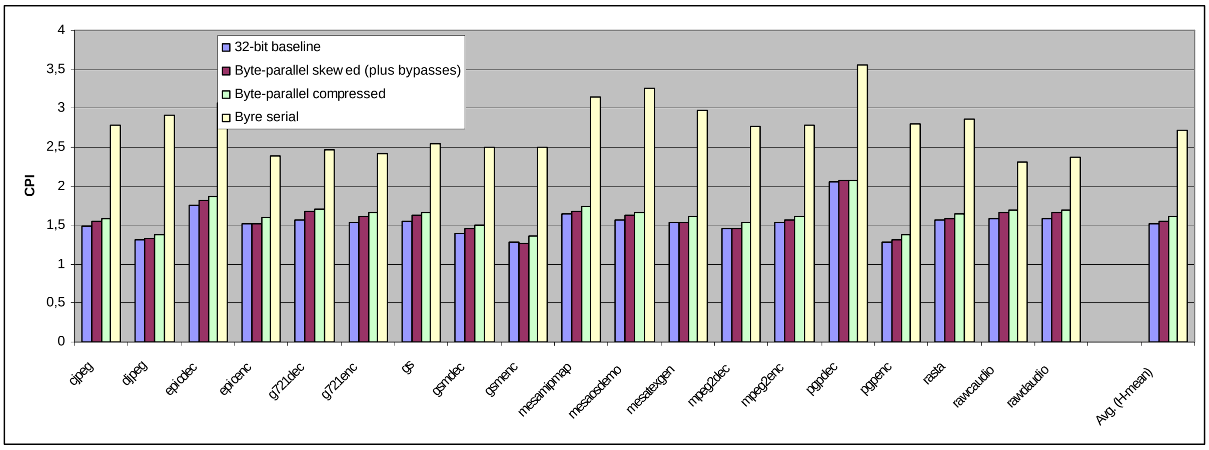


Figure 35: Performance of the byte-parallel compressed and skewed + bypasses microarchitecture

4.8 Summary and Conclusions

The significant bytes of instructions, addresses, and data values essentially determine a minimal activity level that is required for executing a program. For a simple pipeline design, we showed that this level is typically 30-40% lower than for a conventional 32-bit wide pipeline. Every stage of the pipeline shows significant activity savings (and therefore energy savings).

We proposed a number of pipeline implementations that attempt to achieve these low activity levels while providing a reasonable level of performance. The byte-serial pipeline is very simple hardware-wise, but increases CPI by 79%. For some very low power applications, this may be an acceptable performance level, in which case the byte-serial implementation would be a very good design choice.

For higher performance, the pipeline stages can be widened. A rough analysis indicates that three bytes of instruction fetch, two bytes of register access and ALU, and one byte of data cache might provide a good balance of bandwidths. For this configuration, the CPI is 24% higher than that of the full width baseline design. Activities are still at their reduced levels, and this design may provide a very good design point for many very low power applications.

Finally, we considered designs with a four byte wide datapath at each stage. Operand gating is retained for reducing activity, but under ideal conditions throughput is no longer restricted. These designs can come very close in performance to the baseline 32-bit design while again retaining reduced activity levels. The disadvantage of these schemes is an increased latch activity and static energy consumption, or additional forwarding paths or more complex control. We believe that these may be a very important class of implementations however, because of their high performance levels, and low activity.

"There is more life than increasing its speed"
Gandhi

Chapter 5

Value Compression for High-Performance Microarchitectures

Hardware and software techniques to reduce the energy consumption in high-performance 64-bit microarchitectures are explored in this chapter. While hardware techniques can benefit of adapting to the run-time circumstances, compile-time techniques have some advantages over hardware schemes such as the minimal impact on the microarchitecture and the wider scope over the program code. This chapter analyzes several hardware compression mechanisms and two compile-time techniques that can reduce the energy requirements of the code executed.

5.1 Introduction

In Chapter 3, three value compression methods were proposed and their potential energy reduction was computed. In this chapter, we first implement the value compression methods presented in Chapter 3. Then, we propose and study two software-based approaches that exhibit a different set of hardware/software tradeoffs, namely *Value Range Propagation* and *Value Range Specialization*.

Assuming that the instruction set architecture (ISA) contains opcodes that specify operand lengths (e.g. *load byte, add halfword*) -this feature is already available in some conventional instruction sets and could be added as an extension to others- at compile time, or as part of static binary translation, an enhanced version of *Value Range Propagation* is used to determine bounds on the *useful* value ranges of all variables. Then, through proper opcode assignment, only useful portions of data are computed, communicated, and stored, thereby saving power. This approach has much less added hardware complexity than the hardware value compression methods proposed, but requires static analysis by the compiler or translator and may require additional instruction opcodes to specify operand widths (depending on the base ISA).

5.2 Hardware Value Compression

Chapter 3 has shown three value compression mechanisms that are effective in reducing the number of bits needed through the pipeline. In this section, we analyze the performance and the reduction on energy of several configurations of the compression methods. The following table shows the value configuration formats we consider in this section.

Value compression method	Classification of the values	Number of extra bits per value
Size 8-64	8 bits or 64 bits	1
Size 16-64	16 bits or 64 bits	1
Size 32-64	32 bits or 64 bits	1
Size 40-64	40 bits or 64 bits	1
Size 8-16-32-64	8 bits, 16 bits, 32 bits or 64 bits	2
Size 8-16-40-64	8 bits, 16 bits, 40 bits or 64 bits	2
Significance 8-16-24-32-40-64	Bytes 2,3,4,5 sign extended one byte, or byte 6 extended by two bytes.	5
Significance 8-16-24-32-40-48-56-64	Bytes 2,3,4,5,6,7,8 sign extended one byte	7
Zero 8-16-24-32-40-64	Bytes 2,3,4,5 can be zero or bytes 6 through 8.	6
Zero 8-16-24-32-40-48-56-64	Any byte can be a zero	8

A more detailed study of the average compressed value size using the schemes listed above is shown in Figure 36. The average size was computed as the average of the number of bytes for each access to the register file, data cache, instruction cache (address), functional units, and the rename buffers. Note that the structures considered in this study differ from the ones in Chapter 3, thus the numbers showed are different. The first column shows the average data size taking into account the format bits, and the second column shows the average size without the format bits. On average, ignoring the format bits, the *zero* *compression* mechanism achieves the best compression (21.2 bits for the configuration that can compress every byte). However, when the format bits are included, the best scheme is the *size compression* mechanism with an average of 28 bits per value (for the configuration in which the values are compressed to 8, 16, 40 or 64 bits).

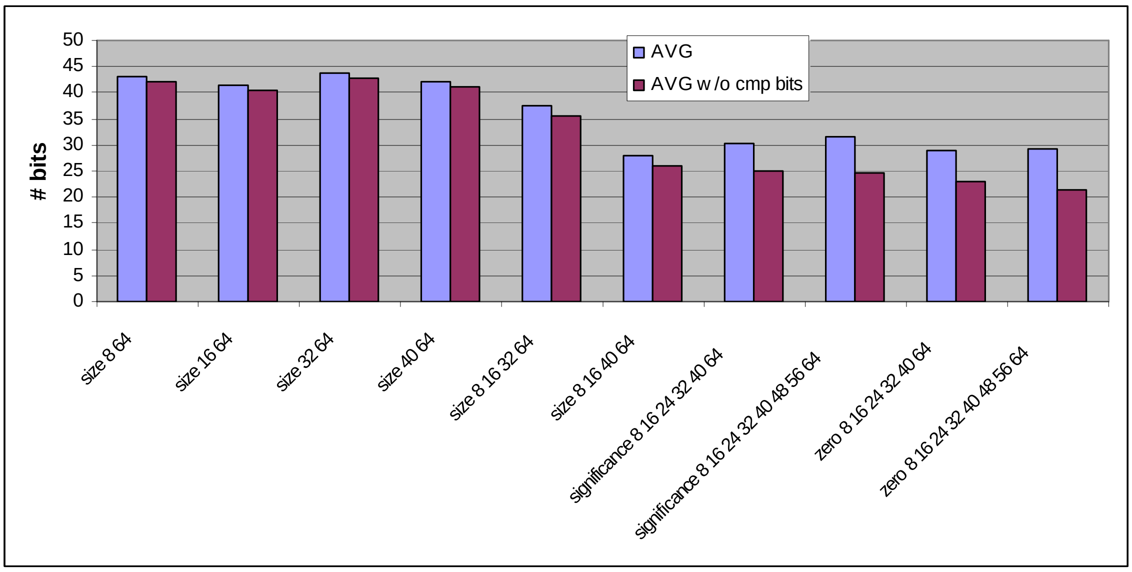


Figure 36: Average Data Size for the SpecInt95

This study points out that any of the three proposed schemes can perform well (they reduce the effective data-width from 64 bits to fewer than 30 bits). In the next section, we describe several proposals for using value compression for different processor's datapath subsystems. Then we analyze the energy consumption for the three value compression mechanisms.

5.2.1 Hardware Extensions to a Conventional Processor

Several data-dependent structures have been modified in order to consider value compression. The structures modified and the ways they are modified are explained in this section.

First, the register file is extended to hold the extension bits necessary to keep the compression information. At the same time, each access to the register file will just use those banks that keep significant data. In other words, compression bits are used to enable/disable the portions of the register file that have useful/unneeded information.

The data-cache holds compressed values. Thus each write access to the cache will consist of the compressed value plus de extension bits. When a read is performed, just the significant data will be read along with the extension bits. In the same way as the register file, the compression information (extension bits) will be used to just access the banks that have significant data. In a similar way, the instruction-cache is compressed. The only difference is that since the instruction word is 32-bit long, the compression mechanisms have to be adapted to this size. Section 5.2.3 describes in detail the exact implementation.

Due to the simplicity of the compression methods used, functional units are capable of computing with compressed values and generate accordingly a compressed output value.

The instruction-queue and the rename-buffers hold compressed values, as well. In this case, the value is kept compressed –together with the extension bits. When a read or a write is performed, just those significant bytes are read/written. Finally, in the case of the BTB, the branch targets are kept in a compressed form. Branch targets are 64-bit addresses, thus it is possible to compress them in the same way as the other values that flow through the pipeline are compressed. The branch predictor cannot be compressed since it hold two-bit saturating counters.

5.2.2 Experimental Framework

The Wattch [8] toolset is used to conduct the evaluation. The main architectural parameters of the assumed out-of-order processor are described in Table 1. We use the programs from the SpecInt95 suite with their reference inputs. All benchmarks are compiled with the Compaq-Alpha C compiler with the maximum optimization level. Each benchmark was run to completion.

Table 7: Machine parameters

Parameter	Configuration
Fetch Width	4 instructions
I-cache	64KB, 2-way set-associative. 32-byte lines, 1-cycle hit time, 6-cycle miss penalty.
Branch Predictor	Combined predictor of 1K entries with a Gshare with 64K 2-bit counters, 16 bit global history, and a bimodal predictor of 2K entries with 2-bit counters.
Decode/Rename width	4 instructions
ROB size	64
Instruction Queue	64
Retire width	4 instructions
Functional units	3 intALU + 1 int mul/div3 fpALU + 1 fp mul/div
Issue mechanism	4 instructions Out-of-order
D-cache L1	64KB, 2-way set-associative. 32-byte lines, 1-cycle hit time, 6-cycle miss penalty 3 R/W ports
I/D-cache L2	256 KB, 4-way set associative, 64-byte lines, 6-cycle hit time. 16 bytes bus bandwidth to main memory, 16 cycles first chunk, 2 cycles interchunk
Physical registers	96

5.2.3 Energy Savings

In addition to the average data size (shown in Figure 36), several other factors such as the switching activity are important when computing dynamic energy reduction. Although storing more compression bits results in wider structures, the activity of these wider structures is the one that sets the energy consumption. Thus, it can happen that a wider structure has less activity than a narrower one (i.e. with less extra-bits but with more switching activity).

In Figure 37, we can see the energy savings of the mechanisms analyzed in this section. The figures show the overall processor energy savings, nevertheless, value compression has been applied to just the d-cache, ALU, register file, rename buffers, instruction queue and BTB. The *significance compression* is the one that achieves higher energy savings (close to 30%) despite the use of 7 extra bits per word. The best *size compression* scheme (close to 20% energy savings) is the one that compresses values to 8, 16, 40 and 64 bits. The fact that it includes memory addresses (typically 5 bytes long) makes it perform better than the other *size compression* mechanisms. The *zero compression* mechanism achieves a maximum of 20% overall energy reduction.

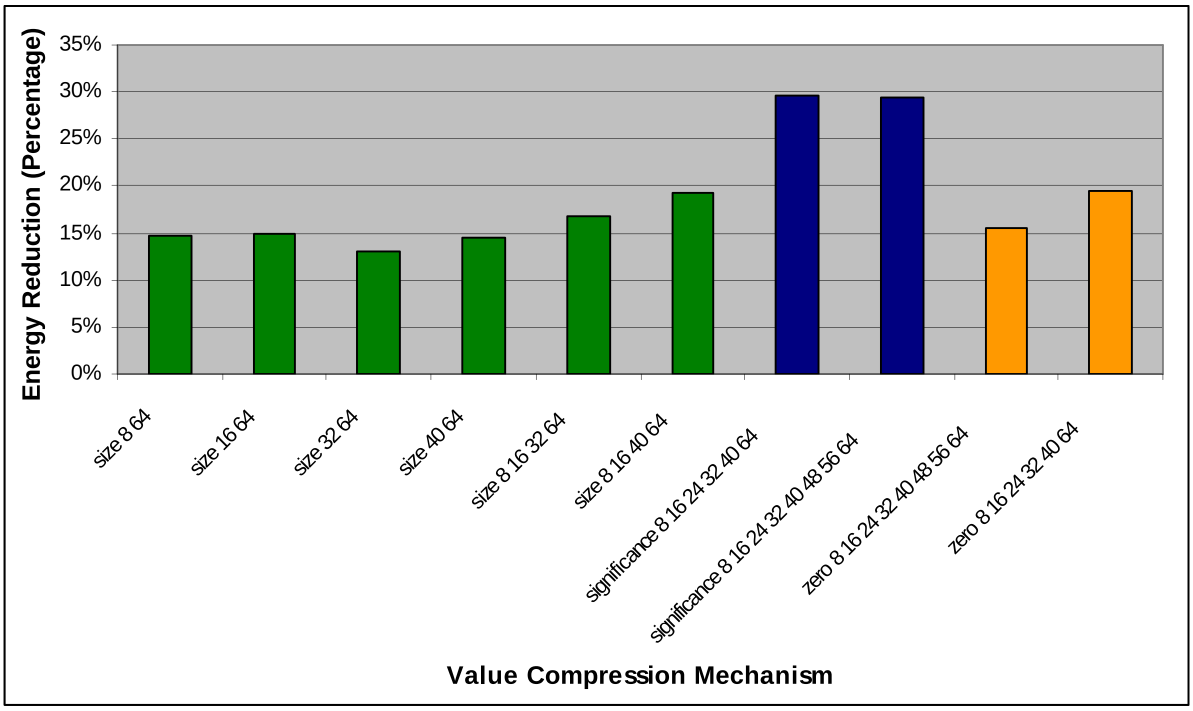


Figure 37: Processor Energy savings

In the following figures, we analyze behaviour of the value compression schemes for several structures (instruction-cache, data-cache, register file and ALU). Figure 41 shows the energy benefits in the data cache (both addresses sent to the cache and the data stored/loaded). The distribution of the energy savings in the data cache is similar to that of the whole processor. In this case, the *significance compression* energy savings are close to 14% and the *significance compression* that compresses all the bytes (not only until the 5th byte) performs better than the other configurations of the *significance compression*. Figure 39 shows the energy savings for the register file. The savings scale up to 50% for the *significance compression* while the *size compression* reaches a 40% reduction in energy and the *zero compression* stays a little bit behind. Figure 40 shows the reduction in the ALU. The difference between the *significance compression* and the other schemes is larger in this case (almost 60% vs 35%). Finally, Figure 41 shows the reduction in energy in the instruction cache. Since the instruction word is 32-bit wide (Alpha ISA) just three mechanisms have been evaluated. The first one (labeled as *size*) compresses the data to 8, 16, 24 or 32-bit data in the same way as the *size compression* presented earlier in this work. The second one (labeled *significance*) compresses the instructions using significance compression to 8, 16, 24, and 32-bit data. Finally the third column (labeled *zero*) compresses the instructions using *zero compression* where each byte of the 32-bit word can be tagged as being zero. All the schemes perform quite well and they achieve more than 30% energy reduction in the instruction cache meaning that the instructions have a very compressible form that the schemes are able to find and exploit.

