Speculative Software Management of Datapath-width for Energy Optimization

Gilles Pokam
gpokam@irisa.fr

Olivier Rochecouste
orocheco@irisa.fr

André Seznec
seznec@irisa.fr

François Bodin
bodin@irisa.fr

IRISA
Campus Universitaire de Beaulieu
35042 Rennes Cedex
France

ABSTRACT

This paper evaluates managing the processor's datapath-width at the compiler level by means of exploiting dynamic narrow-width operands. We capitalize on the large occurrence of these operands in multimedia programs to build static narrow-width regions that may be directly exposed to the compiler. We propose to augment the ISA with instructions directly exposing the datapath and the register widths to the compiler. Simple exception management allows this exposition to be only speculative. In this way, we permit the software to speculatively accommodate the execution of a program on a narrower datapath-width in order to save energy. For this purpose, we introduce a novel register file organization, the *byte-slice* register file, which allows the width of the register file to be dynamically reconfigured, providing both static and dynamic energy savings. We show that by combining the advantages of the *byte-slice* register file with the advantages provided by clock-gating the datapath on a per-region basis, up to 17% of the datapath dynamic energy can be saved, while a 22% reduction of the register file static energy is achieved.

Categories and Subject Descriptors

D.3.4 [**Processors**]: Compilers

General Terms

Algorithm, Experimentation, Management

Keywords

Speculative execution, energy management, compiler, narrow-width regions, clock-gating, reconfigurable computing

1. INTRODUCTION

In a modern processor, the major source of power consumption comes from the switching activities that occur when the hardware components are being exercised, as a result of executing a program. Hence, a common technique used by architects to save energy consists in clock-gating the unused portions of the chip. This has been done at the pipeline level [16] to reduce the energy impact due to executing wrong-path instructions. Recently, with the growing interest for multimedia applications, there has been a new avenue for exploiting fine-grain clock-gating at the operand level. This latter approach is motivated by the fact that most multimedia applications execute on 8- or 16-bit data; thus requiring only part of the full processor's datapath. Brooks et. al. [5] have recognized this opportunity. They observed that with a 64-bit Alpha-like processor, more than 50% of the instructions had their operands with 16-bit or less while executing the MiBench programs suite [12]. A significant fraction of the processor's power-efficiency is thus wasted when operating with these narrow-width operands.

Brooks has proposed to exploit this narrow-width data by means of a hardware-based technique. The implementation targets a general purpose processor. Unfortunately, at the compiler level, very few works have tried to exploit the occurrence of these narrow-width operands as a means to save energy. There has been some attempts to determine statically the operand's width of program statements written in a high-level language [15, 23], sometimes with the assistance of programmer's visible hints. One may think for instance of using this information to save energy, as proposed in [8]. These approaches are however very conservative because they rely on static data flow analysis and make no assumption on runtime data. This latter data comprises the largest amount of narrow-width operands, as demonstrated in [5]. Most of the other research devoted to this topic concentrates on efficiently exploiting SIMD instructions in software [19, 13]. However, the aim of SIMD compilation is not to "gate-off" unused portions of the processor's datapath, but instead to utilize its full capacity by packing narrow-width operands into large registers.

In this paper, we evaluate an integrated hardware/software approach for managing the energy consumption at the compiler level. The ISA is augmented with an instruction indicating that the subsequent code might be executed through a narrower path; thus requiring only narrow datapath and narrow register operands. This instruction is just a hint. At hardware-level, a simple exception management allows to recover instructions executing with full datapath-width on an incorrect hint. The rationale for speculating the processor's datapath-width at the compiler level is justified by the following facts: first, as previously stated, we are already witnessing the limitations of compiler analysis to uncover narrow-width operands, as software-only-controlled datapath-width must be conservative; second, the large occurrence of the dynamic narrow-width operands may suggest that a profiling approach would probably enhance the compiler capability of exploiting this narrow-width data more intensively. In this sense, we introduce datapath-width speculation principally as a means to predict, at the software level, the operand bitwidth of certain program regions; this in order to anticipate the reconfiguration of the processor resources.

The contributions of this paper are two-folded. First, we provide evidence that there exist static code regions corresponding to dynamic program instances, where the vast majority of the operands execute with a narrow-width. We then present a profile-based technique to uncover these regions at code generation time. Second, we present the architectural support that exploits these regions at runtime. The idea is to reconfigure the datapath width as well as the register file width when such a narrow-width operand region is encountered. Central to our approach is the *speculative nature* of the execution width of a region: we present a simple and efficient recovery mechanism for handling such width mispredictions.

The remainder of this paper is organized as follows. In Section 2, we further discuss the motivations to our work, precisely emphasizing the main differences with other related approaches. We show evidence of narrow-width operands regions in Section 3. The architectural support that permits to exploit these narrow-width operands regions is detailed in Section 4. Then, in Section 5, we present a strategy to detect them at code generation time. The experimental results are presented in Section 6, while Section 7 concludes this work.

2. MOTIVATION AND RELATED WORK

Exploiting the processor's datapath with narrow-width operands is not a new topic of research. The SIMD programming paradigm has been introduced primarily as a means to take advantage of the full processor's datapath-width for improving the multimedia performance [13, 19]. As a side-effect of applying SIMD techniques, some studies have shown that energy consumption can also be reduced [9]. This can be primarily attributed to the reduction in the number of executed instructions. However, many issues make the exploitation of SIMD techniques very difficult to realize. In particular, complicated vectorizing techniques are often needed to uncover the parallel operations to be coalesced in a single SIMD instruction. In addition, if the underlying ISA imposes some constraints on memory data alignment, it might even be more challenging to use SIMD instructions. Therefore, the effective parallelism covered by SIMD techniques can be severely restricted due to these constraints.

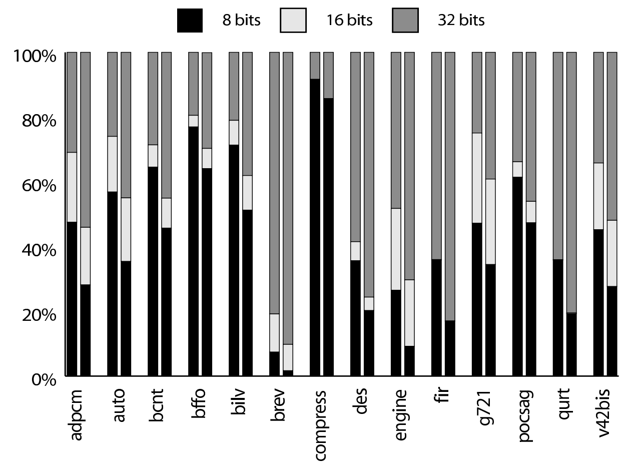


Figure 1: Cumulated distribution of operands bitwidth. The first bar shows results for one operand; the second bar shows results when both operands are considered.

Brooks et. al. [5] emphasized the availability of narrow-width operands in programs. They conducted their experiments with a 64-bit Alpha-like processor. We have performed equivalent experiments on typical embedded applications, e.g. Powerstone benchmarks [20], running on a 32-bit RISC-like embedded processor [10]. The results shown in Figure 1 illustrate that, on average, 45,5% of the instructions have their operands with less than 16-bit or equal. This is already in accordance with Brooks's estimations which found that about 50% of the integer instructions execute with narrow-width operands in multimedia applications running on a general purpose system.

Unlike a general purpose system, however, in an embedded system, the compiler plays a central role in achieving high performance. It is therefore of importance to improve the compiler's effectiveness to manage both power and performance. Since the basic block is the natural compiler granularity, we introduced the possibility to master narrow-width operands at the basic block level. At this granularity, the compiler can even achieve better energy/performance tradeoff, rather than relying solely on the hardware, since much more hardware components can be "turned off" over a longer period of time. Moreover, in contrast to the dynamic approach proposed in [5], considering bitwidth regions at the compiler level provides the additional advantage of reducing the overhead due to clock-gating on a cycle-by-cycle basis.