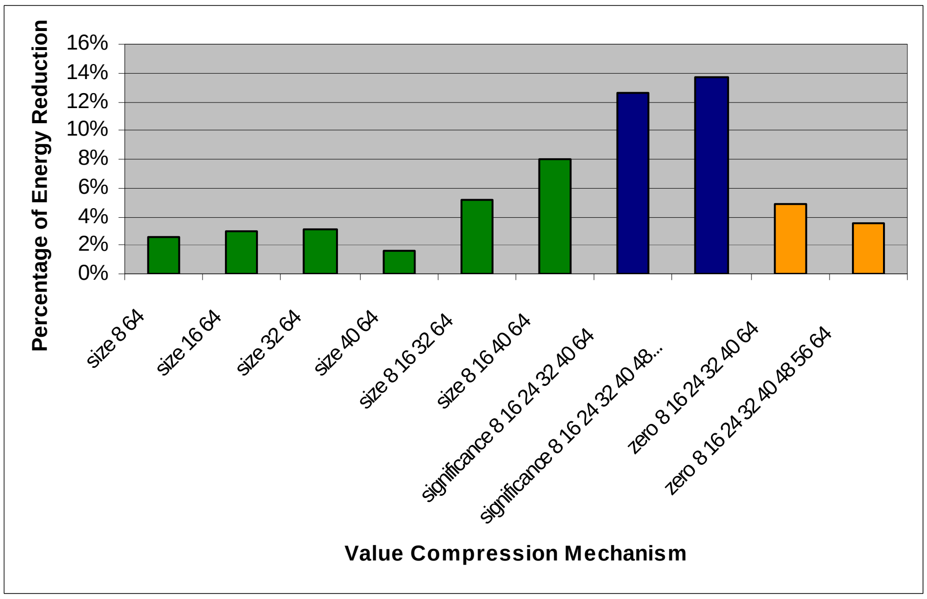


Figure 38: Energy savings for the Data Cache (SpecInt95)

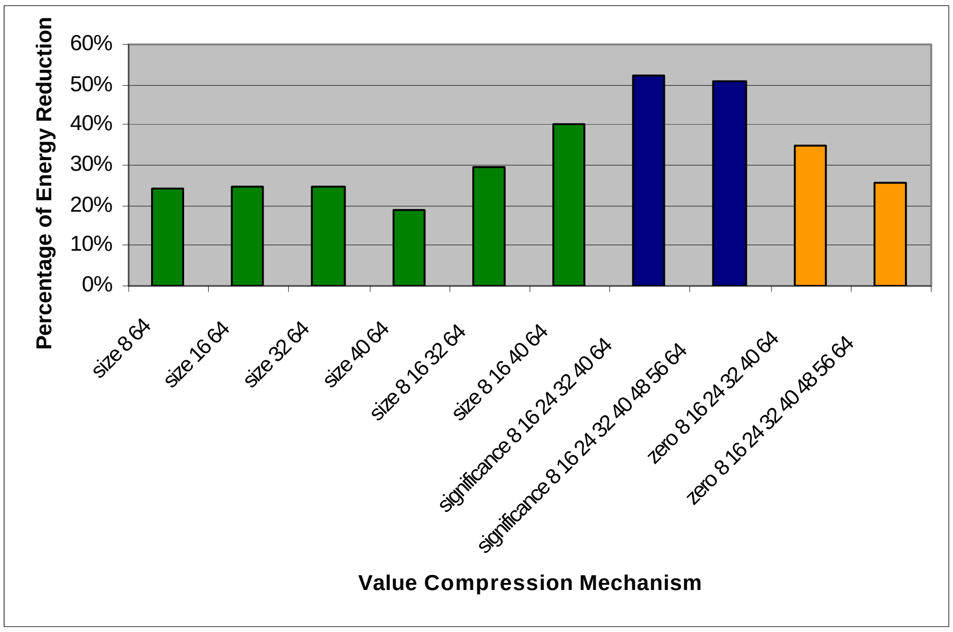


Figure 39: Energy savings for the Register file (SpecInt95)

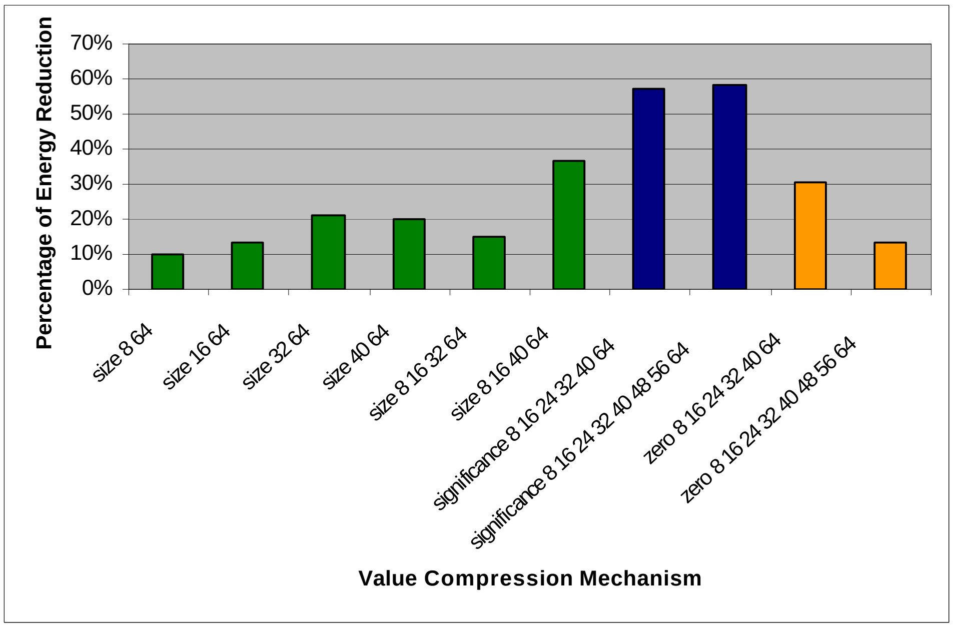


Figure 40: Energy savings for the ALU (SpecInt95)

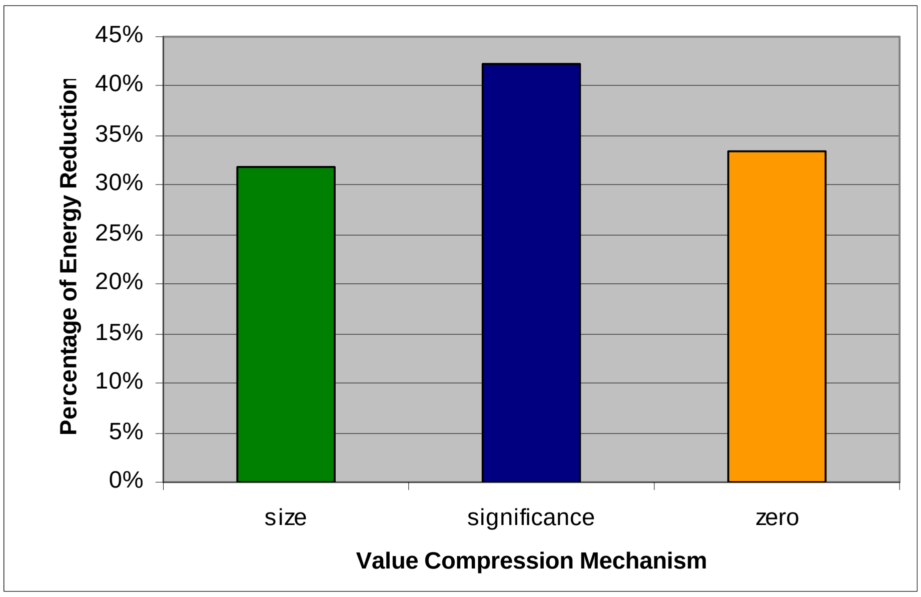


Figure 41: Energy savings for the I-Cache (SpecInt95)

5.2.4 Peak Power Reduction

Peak power is also an interesting metric since it determines the maximum possible burst of power that a processor might dissipate. The peak power shown in Figure 42 corresponds to the execution of the SpecInt95 suite. Nevertheless, one could think of a code where the compression schemes do not achieve any size reduction. In this case, the peak power would not be reduced. In fact, the extra bits needed by the data compression could even increase the worst case peak power. Nevertheless, the small complexity of the required hardware mechanisms does not add a significant overhead in this worst case peak power since there are more power hungry units such as the clock network and the caches. Although one might think that compressing the data might not have a direct impact on peak power since there might be a cycle where every computation will need 64 bits. However, experiments show that peak power is significantly reduced with the proposed compression mechanisms. As in the case of the energy consumption, the *significance compression* mechanism achieves a 25% peak power reduction. It is interesting to see that the configuration of the *significance compression* that achieves the highest energy reduction (see Figure 37) is not the best in terms of peak power reduction (see Figure 42) where the scheme that compresses all the bytes (*significance 8,16,24,32,40,56,64*) performs a little bit better. The fact that it can compress bytes within large words makes it perform better in terms of peak power. The *size compression* mechanism achieves, in its best configuration, a 15% peak power reduction while the *zero compression* mechanism stays below the 15% line.

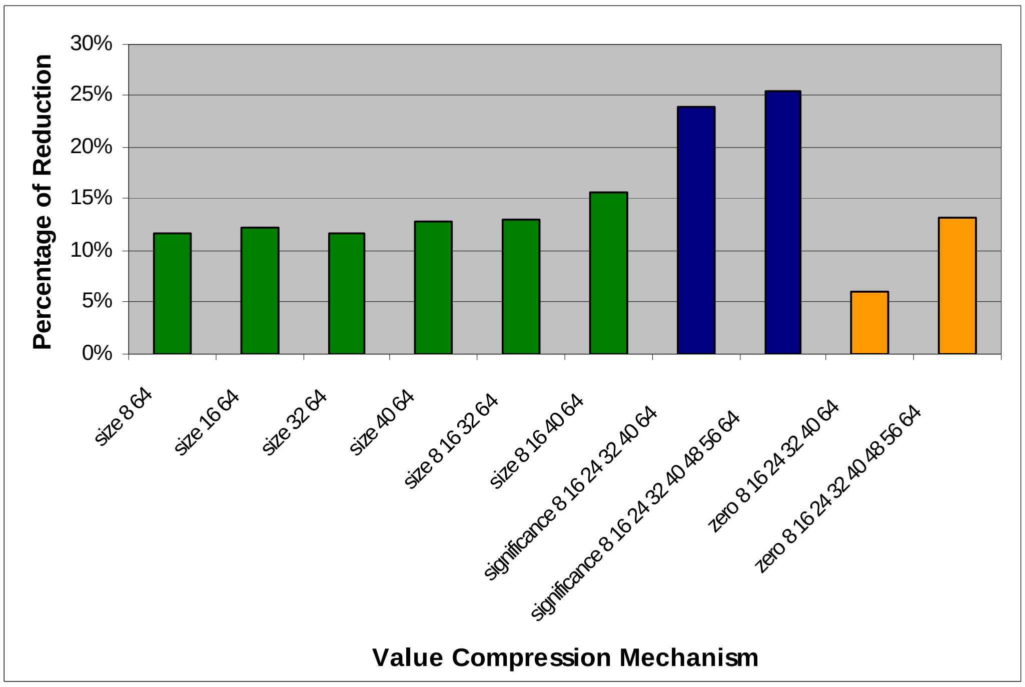


Figure 42: Peak power reduction (SpecInt95)

5.3 Value Range Propagation

If instruction opcodes encode the width of the operands involved (as it is done in conventional ISAs), propagating the range of values that registers will have at run-time can be used to help chose an opcode that determines narrower operands, and thus to reduce the run-time width of the operands involved. A compiler or binary translator can implement the proposed value range propagation (VRP) technique. It first finds individual instructions where value ranges can be easily determined. It then propagates this information to other instructions and determines value ranges that each integer register may have at run time. Methods are given for propagating value ranges within loops and across procedure boundaries. Value range information can then be used to determine the number of bits that must be computed and stored in order to maintain correct the semantics of the original HLL program. Finally, opcodes are assigned to specify the needed value ranges.

In our study, based on a 64-bit architecture, we assume opcodes may specify operand widths of a byte, halfword, word, and doubleword. Many conventional ISAs already support many of the needed opcodes; otherwise opcode sets will need to be enhanced.

VRP is always done in a conservative manner (and thus requires no hardware or software recovery techniques). All the decisions concerning unknown ranges are always in the conservative worst-case direction, ensuring the correctness of results. If there is a case where a given value is used in more than one place with different widths or ranges, the widest range is assumed.

Furthermore, value ranges are not propagated through memory. To perform accurate VRP through memory, address alias analysis is required. To keep things simple in our initial implementation, all memory values are assumed to be 64 bit values (unless the specific data declaration is otherwise).

5.3.1 Finding Initial Value Ranges

Following are cases where individual instructions can have their value ranges immediately bound, irrespective of other instructions in the scope of a compiler or binary translator:

- Instructions that are declared to have narrow-width operands; for example, in terms of a HLL like C: "`int a; a=a+1`" where an *int* is 32 bits. The compiler would select a 32-bit addition if available as an opcode. Or, if binary translation is being used, the original binary would have a narrow-width opcode (I.e. *add_long*), and the binary translator can use this information (the 32-bit addition opcode) to infer that the range of the result value.
- Assignments of the type (VAR=constant). The value range of *VAR* is defined by the single constant value.
- If-condition statements whose evaluation implies a bound on the tested variable along the taken or fall-through path. For example "`if (X >= 7) then….`" places a lower bound on *X* along the taken path and an upper bound on *X* along the fall-through path.