Canal et. al. [6, 7] proposed two approaches to tackle narrow-width operands. In [6], they considered a byte-serial (8-bit) or a semi-parallel (16-bit) pipeline to exploit narrow-width data at the architecture level. The idea relies on appending extension bits to data residing in caches and registers in order to reflect which part of the processing data is significant. Only the useful bytes are loaded, stored or computed on, and therefore a significant fraction of the

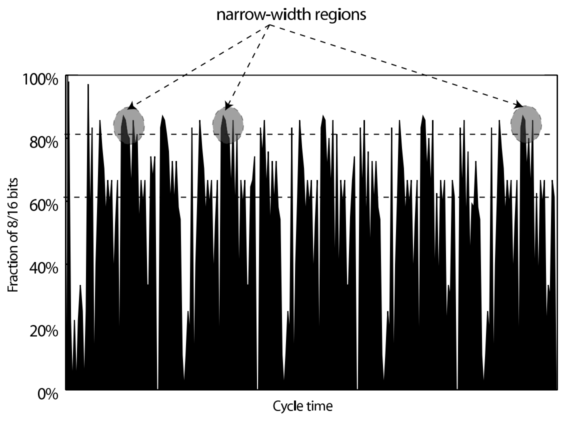


Figure 2: Dynamic bitwidth distribution at the granularity of a basic block for *adpcm*.

switching activities can be reduced. However, the fixed nature of the processor's datapath incurs a high performance penalty when processing operands of a larger bitwidth, e.g. 32-bit or more. This performance degradation can simply not be afforded on performance-critical embedded systems.

In [7], the authors proposed to use a software-based technique to direct the operand-gating decision. The idea is to rely on profiling information used conjointly with static compiler analysis techniques such as value range propagation to discover useful ranges of operand-width. Energy savings is achieved by re-encoding operands with narrower opcodes. This approach is orthogonal to our, since it can also be used to uncover regions with even more narrow-width operands.

Loh [14] proposed an hardware speculation scheme for dynamically predicting narrow-width operands on a per-instruction basis. The solution features a complex hardware mechanism that best fits a superscalar processor. Our approach is superior to the solution proposed by Loh since we address the speculation of narrow-width operands at a coarser granularity, i.e. at the region or basic block level.

Nakra et. al. [18] also proposed to exploit narrow-width operands in the context of embedded systems. They relied on profiling information to speculate narrow-width operands that may be packed together in the same VLIW instruction. The main goal is to achieve a better exploitation of the processor resources.

3. BITWIDTH DISTRIBUTION ANALYSIS

Through profiling, we collected statistics on the width of operands for applications from the Powerstone benchmarks suite [20] on various input data sets. For instance, Figure 1 illustrates that narrow-width operands can be of a large number on *adpcm*. In this section, we consider their availability at the basic block level and we propose to examine their distribution across a program run.

Figure 2 captures a snapshot of the dynamic operands bitwidth profile of the *adpcm* benchmark. Each point of the x-axis identifies a dynamic instance of a given basic block, while the value associated with the y-axis represents the occurrence of the narrow-width operands within that basic block. It can be seen from the figure that a sufficiently large number of basic blocks execute with more than 60% of their operands having 16-bits or less. This last point may

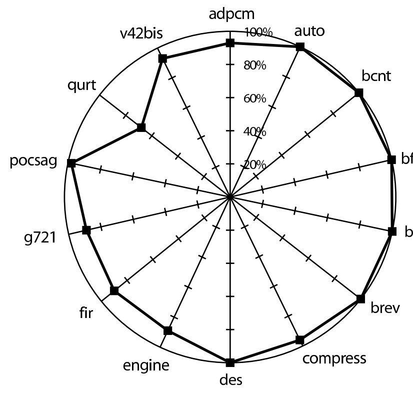


Figure 3: Average operand bitwidth convergence.

lead to two main observations. First, this indicates that a strong narrow-width operand locality exists for the considered granularity. Second, this suggests that executing these basic blocks on a narrower datapath-width may increase the compiler opportunities for savings the energy. Hence, we may try to take advantage of this to speculatively accommodate the width of the processor's datapath to the operand's width of a basic block or region.

However, before we may exploit this fact, we must ensure that basic blocks exhibiting such a behavior verify a property we call *bitwidth convergence*. Bitwidth convergence refers to the fact that, for a given basic block, its operands width may not vary frequently enough during execution. The rationale behind this property is to prevent the compiler from optimizing on very sensitive narrow-width operands regions. We estimated the bitwidth convergence in the following manner. When a basic block is found to execute with 16-bit or less (according to a defined threshold), we record for each future execution of the same basic block the number of times we are wrong. We then average this value on all the basic blocks of concern. This provides us with an estimate of the average bitwidth convergence for a given application. Typically, a high value indicates that bitwidth transitions occur very infrequently from one dynamic instance of a region to another. The results of the bitwidth convergence, considering an 80% narrow-width operands availability, are shown in Figure 3. On most applications in our benchmarks set, basic blocks execute with constant operand's bitwidth on our data set inputs.

4. ARCHITECTURAL SUPPORT

In this section, we examine a potential architectural support for exploiting narrow-width operands regions. One beneficial approach may consist in reducing the pipeline's activity while achieving acceptable performance. Therefore, we present a reconfigurable architecture that may dynamically adapt itself to an application's bitwidth behavior.

4.1 Hardware-exposed reconfiguration instruction

In order to benefit from narrow-width data elements at the software level, we propose to enhance an ISA with an hardware-exposed datapath-width reconfiguration instruction. The effect of this instruction can be deemed only as a hint to predict the execution width of subsequent regions.

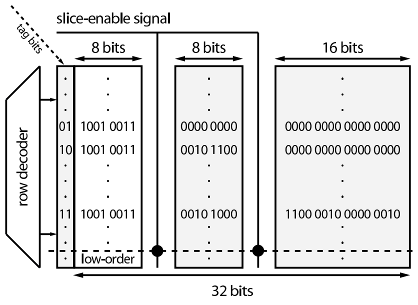


Figure 4: Byte-slice register file.

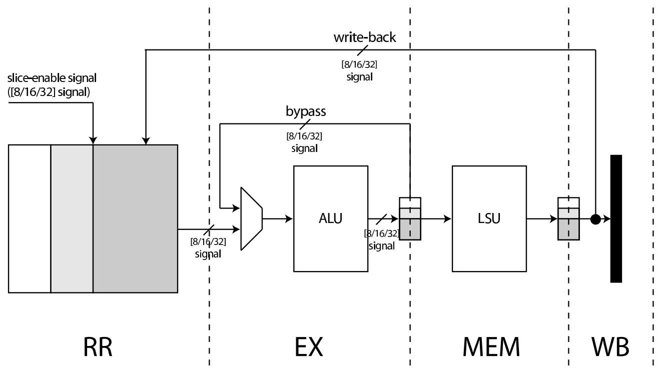


Figure 5: datapath.

Via the use of this instruction, the compiler may speculatively cause the execution of a region to accommodate on a narrower datapath-width (8-bit or 16-bit). Then at runtime, a simple hardware-based exception mechanism will allow to recover instructions executing with full datapath-width in case of a misprediction.

4.2 Register file model

Related Work. Previous research on reducing the register file activity focused on either, limiting the number of registers [3] or, limiting the number of ports [24]. Only few studies attempted to capitalize on narrow-width data for the same purpose. Canal et al. [6] proposed to load, store or compute only significant bytes in the whole pipeline stages. To do so, they designed a byte-serial pipeline where the data is processed on 8-bit slices. In order to provide this 8-bit access, they considered a 32-bit register file partitioned into 8-bit banks. In their study, as only one bank is requested per cycle, this multi-banked approach permits to reduce the register file activity. In contrast to their work, we are considering a data-path that is dynamically resizable according to the application's needs. As a matter of fact, in a multi-banked model, the row decoders are replicated on each bank. Therefore, accessing a wide data would generate redundant decoding and thus, useless power consumption.