5.3.2 Forward and Backward Value Propagation

Given the initial value range information, additional information can be derived for other instructions via a propagation process, as follows. The propagation alternates between forward and backward traversals of the program's data-dependence graph until a constant state is reached. During a forward propagation, the dependence-graph is traversed in a top-down style. For each instruction, the range of the output value (if any) is determined based on the range(s) of the input operands. In a backward propagation, the traversal of the dependence-graph is done in a bottom-up manner. During this traversal the range of the every instruction's input operands is set depending on its output value, the type of operation, and any previously set input values.

Following subsections describe value range propagation for some of the more common instruction types.

5.3.3 Addition

Given the value range information of the inputs (RangeIn1 and RangeIn2); the value range information of the output variable will be:

In a forward traversal:

```
RangeOut.MinVal=(RangeIn1.MinVal+RangeIn2.MinVal)
RangeOut.MaxVal=(RangeIn1.MaxVal+RangeIn2.MaxVal)
```

If an input value can be produced by different instructions, there is an additional step of defining a range for each of the potential input values. Then, the minimum value for a given input variable will be calculated as the minimum value of all the possible producers and the maximum value as the maximum of all the possible producers. This technique assures that the widest possible value range is used.

In a backward traversal:

```
RangeIn1.MinVal=(RangeOut.MinVal-RangeIn2.MaxVal)
RangeIn1.MaxVal=(RangeOut.MaxVal-RangeIn2.MinVal)
RangeIn2.MinVal=(RangeOut.MinVal-RangeIn1.MaxVal)
RangeIn2.MaxVal=(RangeOut.MaxVal-RangeIn1.MinVal)
```

If there is a case where the output value is used in more than one instruction, the above expressions are applied to all of the dependent instructions. Consequently, the input values will take the minimum and maximum values calculated for all the dependent instructions. A similar approach is used in the other arithmetic instructions.

In arithmetic operations there can be the possibility of overflows. In this case, we assume that conventional two’s complement arithmetic is used (i.e. overflows wrap-around). If overflow is possible then the calculated range takes wraparound behavior into account. Although in many cases this may be overly conservative, it ensures correctness of the generated code.

5.3.4 Loads

In a forward traversal, the range of the output value (the loaded value) is set to the maximum and minimum values that the instruction can load, depending on the instruction’s opcode.

During a backward traversal, the range of the loaded value is set for those instructions that use it, possibly reducing the conservative range assumed in the forward traversal.

5.3.5 Stores

Stores are ignored for forward traversals because they do not produce a result, whereas during backward traversals, the stored value may be constrained to a limited range depending on the store width specified by the opcode.

5.3.6 Branches

Unconditional branches do not provide information about value ranges. On the other hand, the comparison(s) upon which conditional branches depend are used to determine value ranges for each path. For example, consider the following code.

```
if (a<100) then {
 /* within the if condition */
} else {
 /* within the else condition */
}
```

In the "within the if condition" piece of code the maxium value of "a" is set to 100, and in the "within the else condition" piece of code minimum value of "a" is set to 101.

5.3.7 "Useful" Range Propagation

Some instructions constrain the range of the values due to their operations. Important cases follow.

- Logical operations. For example, AND R1, 0xFF, R2 (R2←R1&0xFF); OR R1, 0xFFFFFFFF00000000, R2(R2←R1|0xFFFFFFFF00000000).
- Mask operations (already present in the ISA)**.** For example, MSKBL R1, 0, R2 which extracts the least significant byte of R1 and copies it to R2, the rest of the bytes of R2 are set to zero.
- Limited width fields (I.e. shift amounts). For example, SRL R1, R2, R3, i.e. R3←R1>>R2 because the useful range for the shift amount is between 0 and 63.

A conventional VRP would assume that the range of input values is defined to include all possible runtime values. As noted in the introduction, we are only interested in the useful values, i.e. the ones that affect program results. So for example, in the case of the, AND R1, 0xFF, R2 our VRP method would backward propagate just the low order byte of R1 because all other bytes are set to 0 regardless of their previous value.

In other to ensure correctness when propagating "useful" range information, the technique must ensure there is no other point in the program where a wider range of values is used. In the AND example given above, if R1 is used somewhere else in the program execution and it has a wider range, the wider range is used despite the fact that in the AND operation just one byte is needed. Similarly, "useful" backward propagation through arithmetic instructions is disallowed in order to avoid hiding overflows; for example, in the case of a loop whose upper bound is unknown, a value within the loop is incremented, and only the first byte is used.

5.3.8 Example

Figure 43 is a simple example that illustrates value propagation. Numbers label the propagation steps. Steps 1 through 7 and 9 occur during forward propagation and the 8th step during backward propagation.

At step 6, the loop trip count is calculated according to the algorithm described below. *INTmin* and *INTmax* stand for the minimum and maximum possible values for an integer value; those are the default values for any integer value. In step 8, *a1in* refers to the input *a1* value. In the first step, *a0* is assigned the base address of the vector. Since it is an unknown value the range is set to the widest possible. In step two, *a1* is assigned a 0 and the range is set to the specific value. In step 3, *a3* is assigned the product of *a1* and 4. Because *a1* is 0, the multiplication results in 0. Then *a0* is added to *a3*, the sum is assigned the widest possible range of *a2*. Then there is a store, which does not change anything. Then *a1* is incremented (step 6). At this point, the loop is detected and the loop trip count is calculated (as it is explained in next section); then the value range of *a1*at the jump is set (step 7). Once the forward propagation is done, it starts the upward propagation. In step 8, the range of *a1* as an input value is set (according to the range of the output <1,100> and the increment). In the following and last step, the value range of *a3* is updated according to the new range of *a1*.

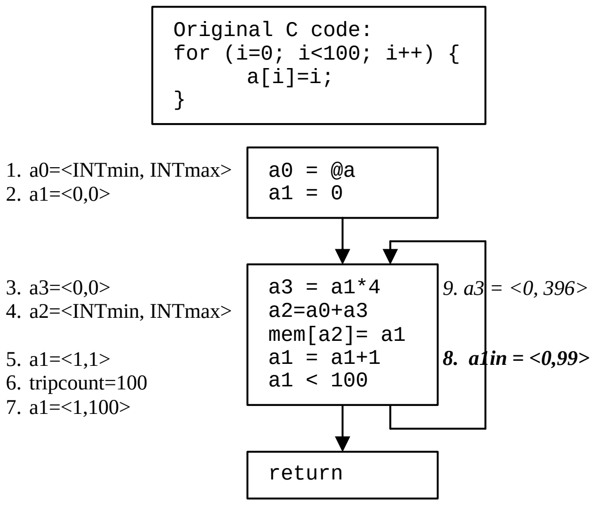


Figure 43. Example of value range propagation

5.3.9 Loops

Loops require special treatment. The range of values generated by instructions in a loop depends on the number of iterations. Consequently, a loop trip count bounding technique is implemented. This technique focuses on loops (typically "for" loops), where the iterator is of the form *x=ax+b* (where a and b are constants and x is the iterator). In this case, loops of the form: "`for (i=constant; i<constant; i=ai+b) ...`" permit the value range of i to be bound. Special cases may arise, like those loops having more than one iterator. Obviously, loops that traverse a list or any structures through pointers, which typically depend on a comparison to finish the traversal, are not subject to this technique and the trip

count may not be determinable at compile time. This introduces limitations on the technique, depending on the source code being analyzed.

Sometimes certain parts of a given loop are executed more or less often than others parts. In these cases, a detailed loop trip count for each section can often be computed. For example, given the loop:

```
for (i=0;i<100;i++) {
 if (i<50) then ...
 else ...
}
```

The number of times each region is executed can be determined because it depends directly on the loop trip count. Nevertheless, there might be cases where this “local” trip count might not be possible to know:

```
for (i=0;i<100;i++) {
 if (a[i]==0) then ...
 else ...
}
```

For a given section of a loop if the trip count, N, is known; the result range of executing an instruction for the N iterations can then be determined for those operations depending on the loop trip count (induction variable). If the trip count is partially known or not known at all, then a worst-case bound must be assumed in order to ensure the correctness of the code generated. In Section 5.3.8 an example of value range propagation using the loop trip count has been shown.

5.3.10 Interprocedural Analysis

Our implementation of VRP includes interprocedural analysis. In this case, all the values passed from one function to another through registers keep their range information. At a procedure entry point, the possible ranges of the registers are analyzed, and the most conservative range is calculated. For return values, the range of the value of returned register(s) is set, and any instruction reading this register will read the range information, too. Since value propagation through memory is not taken into account, as mentioned above, by-reference parameters do not have value range information.

5.4 Value Range Specialization

Value Range Specialization is a compile-time technique based on profiling. The technique has three steps:

1. It identifies instructions (candidates) where the specialization may be profitable.
2. Through profiling, the run-time range of the values of the possible candidates is computed.
3. Using the profile information, the candidates that are deemed profitable are specialized.

The first step defines the candidates for specialization. In other words, it defines the set of instructions that have most chances of being profitable specialization points. These candidates are then profiled. With the profiling information on the value ranges, each candidate is then evaluated. If the evaluation results positive (there are energy savings of specializing the candidate for a certain range), the candidate is set to be specialized. Finally the specialization phase clones the section of code being specialized; and it adds the tests for the specialized and propagates the new range to the specialized region.

5.4.1 Computing the Energy Savings of Specialization

The energy savings of specializing a given candidate are guided by estimates of the benefit obtained through the specialization of a given instruction and output register. For each single instruction, *InstSaving(I,r,min,max)* is the energy saved for the instruction when the input register *r* has a given value range [*min,max*]. *InstSaving* is calculated the following way: given the range of the input operands (on of which is *r*), the range of the output register is set; then, if the width of the output register has changed (meaning it may need a narrower opcode), the energy savings are computed depending on each instruction type (as Table 8 shows for ALU operations). These instruction-type dependant energy savings have been empirically defined for each instruction type and operand-width through the observation of its energy requirements (see Section 5.5.1 for the experimental framework). The energy savings for a given instruction *I* is denoted by *Savings(I,r)* and it is the energy saved for all dependant instructions on the ouput register of instruction *I* (which is *r*).

$$Saving(I,r) = \sum_{\forall D \in Uses(I,r)} \left[InstCount(D) x InstSaving(D,r,\min,\max) + Saving(D,r',\min',\max') \right]$$

Where *r’* is the output register of instruction D and its range is [*min’,max’*]. *Uses(I,r)* are all the instructions that use the output register *r* of instruction *I*. And, *InstCount(D)* is the number of times instruction *D* is executed.

The savings of each instruction type have been empirically computed by determining the energy-savings (in nano Joules) of a given instruction type with different operand width. Table 8 depicts the energy savings for ALU operations:

Table 8: Energy savings for ALU operations (in nJoules)

	Source Width			
Target Width	64	32	16	8
64	-	-1	-3	-6
32	1	-	-2	-5
16	3	2	-	-3
8	6	5	3	-

This function, *Savings(I,r)*, will be useful both when looking for candidates to profile and then, together with the profile data, to determine the estimations of the run-time savings.

5.4.2 Computing the Cost of Specialization

The benefits of specialization have to be weighted against the costs incurred due to the runtime tests. The cost of such tests depends on the actual range tested. For instance, if the minimum and the maximum of a given range are the same value just one comparison is needed, otherwise, two tests are needed (one to check the minimum and one to check the maximum). At the same time, due to the tests implemented in the alpha architecture, testing for a zero value can be done in one single instruction but testing for another value has two be done in two instructions.

In order to compute the cost in terms of energy, each instruction needed in the test is given an energy requirement in relation to its instruction-type (branches, comparisons, and additions).

$$InstCost(I,r) = Nbranches * CostBranch + NComparisons * CostComparison + NAdds * CostAdd$$

$$Cost(I,r) = InstCount(I) * InstCost(I,r)$$

Nbranches, NComparisons and *Nadds* are, respectively, the number of branches, the number of comparisons and the number of additions needed to perform the test and *CostBranch*, *CostComparison* and *CostAdd* is the energy budget for each of these instruction types.

5.4.3 Identifying the Candidates for Specialization

In order not to profile every single instruction in the program code, we attempt to identify candidates (instructions) that the specialization could yield benefits in terms of energy savings if the run-time range would be sufficiently skewed. In order to reduce the number of candidates, a minimum cost is assumed. The minimum cost is found when specializing for the range [0, 0], since it needs just a single comparison. Setting the minimum cost reduces the number of instructions profiled to those that –at least- can get a benefit over the minimum cost.

Given the set of candidates to be profiled, we use the scheme proposed by Calder *et al.* [12] to perform the profiling. This technique inserts a function in the program that is called at the profiling points and stores the value of the output register analyzed in a fixed-size table. If the value is already in the table, the count of that value is incremented. Otherwise, if the table is not full, the value is added. If the table is full the value is ignored. Periodically, the table is cleaned by evicting the least frequently used values from the table: this allows new values to enter the table. The total number of times the profiling point is executed is also kept in a separate counter. After the execution of the program and for each profiling point, a distribution of the run-time output values is available.

5.4.4 Specialization of the Candidates

The specialization is done in two steps. First, the profile data is analyzed and the set of candidates is reduced to those that produce benefits. Second, the program is transformed accordingly for those beneficial points.

The benefit of specializing is computed through the formulas presented before in this section. For each candidate and using the profile information the energy savings and the energy costs are computed. Specializing a given program instruction *I*, for a range of [*min,max*] of its output register *r* is worthwhile if the overall benefit given by the next expression is positive (greater than zero).

$$Savings(I,r) * Freq(\min,\max) - Cost(I,r)$$

Where *Freq(min,max)* is the frequency that the range of the value of *r* is within the range of the specialized region, in other words, the frequency that the program path will go through the specialized code.

There are two possible transformations of the code. In the case that the range [*min,max*] results in the specialization for a single value (i.e. *min=max*), a value specialization method is used. Otherwise, a value range specialization method is used. The value specialization method is based on the one proposed by Muth, *et al.* [50]. In the case of the value range specialization, the method is a variation of the single value specialization adapted to ranges.

The value range specialization consists basically of duplicating the regions of code that are affected by the specialization and, after that, inserting the tests to select the region that will be executed: either the specialized or the not specialized. The value range specialization technique introduces a set of comparisons to check that the value is between the limits of the specialized range. In order to minimize the costs of executing two conditional branches, the comparisons are implemented performing two comparisons and an *AND* operation and not through conditional branches. The condition tested is (*x=min && x=max*) where *x* is the value that is being specialized. After the tests are inserted, the value range propagation technique is run and thus the range of all the instructions in the specialized region is set.

5.5 Evaluation

5.5.1 Experimental Framework

To evaluate the proposed technique, we use an extended version of Wattch [8] for power analysis. The extensions include activity counts for all the blocks that allow data-width power gating. The main architectural parameters of the out-of-order machine are described in Table 7 in section 5.2.2. VRP and VRS were implemented in the Alto [42] binary optimizer. Some modifications where done in the optimizer by expanding the use-def algorithm to allow intra-basic-block and inter-procedural, forward and backward traversals. In order to implement the VRS, the profiling part of Alto was modified by inserting the capability of profiling value ranges, computing the cost/benefit equations in terms of energy and modifying the specialization function to insert the correct specialization code for value ranges. We used the programs from the SpecInt95 suite with their reference inputs (and train inputs to perform the profiling). All benchmarks were compiled with the HP-Alpha C compiler. The resulting binaries were optimized at the maximum level and post-processed with our binary optimizer in order to perform VRP. All benchmarks were run to completion.

5.5.2 Benefits of "Useful" Value Range Propagation

Figure 3 shows the distribution of the run-time instructions (on average for SpecInt95) according to the widths determined by value range propagation. The extended value range propagation technique that distinguishes useful values from all actual values (labeled in the figure as *Proposed VRP*) can identify more instructions with small operands than the *conventional VRP* –without "useful" value range propagation (see section 5.3.7) . Overall, the number of 64-bit instructions is reduced from a 51% to a 42%.

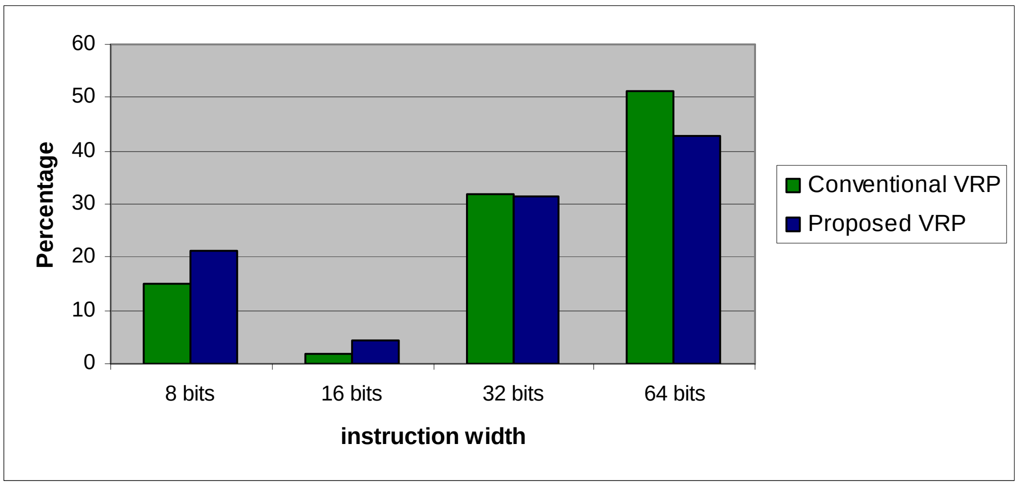


Figure 3: Dynamic instruction distribution according to value size.

5.5.3 Required Opcode Extensions

Depending on the initial instruction set, some new opcodes may have to be implemented to fully support the proposed VRP technique. In this section, we analyze extensions required for the Alpha ISA.

The technique as presented applies only to integer computations. Thus, floating-point operations are not analyzed. In addition, branch instructions are not taken into account because they manipulate addresses (i.e. wide data). The Alpha ISA already supports byte, halfword, word and double word memory operations (Load[2] and Store). In Table 9, the distribution of other instructions is presented, in order of their dynamic percentage of occurrence (for SpecInt95). The first column lists the operation type, the second column gives the percentage of dynamic instructions of the given type, and the remaining columns give the various data widths as a percentage of instructions *of that type*, i.e. as a percentage of the column 2 percentages. Hence, 24 percent of the ADD instructions operate on 8 bits, or about 6.65 percent of all dynamic instructions.

Table 9: Distribution of operation types

	Percentage of run-time instructions	64b	32b	16b	8b
ADD	27.66	58.04	14.37	10.98	24.03
MSK	5.18	35.50	13.41	13.57	37.63
CMP	3.78	6.18	7.79	20.53	64.84
SHIFT	2.75	29.91	19.56	22.90	32.23
SUB	2.35	13.87	15.76	16.82	60.78
AND	1.92	16.11	8.73	27.03	48.94
OR	1.79	23.77	5.90	3.53	68.62
XOR	1.15	17.04	7.52	29.77	43.89
CMOV	0.80	18.42	20.04	25.33	39.61
MUL	0.18	47.95	23.39	7.04	26.87

Because the MUL operation is rarely used and almost half the time it uses 64 bits anyway, there is no advantage to implementing narrow-width MUL instructions. Similarly, there are very few 16-bit operations overall. Only ADD is likely to be important enough to warrant a 16-bit version.

Overall, new opcodes added to the Alpha ISA are: byte and halfword addition; byte substraction; byte and word logical operations (and, or, xor) and; byte and word shifts, conditional moves and comparisons.

If some narrow data-width opcodes are not available for a chain of dependant instructions, value range propagation must ensure that the values read at run time contain significant data for all the input

[2] Although the byte and halfword load are unsigned it has no effect on the value range propagation.

bytes (meaning that unused leading bytes have a defined value –either zero or one). This will ensure that the wider instructions do not have undefined leading bits. For example, it would be unacceptable for the result of a 16-bit load (i.e. 0x0000FFFF) to be used as input to a 32-bit multiplication (i.e. 0xXXXXXXXX0000FFFF –where X indicates an undefined value).

5.5.4 Energy Savings

The VRP mechanism does not affect the performance of the benchmarks since it just reencodes the instructions with narrower opcodes. These narrower opcodes are then used to gate-off the portions of the datapath that are not relevant for the computation of the final results. Figure 44 shows the power savings of the processor when using the proposed VRP mechanism over the execution of the same binary without VRP. The power results for the rename logic, branch prediction, instruction cache and second level cache are not given because are not affected by the VRP. Nevertheless, the power consumption of these components is part of the “processor” column.

The power savings for the most data intensive structures is up to 18% (i.e. functional units), and for most other structures the savings are around 15% (instruction-queue, rename buffers, register file and the result busses). The memory management structures (LSQ and L1 data cache) exhibit a minor improvement since they handle memory addresses as well. Overall, the savings in these structures result in an overall energy savings close to 6% on average for the SpecInt95 suite.

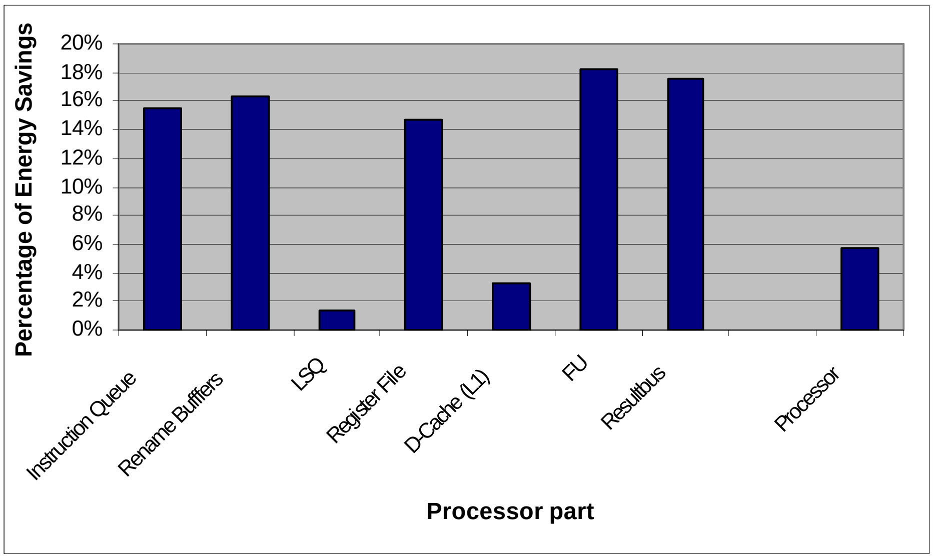


Figure 44: Power savings for the VRP

5.5.5 Potential Benefits of Value Range Specialization

Figure 45 shows the potential of the Value Range Specialization. The potential has been computed assuming no overhead for the specialization (i.e. extra comparisons and branches) and taking into account every instruction that has a narrower value range than that set by the VRP technique -at least, 90% of the times it is executed. The instructions are divided between the ones that the range is a single value or multiple values. The benefits of specializing for a given value are greater since some instructions may be eliminated through constant propagation. The baseline is the energy spent with the Value Range Propagation mechanism.

We can see in Figure 45 that the potential benefit of using profiling varies between a 9% and a 20% reduction of the total processor energy over the VRP mechanism. This means that the technique without profiling can achieve a good glimpse of the run-time values in most of the benchmarks and that the benefit of using profiling can be important for some benchmarks where the static approach cannot get a good result; in other words, there is still room for improvement for the VRS mechanism.

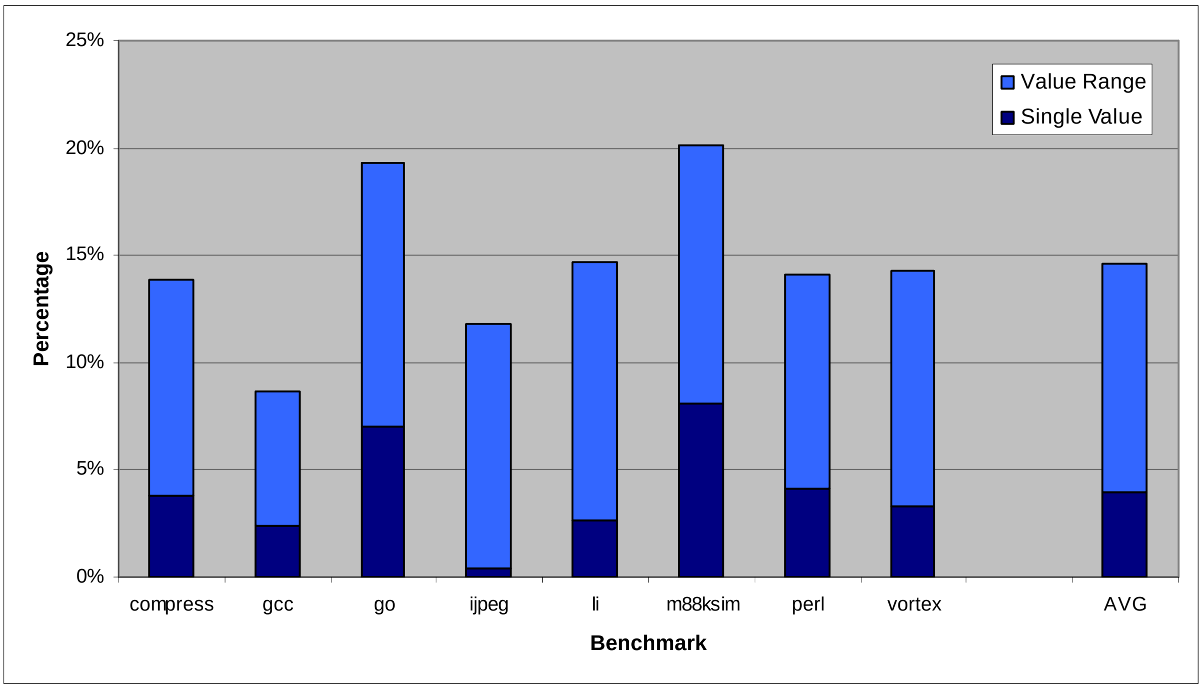


Figure 45: Potential energy savings of VRP+VRS

5.5.6 Benefits of Value Range Specialization

As explained in Section 5.4, the Value Range Specialization is a profiling-based technique. Figure 46 shows the distribution of the candidates (i.e. instructions profiled) when they are analyzed right before performing the specialization. The number on the top of each bar is the total number of candidates (instructions that may are a potential source for specialization) profiled for each benchmark. Several filters have been implemented in order to select only those candidates that result in an energy-benefit. As shown in Figure 46, most of the candidates are finally not considered for specialization since they produce no benefit (88%). The other reason to eliminate a candidate is that it is dependent on another candidate (since specializing one candidate will result in the specialization of the dependant one). On average this only happens for a 2% of the candidates. At the end, the number of specialized candidates is on average a 7% of the profiled ones. This means that on average, for the SpecInt95, 15 candidates are specialized per benchmark, and individual benchmarks range from 3 (perl) to 55 (gcc).

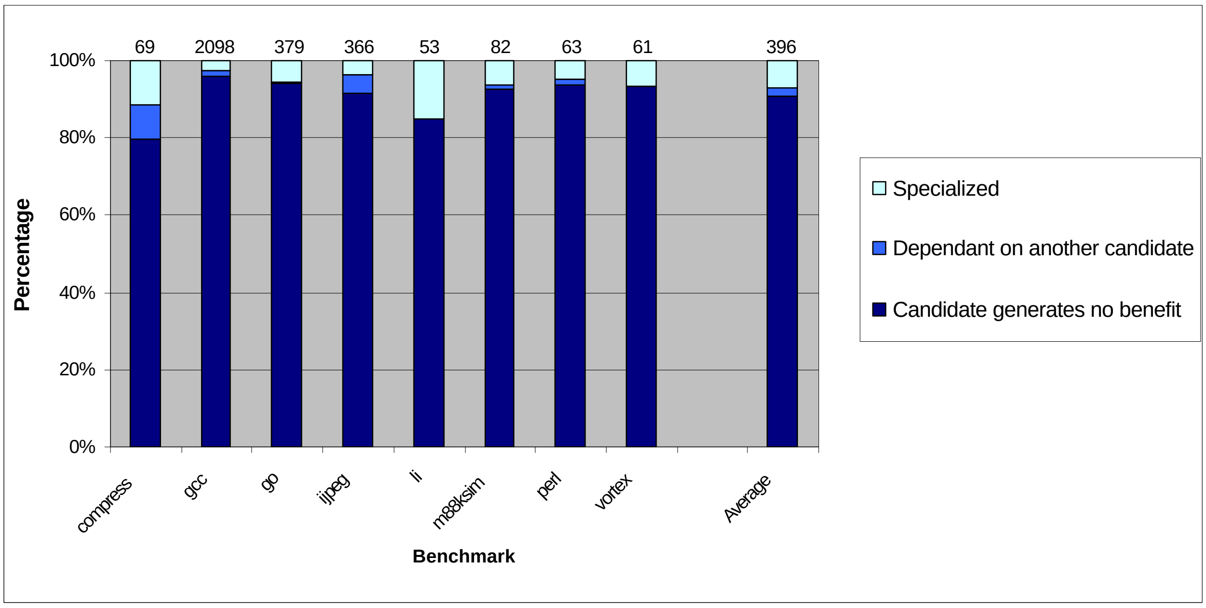


Figure 46: Distribution of the candidates profiled after specialization

The information about the number of specialized candidates is interesting to see the effectiveness of the profiling. At the same time, a more concrete result is the number of instructions that result from the specialization of these candidates. Figure 47 shows the distribution of the instructions specialized for each benchmark and on average. Although in most of the cases the instructions are specialized (in other words, a new and more concise range is set for these instructions), there is also a significant amount of instructions (especially in m88ksim and vortex) that are removed from the specialized sections of the code. Since the technique is capable of specializing for a given value, and because constant propagation is applied to the resulting code; some instructions can be eliminated.

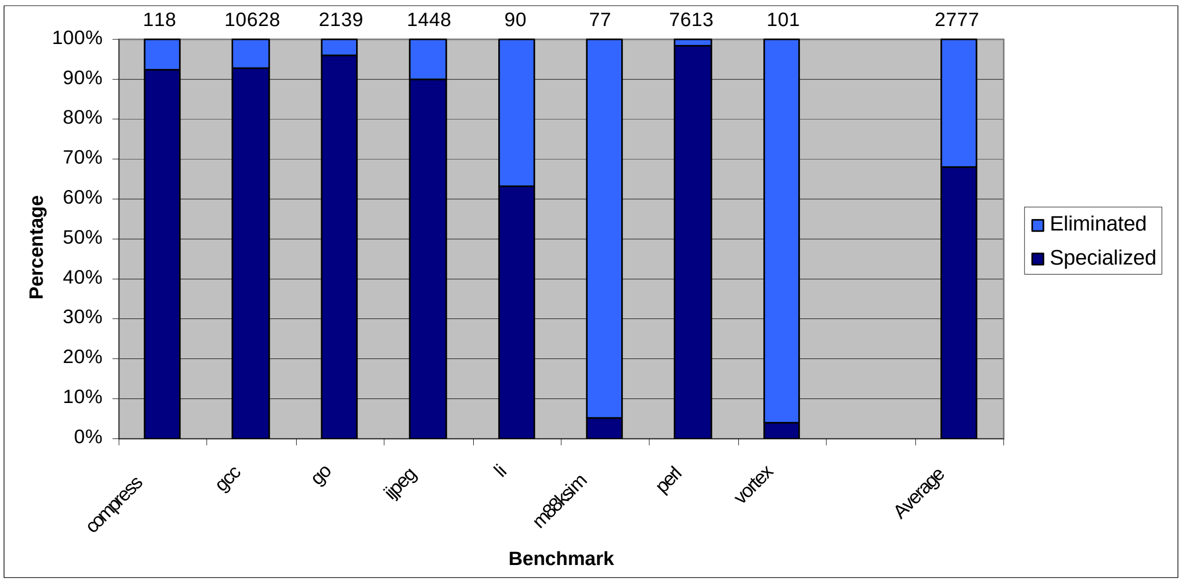


Figure 47: Distribution of the specialized instructions at compile-time

At run time, the distribution of specialized instructions is showed in the first column of each benchmark in Figure 48. At the same time, the percentage of instructions needed to specialize a point (comparisons, etc) is reported in the second column. As it was shown in Figure 47, m88ksim and vortex eliminate almost all the specialized instructions, which results in a minimal run-time occurrence of specialized instructions. On average, more than 15% of the executed instructions are specialized -with a maximum of 35% for perl; whereas the comparisons represent 1% of the executed instructions on average.

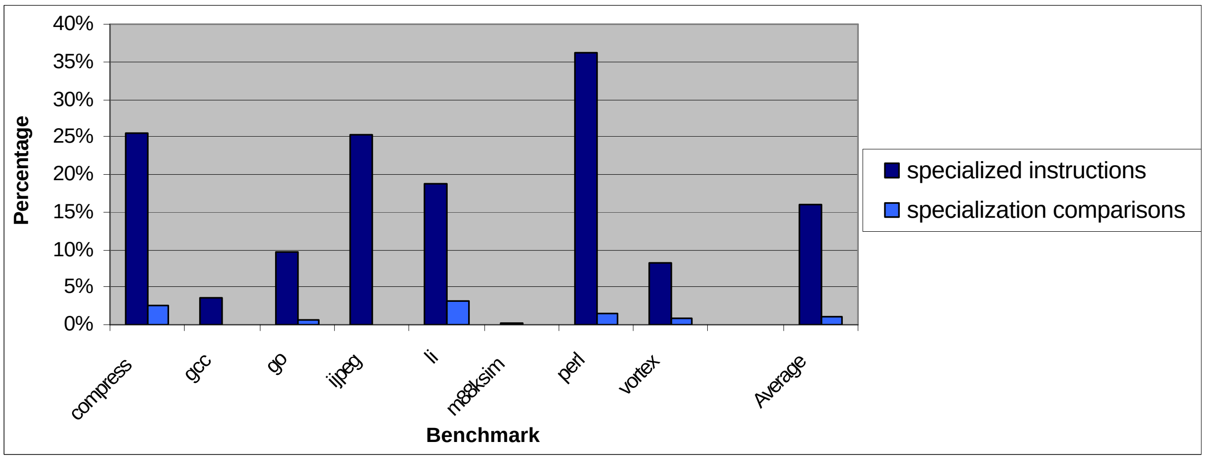


Figure 48: Distribution of the run-time instructions

Another interesting statistic is the distribution of run-time instructions through the different value range propagation mechanisms. Figure 49 shows the distribution of the run-time instructions on average for the SpecInt95. The first column is the baseline where no mechanism is implemented. In this case, most of the instructions deal with 64 or 32 bits. When Value Range Propagation is implemented, the amount of 64-bit instructions decrease to 40%, the percentage is further reduced to a 30% with the Value Range Specialization mechanism (VRS 50uJ, meaning that the cost of specialization is set to 50 nano Joules). This decrease in 64-bit instructions turns (mainly) into an increment in 8-bit instructions.

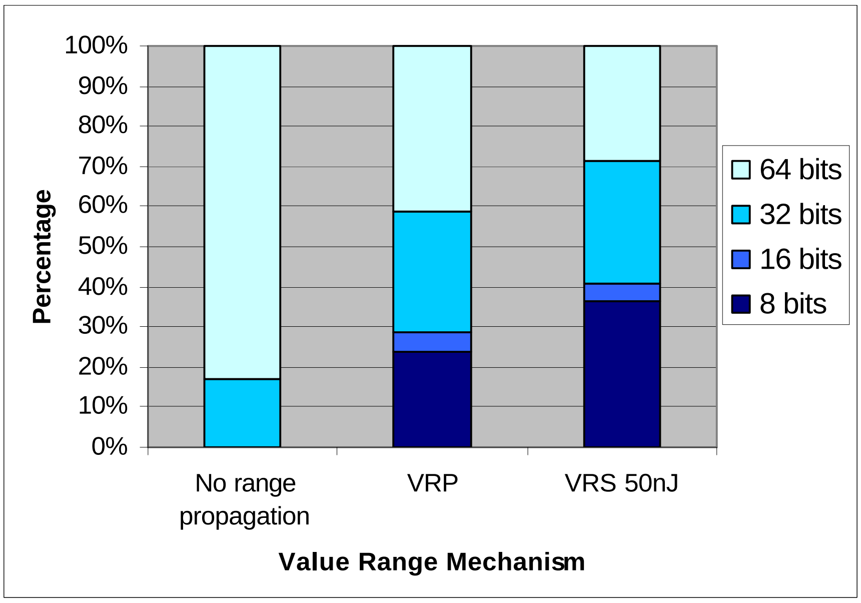


Figure 49: Distribution of the run-time instructions according to its execution size

Figure 50 shows the energy savings in relation to the baseline (the architecture without any value range mechanism). In the case of the VRS mechanisms, different configurations have been studied depending on the cost of specializing in terms of energy. As Figure 50 shows, the difference in cost does not turn into big differences in energy reduction. This suggests that the set of candidates profiled have a very skewed distribution according to the benefits they produce: either they are extremely good for specialization or they are extremely bad.

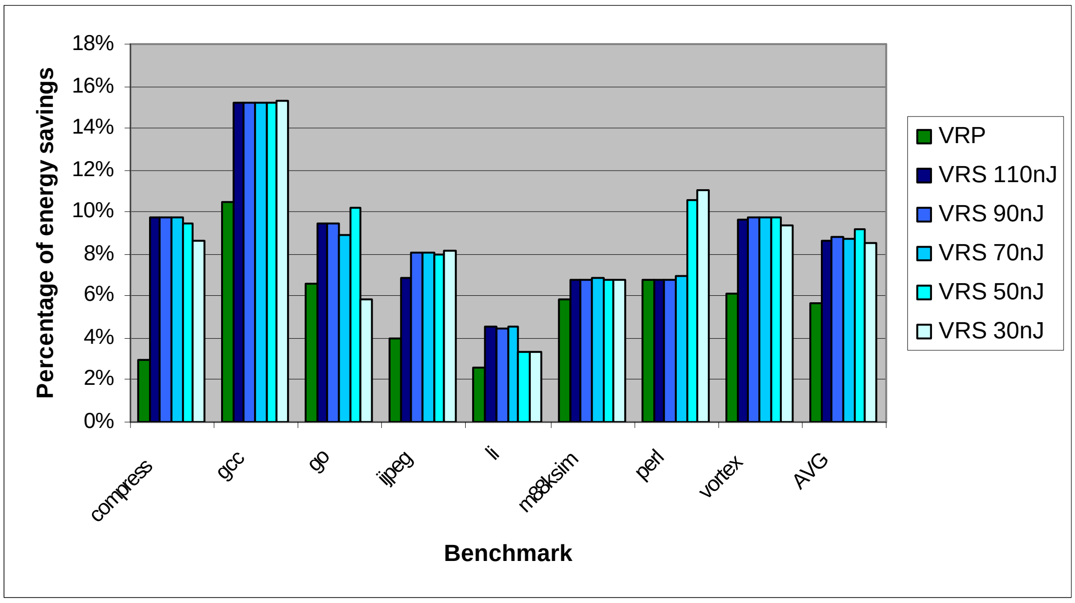


Figure 50: Energy savings for the Spec95

Detailed energy benefits for every part of the processor are shown in Figure 51. Due to the nature of the mechanism, the parts that most benefit of the value range mechanisms are those that directly manipulate data values (i.e. issue queue, rename buffers, register file, functional units and result busses) as it happened with VRP in Figure 44. Since VRS mechanisms manipulate code (by adding the comparisons to perform the specializations and removing instructions in the specialized section), all the parts of the processor are printed in Figure 51. Minimal energy benefits arise from the reduction of the instructions but more impressive benefits arise from the data-dependent structures –the ones the technique was focused on. Overall, the energy benefits of the VRS mechanism are around 9% while most of the data intensive structures are over 20% of energy savings.

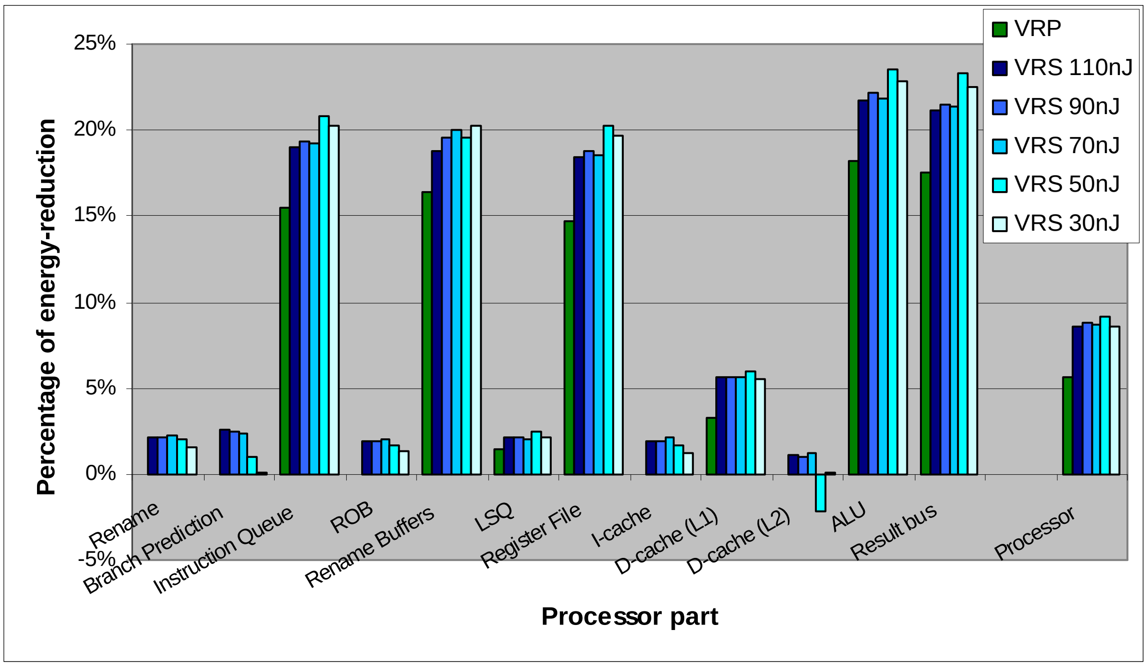


Figure 51: Energy benefits for the different parts of the processors

Since the code is modified by inserting the comparisons of the VRS and eliminating the instructions in the VRS sections of specialized code, it is important to see the impact on the execution time of the benchmarks with the new technique. Note that the VRP mechanism does not affect performance since it does not add any instruction to the code, it just recodes the instructions with narrower opcodes. Figure 52 shows the reduction in execution time. Except for one configuration of VRS in *go*, the rest of the binaries perform slightly better when VRS is included.

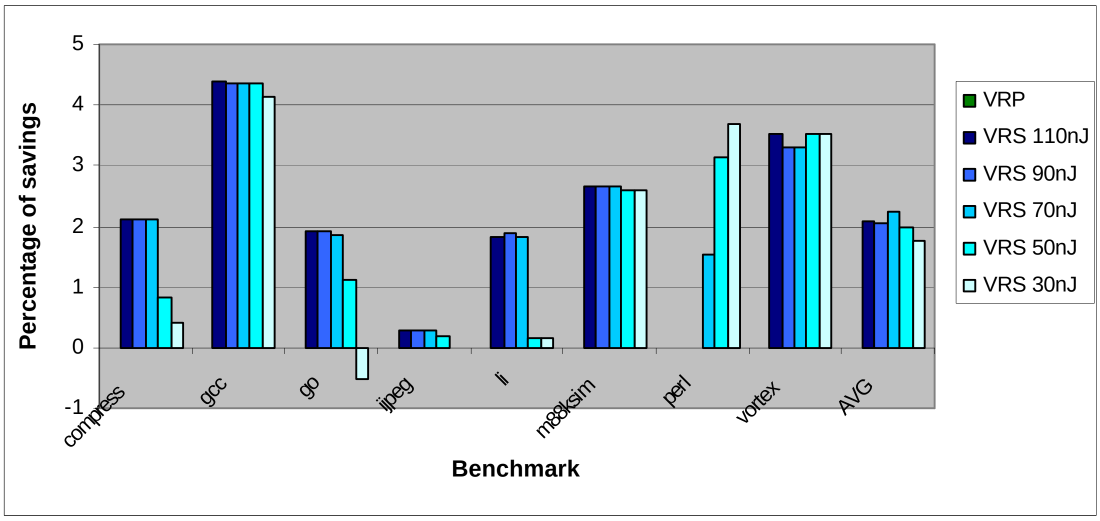


Figure 52: Execution time savings

In order to compare the benefits of using the VRS mechanism, the energy-delay^2 metric provides a fair comparison of all the design points taking into account both energy and execution time savings. Figure 53 shows the improving in energy-delay^2 for all the benchmarks in the SpecInt 95. On average the benefits of the VRP mechanism are a little bit above 5% but, when using the VRS mechanism, the benefits scale up to almost 15%. In the case of the VRS 90nJ mechanism, the benefit rises to 25% in the case of *gcc*.

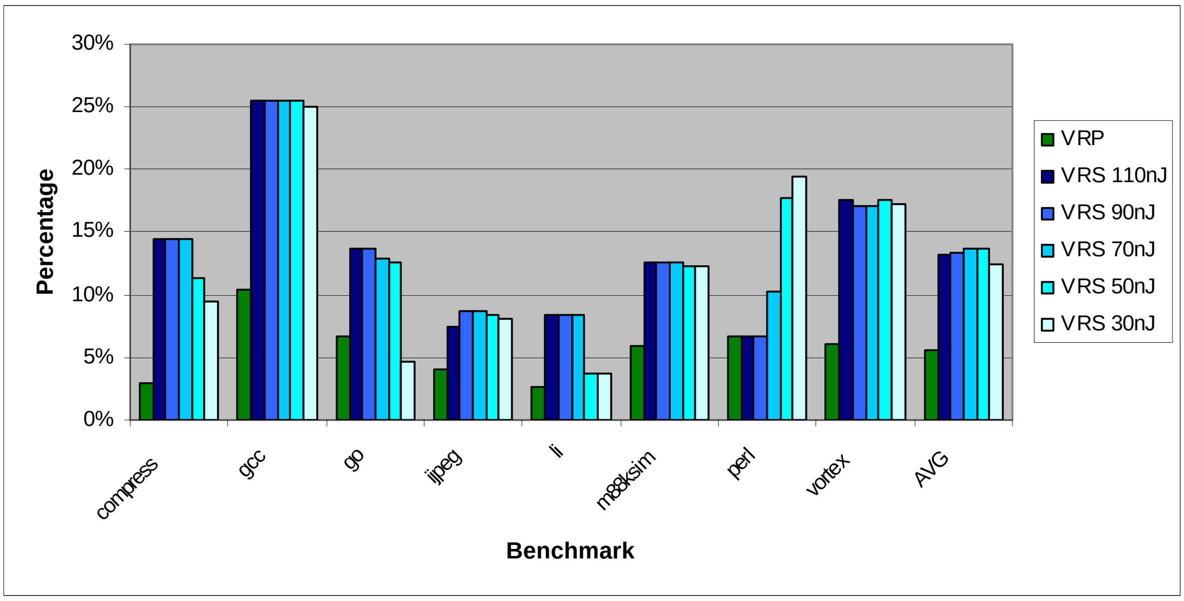


Figure 53: Energy-Delay^2 Product for the Spec95

5.5.7 Comparison with a Hardware Approach

To compare with a hardware approach, we use the mechanisms presented in Chapter 3. Significance compression is implemented through seven tag bits added per data word (64 bits) to indicate the number of significant bytes. For comparison purposes size compression is also evaluated. In this mechanism, two extra-bits are added per data word in order to indicate whether the value is 1, 2, 5 or 8 byte long.

Figure 54 shows the energy reduction of the different hardware approaches. On average, a 16% of the overall power is reduced. The hardware approach has the advantage of enabling multiple-size operands in the functional units. For example, an addition of a 16-bit plus a 32-bit value producing a 64-bit value could be possible. Furthermore, the data-width of the same static instruction for several different dynamic instances can be different (this being a big difference to the software schemes presented in this work). Overall, the hardware approach has more opportunities to reduce the power consumption despite the cost of keeping several bits per data word.

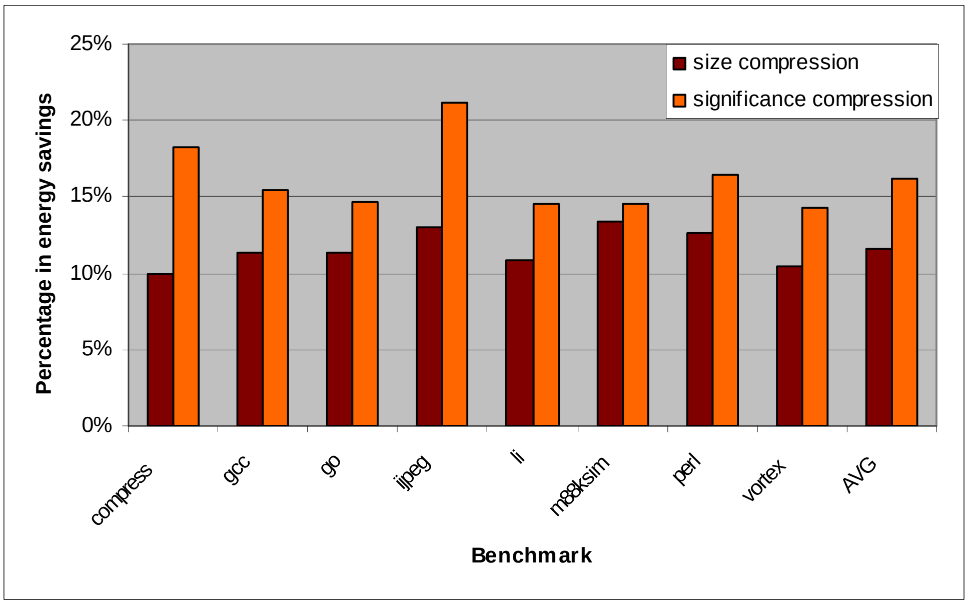


Figure 54: Energy savings for the different hardware approaches

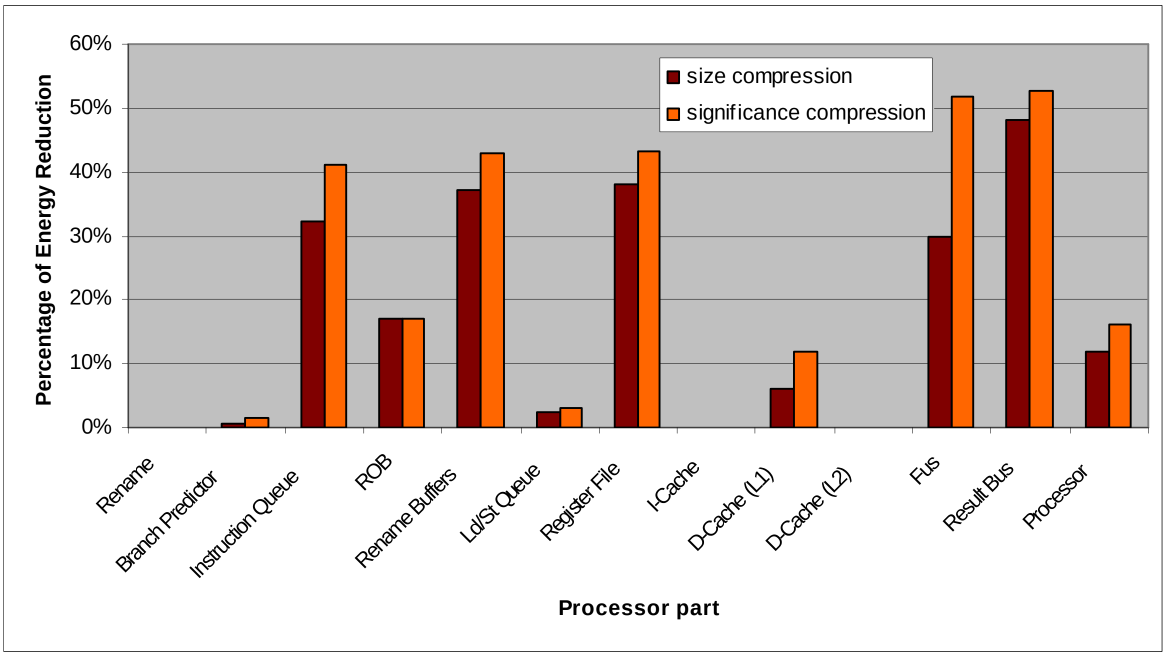


Figure 55: Energy savings for each processor part (Average SpecInt 95)

Figure 56 presents the energy delay savings when combining the hardware and software schemes. Both hardware and software schemes cooperate to achieve a higher percentage of energy-delay^2 savings. The energy-delay^2 savings of the VRS mechanism are (on average) very close to those of the *significance compression* mechanism and better than the *size compression* mechanism. In some benchmarks (gcc, perl and vortex), the VRS mechanism has a better performance in terms of energy-delay^2 than the hardware mechanisms. The reasons behind this competitive performance are the reduction of the execution time, the minimal extra cost (in hardware) of the software mechanism and the accuracy of the value range analysis. The benefit of using the profiling technique is even higher when a hardware scheme is used in the architecture since the additional energy savings of the hardware scheme are added to the reduction in execution time of the profiling mechanism. For instance, *gcc* achieves almost a 38% of energy-delay^2 benefit when combining the *significance compression* and the VRS.

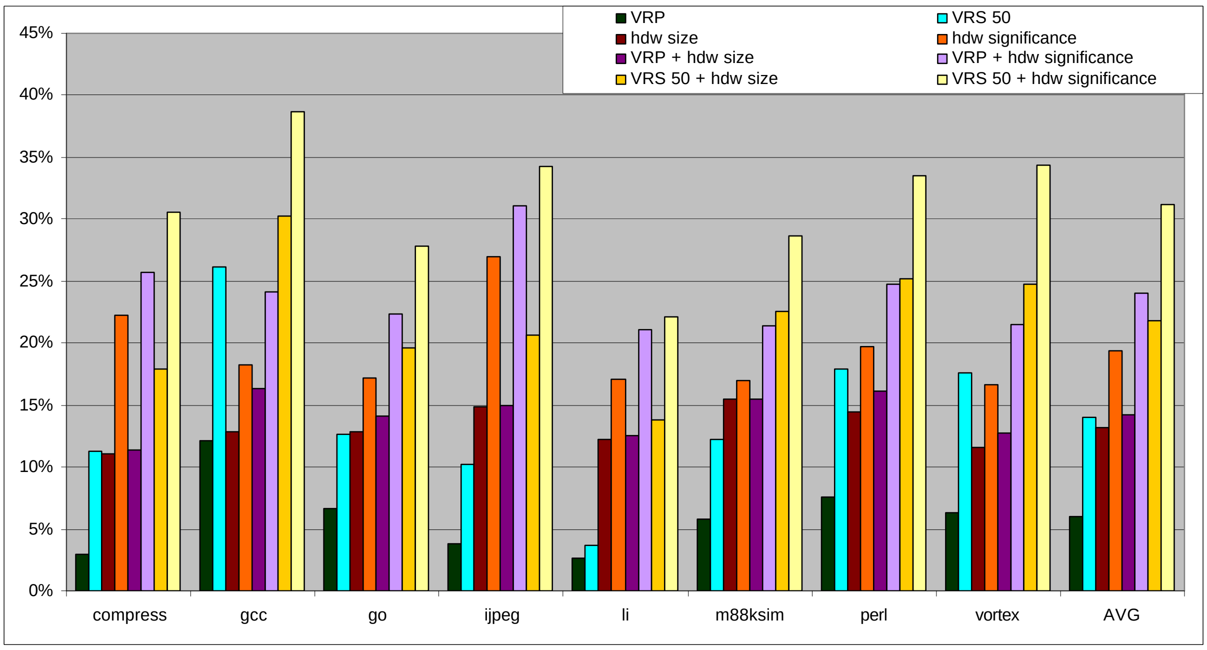


Figure 56: Energy Delay^2 savings for different hardware and software configurations

5.5.8 Hardware and Software Trade-offs

Both hardware and software approaches to operand gating have advantages and disadvantages. In general, the hardware approach requires no change in the ISA and no recompilation or binary translation. On the other hand, software approaches tend to make the microarchitecture simpler. Changes to microarchitecture in order to support the hardware mechanism are the following:

- A set of tag bits (2 or 7 as explained earlier in this section) that have to be stored with all values (in the register file and in the cache).
- The tag bits of a register value must be read before the data in order to know which bytes are significant and require a register file access. This may introduce a delay in the register file access.

- The functional units must be able to generate sign extended bytes for those values that need extra sign extension when a computation is performed (i.e. when adding a 16-bit value with a 32-bit one).
- The functional units must generate the tag bits for results.

The software approach encodes the width of the values used in the operation mnemonic and thus requires minimal hardware changes.

In terms of value range propagation, the proposed software approach can only identify and propagate useful value ranges through the static program code, and may eliminate significant data bits in the process if they are not required for semantically correct results. On the other hand, a dynamic hardware-based significance compression mechanism is able to detect value ranges more accurately than a static software method.

This suggests a *cooperative hardware-software* approach where both methods are used in the same processor. To implement this cooperative approach, value compression is initiated both through compiler-generated instructions and through hardware-generated compressed values at runtime. In both cases, two significance compression tag bits follow values in the pipeline. Although the compiler generates opcodes for 8, 16, 32 and 64-bit wide data, when a hardware compressing mechanism is used, the instructions executed will use 8, 16, 40 or 64 bits within the microarchitecture. The cooperative hardware and software approach achieves a 31% overall energy-delay^2 savings in relation to the base architecture (the architecture without any compile or run-time data compression).

Overall, the software scheme seems more appropriate to an environment where the ISA already supports multiple-size operations and where hardware overhead is unaffordable. On the other hand, for scenarios where recompilation is an issue and where hardware changes are not critical, the hardware approach may be best. Finally, only for environments where power is very critical, one may be willing to pay the overheads of a combined software and hardware approach in order to achieve the extra 10% power reduction that the combination provides over the hardware-only scheme.

5.6 Related Work

5.6.1 Processor Front-End

The primary functions performed in a processor's front end are instruction caching and branch prediction. Simple zero compression was proposed for the instruction cache [71], resulting in a 10% reduction in the energy consumption of the cache.

To the best of our knowledge there have been no published results on value compression to reduce energy requirements of branch prediction. However, in Section 5.2.3, we show performance figures of applying the zero compression mechanism of Villa *et al.* [71] and our significance compression method to branch predictors. The power savings during branch prediction comes from compressing values held in the branch target buffer (BTB).

Sato and Arita [60] split the value predictor structure that keeps the predicted values into two similar structures, where one holds byte-wide data and the other holds 64-bit data. This structure is shown to be beneficial for energy saving because most of the instructions' ouput-value widths do not change, and a large portion of them (as shown in the data width distribution in Figure 21) are narrow.

5.6.2 Processor Back-End

Brooks *et al* .[7], Loh [39] and Nakra *et al.* [51] propose techniques for exploiting narrow width operands to reduce functional unit energy requirements and, at the same time, to increase performance. Their techniques pack instructions that use narrow operands so that they can be executed in a single ALU (i.e. one 64-bit adder can compute four 16-bit additions). The differences between the various approaches lie in the ways the narrow widths are obtained. Brooks *et al.* introduce hardware that dynamically detects the width of operand values. Loh extracts the data-width from a data-width predictor and thus a recovery mechanism is needed in case the prediction is wrong. Finally Nakra *et al.* set the width at compile-time. In these works [7][39][51], the register file is modified in two possible ways: either by incrementing the number of read and write ports to the banks of the register file holding the low-order bytes; or by replicating the lower part of the register file.

The implications to the functional units result in two alternatives: Brooks [7], Loh [39] and Nakra [51] extend the functional units with the capability to execute multiple narrow-width instructions (see Figure 57a). On the other hand, as we proposed in the previous chapter, the functional units can be extended so that the FUs can operate with compressed values and generate the compression bits (see Figure 57b). In terms of implementation of these alternatives, Choi *et al*. [22] present several FU implementations that turn off the portions of the FU that compute the high-order bits when these are just a sign-extension of the least significant ones (the boundary between the high-order and low-order bits is analyzed and set in their work).

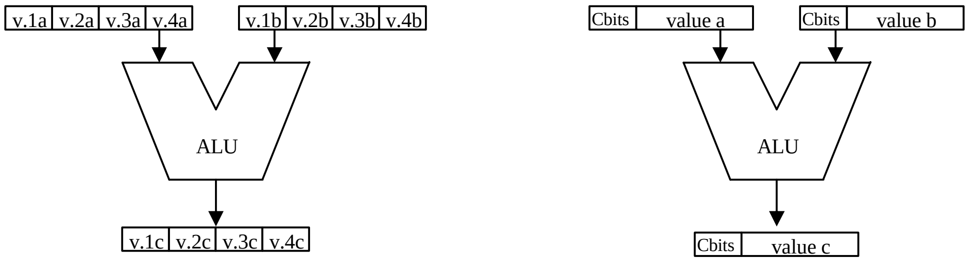


Figure 57: (a) ALU with packing capabilities, (b) ALU with value compression capabilities

5.6.3 Data Cache

Several value compression methods have been proposed for reducing energy consumption in the memory subsystem. Most of them are focused on on-chip caches. The data cache has been shown to be one of the more power-hungry structures in a microarchitecture. Figure 58 shows a data cache enhanced with value compression capabilities. Typical implementations compress and decompress the data when it is moved between the first and the second level caches. The same compression mechanisms can be used in all the memory hierarchy, and more sophisticated schemes [69] can be used in lower levels of the memory hierarchy for achieving higher compression ratios at the expense of some increase in latency -- not critical in lower memory levels.

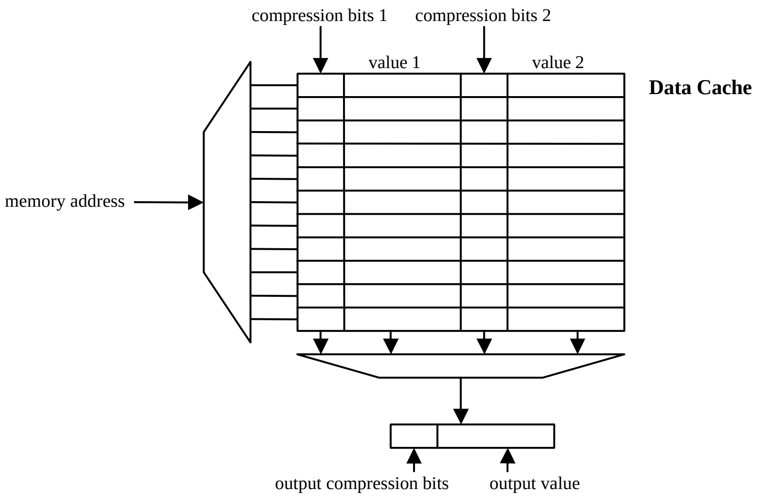


Figure 58: Data cache with value compression capabilities

Several compression mechanisms have been proposed: *zero compression* [71] eliminates the bytes that are set to zero; *active data-width* [52] compresses the values to certain ranges (6,14,24 or 32 bit); *frequent value cache* [76] has a list of most frequent values for the high-order bits (32 bits); and the last scheme analyzed is the *significance compression* that we have proposed in the previous chapter which eliminates the bytes that are a sign-extension of the previous one.

Villa *et al.* [71] propose an encoding where one bit per byte indicates whether this byte is null (zero) or not. The data-cache is modified to store the necessary bits. When the data is accessed, the additional bits are read first in order to just read the bytes that have a value different from zero. Okuma *et al.* [52] propose to divide the cache into several sub-banks where each sub-bank keeps a portion of the value (32-bit wide in their case). For each memory access, just the sub-banks with significant data are accessed. In their case, one sub-bank holds the lowest significant six bits, next sub-bank holds the following 8 bits, the third sub-bank keeps the next 10 and the last bank holds the last (most-significant) 12 bits. This compression scheme needs two bits per word and is very similar to the more general one analyzed in this paper under the name of *size compression.*

There are several other studies that use some properties of the values manipulated by the cache in order to save some energy. For instance, Yang and Gupta [76] propose to add an extra structure in the data cache that keeps the most frequent accessed high-order bits of the values in a compressed form. When a load is performed, the low-order bits bank is accessed, together with the bits indicating whether the remaining 32 bits are stored in a compressed form or not. If they are compressed, the frequent value bank is accessed; otherwise the high-order bits bank is used. This compression scheme adds 5 bits per data value when 16 frequent values are stored (i.e. $\log_2 n + 1$ bits, where n is the number of frequent values stored and one extra bit to indicate whether it is a frequent value or not). Moshnyaga *et al.* [46] propose a similar approach to the Frequent-Value Cache [76]. In particular, they propose to compress the values for the video-memory and just for the MPEG2 encoder. In this case the upper bits are not stored if they are the same as the previous value in the cache. One extra bit per byte indicates this situation. Finally, Park *et al.* [55] propose to invert the values when storing them in the cache. The values are stored either in their true or complemented value in order to minimize the transitions of the precharged value. One extra bit is necessary to differentiate between the true or complemented value.

The last work in this section refers to a proposal to reduce the energy consumption in the I/O pins. Musoll *et al.* [48] focus on the energy consumption of the pins communicating the processor cache to the off-chip memory subsystem. According to Mussoll’s study, most of the off-chip memory accesses have locality. Since most of the programs work on a few memory zones, each zone is given a code. Then, when performing an off-chip memory access, the zone code is sent along with the offset of the access to the previous reference instead of sending the whole memory address. In the same way, Benini *et al.* [3], Stan *et al.* [64] and Yang *et al.* [77] propose several new encodings for the values to save power in the I/O operations.

5.6.4 Software Controlled Value Compression

Value Range Propagation has been already explored in early stages of the compiler. Techniques using value range propagation have been used in high-level code transformations [9][40][56][65]. Our work applies at a much later code development step and to binary code. The technique proposed here is more CPU specific but totally compiler independent. For instance, forward propagation and loop analysis have some precedent in high-level program representations mainly used by symbolic analysis for different applications such as parallelizing compilers [4]. Other applications of the value range propagation include VLSI synthesis targeted to custom processors (i.e. that execute only a certain application) [21][40][65]. In [21] the authors propose a datapath-width optimization framework based on runtime information concerning the size of operands. This information is used to rewrite the source code where each data type has a width component. What we call *Useful Value Range Propagation* has not been previously implemented to the best of our knowledge; though Budiu *et al.* [9] implemented useful bit-width computation (where each bit was tagged whether it was useful or not). Alternatively, value range propagation has been used for branch prediction [56], although in this case backward propagation and loop-carried expressions were not considered.

Alternative approaches for narrow values are multimedia extensions like MMX-SSE, AltiVec and 3DNow!. These approaches are currently being most efficiently used directly by the programmers.

5.7 Conclusions

The compression of data values for different microarchitecture components has been shown to be an effective way of reducing the overall power consumption of processors. In this chapter, we have focused on the value compression paradigm and the proposals around this topic for high-performance processors. Hardware value compression has been shown to be a good method to reduce the activity and thus the energy consumption with no impact on performance.

Software operand gating has been shown to be an effective way to increase the processor energy efficiency. A software technique for operand gating has been evaluated. With minimal extensions to the ISA, the software approach is able to extract width information from the binary code and then propagate it through the program code. By extending the propagation through profiling to further constraint the value ranges, multiple versions of code for certain program segments are created and the most efficient is dynamically chosen based on the actual values of the operands. The proposed technique achieves an overall 14% energy-delay^2 reduction for the SpecInt95 set. It achieves a much better reduction for data-intensive structures where the energy benefits are over 20%.

Although the hardware schemes require some extensions to the microarchitecture, they can reduce the energy for any data that has a small number of significant bits. Since the values are checked dynamically at run-time, operand gating is optimized for each particular value. On the other hand, the software approach has to make conservative assumptions for two reasons. First, it has to assume the worst-case range when the compiler does not know the potential values of a variable. Second, it uses a unique range for each static instruction that includes all the values of the corresponding dynamic instances of the static instruction. However, a software scheme has a number of advantages. For instance, software analysis can detect useful bits, which in general are more restrictive than significant bits. This suggests that a combined hardware-software approach can further reduce the energy consumption. Our experiments show an average energy-delay^2 benefit of 31%.

"Human subtelty will never devise an invention more beautiful, more simple or more direct than does Nature, because in her inventions, nothing is lacking and nothing is superfluous"
Leonardo DaVinci

Chapter 6

Conclusions and Future Work

This chapter summarizes the main conclusions of this work and outlines some future work.

6.1 Conclusions

Several low power techniques have been proposed and analyzed in this thesis either specifically for one processor structure (issue logic) or for the whole datapath. We have investigated different alternatives that reduce the associative structures and increase the power savings.

In the case of the issue logic, we proposed alternative implementations that are much simpler and retain most, if not all, the ability of the full out-of-order mechanism to exploit instruction-level parallelism.

In particular, we have studied two families of schemes. The first one based on the so called "dependence-tracking" family presents the N-use scheme where the dependences between instructions are kept in a table indexed by physical register. Another alternative is the "prescheduling" family. Both the Distance scheme and the Deterministic Latency scheme presented provide a performance very close to the out-of-order processor and at the same time they constrain the associative searches to a small queue of 8-16 entries.

Chapter 3 showed the compressibility of the values manipulated in the pipeline. Three value compression methods and their reduction in the average number bits needed to be computed were presented. From the study in Chapter 3, two different processor segments were differentiated: ultar-low power and high-performance ones. Chapter 4 has shown how value compression can be applied to ultra-low power processors through redesigning the datapath. Since the significant bytes of instructions, addresses, and data values essentially determine a minimal activity level that is required for executing a program (typically 30-40% lower than for a conventional 32-bit pipeline), we developed several pipeline implementations that attempt to achieve these low activity levels while providing a reasonable level of performance. The byte-serial pipeline is very simple hardware-wise, but increases CPI dramatically. For some very low power applications, this may be an acceptable performance level, in which case the byte-serial implementation would be a very good design choice. For higher performance, the pipeline stages can be widened. An analysis of bandwidths suggests a balanced design. Activities are still at their reduced levels, and this design may provide a very good design point for many very low power applications. Finally, we considered designs with a four byte wide datapath at each stage. Operand-gating is retained for reducing activity, but under ideal conditions throughput is no longer restricted. These designs can come very close in performance to the baseline 32-bit design while again retaining reduced activity levels. The disadvantage of these schemes is an increased latch activity, or additional forwarding paths or more complex control. We believe that these may be a very important class of implementations however, because of their high performance levels.

Finally, in Chapter 5 we showed how software-controlled operand-gating is an effective way to increase the processor energy efficiency. With minimal extensions to the ISA, the software approach is able to extract width information from the binary code and then propagate it through the program code. By extending the propagation through profiling to further constraint the value ranges, multiple version of code for certain program segments are created and the most efficient is dynamically chosen based on the actual values of the operands.

Hardware value compression methods were also implemented in Chapter 5. While hardware schemes require some extensions to the microarchitecture, they can reduce further the energy since the values are checked dynamically at run-time and operand gating is optimized for each particular value. On the other hand, the software approaches have to be conservative since either they do not have all the information to take a decision or they have to assume the widest range possible of all run-time instances. Combining the hardware and software approaches resulted in a benefit of 31% in terms of energy-delay^2.

Overall, the thesis began with some insight into the issue logic complexity. Afterwards, we focused on wider-scope techniques based on novel value compression methods for the whole datapath. Starting in the ultra-low power segment, we proposed several microarchitrectures that achieved a minimal activity, and thus, energy consumption. After this work, we targeted high-performance processors. In this scenario, the compression overheads have to be kept to a minimum to minimize the impact on performance. Two compile time techniques have been proposed to enhance the benefits of value compression, either based on just static analysis or with the help of profiling information. We have shown that the compiler can play an important role when trying to reduce the energy requirements.

6.2 Future Work

Due to the increasing trend to miniaturization, several research topics arise that are interesting follow-ups of this thesis. Power and energy related issues are becoming one of the more intensive research areas. Thus, important effort should be directed to study new techniques and methods that can mitigate the restrictions that power and energy-related issues have on future microprocessors. For instance leakage currents and temperature-aware computing are still mainly unexplored and we believe they may be important in the future.

6.2.1 Leakage Current

As pointed out in the introduction of this thesis, for future technologies static power consumption will likely become the main component of the energy consumption equation. In this sense, future works should be targeted to reduce the leakage currents effectively.

In this case, the energy consumed is proportional to the number of devices (transistors) in the chip. Thus, techniques that reduce switching activity might be less effective in this environment and techniques that power down sections of the chip (or structures) might be an interesting focus of our research. At the same time, compile-time techniques will be a good choice since they have a smaller hardware impact because they usually require minimal hardware modifications (as it was shown in Chapter 5, for high-performance value compression methods).

6.2.2 Temperature-Aware Computing

Temperature-aware computing has started to become an interesting research for microarchitects. Given the scenario where the cooling and the packing costs set the temperature limit at which the processor can operate, it is an interesting research to evaluate performance- and energy-wise techniques that yield the best performance/energy requirements for this scenario.

It presents a challenge in terms of simulation since many implementation parameters have to be taken into account. An interesting start in this field of research is how thermal modeling can be implemented into a conventional simulator and then validated. Simulating temperature-wise behavior of a processor is a challenge, but the benefit of knowing such information and devising techniques that can increment the efficiency of the design may be deemed profitable in the near future.

6.2.3 System on a Chip

Another interesting scenario is the one that is being developed nowadays for mobile computing and it might become the standard technology in the near future for mobile devices such as mobile phones. In this scenario, a chip includes the processor, the memory, and several components for communication protocols (i.e. GSM, GPRS, Bluetooth, etc.).

Computer architects should work on these designs as well. Although it may be claimed that the inclusion of several functions in the chip is not a novel idea; the fact that they include communication units, and an increasing demand of multimedia processing makes it a very interesting field to research on and energy conservation techniques might be important for such devices.

References

[1] J. Abella, R. Canal and A. González, “Power- and complexity-aware issue-queue designs”, IEEE Micro, Volume 23, Issue: 5, pp. 50–58, September-October 2003.

[2] C.S. Ananian. “The Static Single Information Form”. Technical Report MIT-LCS-TR801, Massachusetts Institue of Technology, 1999.

[3] L. Benini, G.D. Micheli, E. Macii, M. Poncino and S. Quer, “Reducing Power Consumption of Core Based Systems by Address Bus Encoding”, IEEE Transactions VLSI Systems, Volume 6, Issue 4, pp. 554-562, 1998.

[4] W.J. Blume, “Symbolic Analysis Techniques for Effective Automatic Parallelization”, Ph.D. Thesis University of Illinois at Urbana-Champaign, 1995.

[5] M.T. Bohr, "Interconnect Scaling - The Real Limiter to High Performance VLSI", in Proceedings of the 1995 IEEE International Electron Devices Meeting, pp. 241-244, 1995.

[6] S. Borkar, “Design Challenges of Technology Scaling”, IEEE Micro Volume 19, Issue: 4, pp. 23-29, July-August 1999.

[7] D. Brooks and M. Martonosi, “Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance”, in Proceedings of 5th. International Symposium on High-Performance Computer Architecture, 1999.

[8] D. Brooks, V. Tiwari and M. Martonosi, “Wattch: A Framework for Architectural-Level Power Analysis and Optimization”, in Proceedings of the 27th Annual International Symposium on Computer Architecture, June 2000.

[9] M. Budiu, S. Goldstein, M. Sakr and K. Walker. “BitValue inference: Detecting and exploiting narrow bitwidth computations”. In Proceedings of the 2000 European Conference on Parallel Computing, August 2000.

[10] D. Burger, T.M. Austin, S. Bennett, "Evaluating Future Microprocessors: The SimpleScalar Tool Set", Technical Report CS-TR-96-1308, University of Wisconsin-Madison, 1996.

[11] G. Cai and C.H. Lim, “Architectural Level Power/Performance Optimization and Dynamic Power Estimation”, in the Cool Chips tutorial of the 32nd International Symposium on Microarchitecture 1999.

[12] B. Calder, P.Feller and A. Eustace, “Value Profiling”, in Proceedings of the 30th International Symposium on Microarchitecture, December 1997.

[13] R. Canal, A. González. “A Low-Complexity Issue Logic”. Proceedings of the 2000 International Conference on Supercomputing, pp. 327-335, May 2000.

[14] R. Canal, A. González. “Reducing the Complexity of the Issue Logic”. Proceedings of the 2001 International Conference on Supercomputing, pp. 312-320, June 2001.

[15] R. Canal, A. González and J.E. Smith, “Power-Aware Computing through Value Compression”, Technical Report UPC-DAC-2003-32.

[16] R. Canal, A. González and J.E. Smith, “Software-Controlled Operand-Gating”, in Proceedings of the 2004 International Symposium on Code Generation and Optimization, March 2004.

[17] R. Canal, A. González and J.E. Smith, “Very Low Power Pipelines using Significance Compression”, in Proceedings of the 33rd International Symposium on Microarchitecture, pp. 181-190, December 2000.

[18] R. Canal, J.M. Parcerisa, A. González, "Dynamic Cluster Assignment Mechanisms", in Proceedings of the International Symposium on High-Performance Computer Architecture, pp. 133-142, 2000.

[19] R. Canal, J.M. Parcerisa, A. González, "Dynamic Code Partitioning for Clustered Architectures", International Journal of Parallel Programming, Volume 29 Issue 1 , pp. 59-79, February 2001.

[20] R. Canal, J.M. Parcerisa, A. González, "A Cost-Effective Clustered Architecture”, Proceedings of International Conference on Parallel Architectures and Compilation Techniques, October 1999.

[21] Y. Cao, H. Yasuura, “A System-Level Energy Minimization Approach Using Datapath Width Optimization”, in Proceedings of the International Symposium on Low Power Electronics and Design, August 2001.

[22] J. Choi, J. Jeon and K. Choi, “Power Minimization of Functional Units by Partially Guarded Computation”, in Proceedings of the 2000 International Symposium on Low Power Electronics and Design, pp. 131-136, August 2002.

[23] J.Ll. Cruz, A. González, M. Valero, N. Topham, "Multiple-Banked Register File Architectures" in Proceedings of the 27th International Symposium on Computer Architecture, 2000.

[24] K.I. Farkas, P. Chow, N.P. Jouppi, Z. Vranesic, "The Multicluster Architecture: Reducing Cycle Time through Partitioning", in Proc of the 30th. Annual Symposium on Microarchitecture, pp. 149-159, 1997.

[25] J.A. Fisher, "Very Long Instruction Word and ELI-512", in Proceedings of the 10th Symposium on Computer Architecture, pp. 140-150, 1983.

[26] M. Franklin, "The Multiscalar Architecture", Ph.D. Thesis, Technical Report TR 1196, Computer Sciences Department, Univ. of Wisconsin-Madison, 1993.

[27] D. Folegnani, A. Gonzalez, “Reducing Power Consumption of the Issue Logic” Workshop on Complexity-Effective Design, Vancouver, June 2000.

[28] R. Gonzalez and M. Horowitz, “Energy Dissipation in General Purpose Microprocessors”, IEEE Journal of Solid-State Circuits, Volume 31 Issue 9, pp. 1277-1284 September 1996.

[29] M.K. Gowan, L.L. Biro, D.B. Jackson, “Power Considerations in the Design of the Alpha 21264 Microprocessor”, Proceedings of the 35th ACM/IEEE conference on Design Automation Conference, pp. 726-731, 1998.

[30] W. Harrison. “Compiler Analysis of the Value Ranges for Variables”. IEEE Transactions of Software Engineering 3:243-250, May 1977.

[31] D.S.Henry, D.C.Kuszmaul, G.H.Loh, R.Sami, “Circuits for Wide-Window Superscalar Processors”, Proceedings of the 27th International Symposium on Computer Architecture, pp. 236-247, June 2000.

[32] R. Iris Bahar and S. Manne, “Power and Energy Reduction via Pipeline Balancing”, in Proceedings of the 28th Annual International Symposium on Computer Architecture, June 2001.

[33] G.A. Kemp, M. Franklin, "PEWs: A Decentralized Dynamic Scheduler for ILP Processing", in Proceedings of the International Conference on Parallel Processing, Volume 1, pp 239-246, 1996.

[34] K.D. Kissell, “MIPS16: High-density MIPS for the Embedded Market”, SGI MIPS group, 1997.

[35] K. Knobe and V. Sarkar. “Array SSA form and its use in Parallelization”, in Proceedings of the 25th International Symposium on Principles of Programming Languages, pp 107-120, January 1998.

[36] M. Kozuch and A. Wolfe, “Compression of Embedded Systems Programs”, in Proceedings of the International Conference on Computer Design, 1994.

[37] C. Lee, M. Potkonjak and W. H. Mangione-Smith, “Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems”, in Proceedings of the 30th International Symposium on Microarchitecture, pp. 330-335, December 1997.

[38] C.R Lefurgy, E.M Piccininni and Trevor N Mudge, “Evaluation of a High Performance Code Compression Method”, in Proceedings of the 32nd International Symposium on Microarchitecture 1999.

[39] G. Loh, “Exploiting Data-Width Locality to Increase Superscalar Execution Bandwidth”, in Proceedings of the 35th International Symposium on Microarchitecture, pp. 395-405, November 2002.

[40] S. Mahlke, R. Ravindran, M. Schalnsker, R. Schreiber and T. Sherwood, “Bitwidth Cognizant Architecture Synthesis of Custom Hardware Accelerators”, IEEE transactions on Computer-Aided Design of Integrated Circuits and Systems, Volume 20 Issue 11, pp. 1355 -1371, November 2001.

[41] S. Manne, A. Klauser and D. Grunwald, “Pipeline Gating: Speculation Control for Energy Reduction”, in Proceedings of the 25th International Symposium on Computer Architecture, pp.132-141, June 1998.

[42] P. Marcuello, A. Gonzálex, J. Tubella, "Speculative Multithreaded Processors", in Proceedings of the International Conference on Supercomputing, pp. 77-84, 1998.

[43] D. Matzke, "Will Physical Scalability Sabotage Performance Gains", IEEE Computer Volume 30, Issue 9, pp.37-39, 1997.

[44] P. Michaud, A. Seznec, “Data-Flow Prescheduling for Large Instruction Windows in Out-of-Order Processors”, in Proceedings of the 7th International Symposium on High-Performance Computer Architecture, pp. 27-36, 2001.

[45] J. Montanaro and et. All. “A 160-MHz, 32-b, 0.5 W CMOS RISC Microprocessor”, Digital Technical Journal, volume 9. DEC, 1997.

[46] V. Moshanyaga, K. Inoue and M. Fukagawa, “Reducing Energy Consumption of Video Memory by Bit-Width Computation”, in Proceedings of the 2002 International Symposium on Low Power Electronics and Design, pp. 142-147, August 2002.

[47] E. Musoll, “Predicting the usefulness of a block result: a micro-architectural technique for high-performance low-power processors”, in Proceedings of the 32nd International Symposium on Microarchitecture 1999.

[48] E. Musoll, T. Lang and J. Cortadella, “Exploiting the locality of memory references to reduce the address bus energy”, in Proceedings of the 1997 International Symposium on Low Power Electronics and Design, pp. 202-206, August 1997.

[49] R. Muth, S. Debray, S. Watterson and K. De Bosschere “Alto: A Link-Time Optimizer for the Compaq-Alpha”, Software Practice and Experience 31:67-101, January 2001.

[50] R. Muth, S. Watterson, S. Debray, “Code Specialization based on Value Profiles”, in Proceedings 7th International Static Analysis Symposium, June 2000.

[51] T. Nakra, B. Childers, and M.L.Soffa, “Width Sensitive Scheduling for Resource Contained VLIW processors”, Workshop on Feedback Directed and Dynamic Optimizations, December 2001.

[52] T. Okuma, Y. Cao, M Muroyama and H Yasuura, “Reducing Access Energy of On-Chip Data Memory Considering Active Data Width”, in Proceedings of the 2002 International Symposium On Low Power Electronics and Design, pp. 88-91, August 2002.

[53] S. Önder, R. Gupta, "Superscalar Execution with Dynamic Data Forwarding", in Proceedings International Conference on Parallel Architectures and Compilation Techniques, pp. 130-135, 1998.

[54] S. Palacharla, N.P. Jouppi, and J.E. Smith, "Complexity-Effective Superscalar Processors", in Proceedings of the 24th. International Symposium on Computer Architecture, pp 1-13, 1997.

[55] B.I. Park, Y.S. Chang and C.M. Kyung, “Conforming Inverted Data Store for Low Power Memory”, in Proceedings of the 1999 International Symposium on Low Power Electronics and Design, pp. 91-93, August 1999.

[56] J. Patterson. “Accurate Static Branch Prediction by Value Range Propagation”. In Proceedings of the Conference on Programming Languages Design and Implementation, pp 67-78, June 1995.

[57] PowerPC 405CR User Manual”, IBM/Motorola, June 2000.

[58] C. Price, “MIPS IV Instruction Set”, MIPS Technologies Inc, 1995.

[59] E. Rotenberg, Q. Jacobson, Y. Sazeides and J.E. Smith, "Trace Processors", in Proc of the 30th. Annual Symposium on Microarchitecture, 1997.

[60] T. Sato and I. Arita, “Table Size Reduction for Data Value Predictors by Exploiting Narrow Width Values”, in Proceedings of the 2000 International Conference on Supercomputing, pp.196-205, May 2000.

[61] Semiconductor Industry Association, "The National Technology Roadmap for Semiconductors", 1997.

[62] J.E. Smith, G.S. Sohi. “The Mircoarchitecture of Superscalar Processors”, Proceedings of the IEE, Volume 83, Issue 12, pp. 1609-1624, December 1995.

[63] G.S. Sohi, S.E. Breach, and T.N. Vijaykumar, "Multiscalar Processors", in Proceedings of the 22nd International Symposium on Computer Architecture, pp. 414-425, 1995.

[64] M.R. Stan and W.P. Burleson, “Bus-Invert Coding for Low-Power I/O”, IEEE Transactions on VLSI Systems, Volume 3, Issue: 1, pp. 49-58, March 1995.

[65] M. Stephenson, J. Babb and S. Amarasinghe, “Bitwidth Analysis with Application to Silicon Compilation”, in Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, pp. 108-120, 2001.

[66] R.M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units", IBM Journal of Research and Development, Volume 11, pp. 25-33, 1967.

[67] D.M. Tullsen, S.J. Eggers, H.M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism", in Proceedings of the International Symposium on Computer Architecture, pp. 392-403, 1995.

[68] J. Turley, “Thumb Squeezes Arm Code Size”, Microprocessor Report, Volume 9, Issue 4, March 1995.

[69] J. Turley, “PowerPC Adopts Code Compression”, Microprocessor Report, October 1998.

[70] N. Vijaykrishnan, M. Kandemir, M.J. Irwin, S.H. Kim and W. Ye, “Energy-Driven Integrated Hardware-Software Optimizations Using SimplePower”, in Proceedings of the 27th International Symposium on Computer Architecture, pp. 95-106, 2000.

[71] L. Villa, M. Zhang, and K. Asanovic, "Dynamic Zero Compression for Cache Energy Reduction”, in Proceedings of the 33rd International Symposium on Microarchitecture, December 2000.

[72] D.W. Wall, "Limits of Instruction-Level Parallelism", Techincal Report WRL 93/6, Digital Western Research Lab, 1993.

[73] S. Weiss, J.E. Smith, "Instruction Issue Logic in Pipelined Supercomputers", in the IEEE Transactions on Computers, Volume c-33, Issue 11, pp 1013-1022, November 1984.

[74] A. Wolfe and A. Channin, “Executing Compressed Programs on an Embedded RISC Architecture”, in Proceedings of the 19th International Symposium on Microarchitecture, 1992.

[75] W. Yamamoto, M. Nemirovsky, "Increasing Superscalar Performance through Multistreaming", in Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, pp. 49-58, 1995

[76] J. Yang and R. Gupta, “Energy Efficient Frequent Value Data Cache Design”, in Proceedings of the 35rd International Symposium on Microarchitecture, pp. 197-207, November 2002.

[77] J. Yang and R. Gupta. “FV encoding for Low PowerData I/O”. In Proceedings of the 2001 International Symposium on Low Power Electronics and Design, pp. 84-87, August 2001.

[78] V. Zyuban, “Inherently Lower-Power High-Performance Superscalar Architectures”, PhD. Thesis, Dept. of Computer Science and Engineering, University of Notre Dame, (Indiana), January 2000.

[79] L. Villa, M. Zhang, and K. Asanovic, “Dynamic Zero Compression for Cache Energy Reduction”, in Proceedings of the 33rd International Symposium on Microarchitecture, December 2000.

List of tables

List of figures