Our Approach. We introduce a novel register file organization, the *byte-slice* register file. This energy-aware design permits to dynamically resize the register file width so that it can be viewed as a 8-, 16- or 32-bits conventional register file, as depicted in Figure 4. The register file is logically splitted into three slices: the first slice, representing the low-order data byte, is always enabled, whereas the others are controlled by means of a "slice-enable" signal. In our scheme, at anytime, the registers can hold different bitwidth data; and thus, it is not possible to turn off unused slices, unless there is a way to recover the lost information. Considering this fact, the slices are turned off in a *low-power* mode, achieved by the drowsy state [11]. In order to support such a state, we assume that the memory cells are modified as described in [1]. Technically, the drowsy circuitry is a state-preserving circuit that relies on voltage scaling for leakage reduction. A slice in a low-power mode preserves its data, although, it must switch back to the normal mode to get the correct information. The tag bits illustrated in Figure 4 provide this feature. We will get back to this later when we will discuss the recovery mechanism.

In contrast to [6], the adaptability and the simplicity of the *byte-slice* concept provide the advantages of being well suited to dynamically reconfigurable pipelines. In addition, the drowsy circuitry, which represents only a small area overhead [1], makes our design inherently low-power.

4.3 Reconfigurable datapath

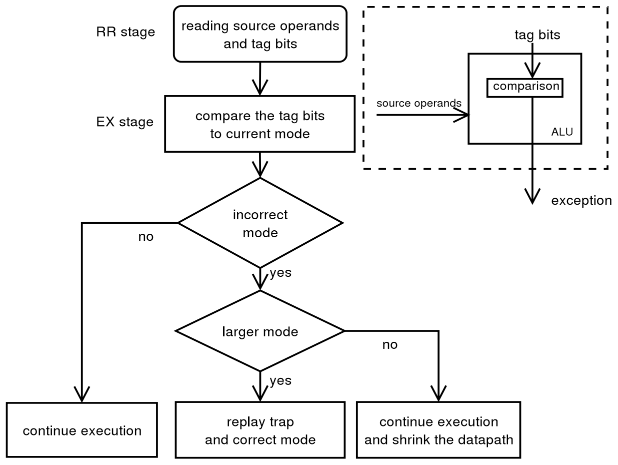
In this section, we describe a power-effective pipeline that may take advantage of the narrow-width regions. As depicted in Figure 5, the datapath has the ability to adapt to the bitwidth behavior of an application. This reconfigurable aspect is done via the clock-gating technique [5]. Clock-gating is a well-known scheme used to reduce the dynamic power consumption in today's processors [16]. In our approach, the coarser clock-gating granularity (at region level) reduces the amount of dynamic power dissipated by the clock-gating circuitry [2].

4.4 Recovery mechanism

To tackle the disadvantages of a static compiler analysis, as pointed in [5], we propose to statically construct narrow-width regions by using runtime information. In order to increase the number of these regions, we also consider the ones that verify the *bitwidth convergence* property; thus introducing datapath-width speculation. However, since it is not realistic to profile each application for each input data, or due to a dynamic event, a datapath-width misprediction may occur. In this section, we present a recovery mechanism that identifies the malformed regions and acts accordingly.

The main idea is to use a few tag bits to decide whether the current narrow-width region has been correctly predicted. In this respect, we use two tag bits appended to each register (see Figure 4) in order to discriminate between the different datapath modes (i.e. 8, 16, 32-bit mode). With a 32-bit width register file, this represents a negligible area overhead, with only 6% of the area being devoted to the tag bits. These tag bits reflect the true data-width and are generated by the functional unit, upon completion of an operation, and by the memory unit, upon a load instruction. [6] uses a similar scheme, however, we employ the tag bits in a different manner. While in [6] they act as a way to serialize the execution, in our proposal the tag bits dictate the use of the recovery mechanism.

The flow chart shown in Figure 6 illustrates the basic concept of this recovery mechanism. When an instruction reads its source operands from the register file, both the data and the tag bits are fed to the functional unit. A



RR stage
reading source operands and tag bits
EX stage
compare the tag bits to current mode
tag bits
comparison
source operands
ALU
exception
incorrect mode
no
yes
larger mode
no
yes
continue execution
replay trap and correct mode
continue execution and shrink the datapath

Figure 6: Recovery mechanism.

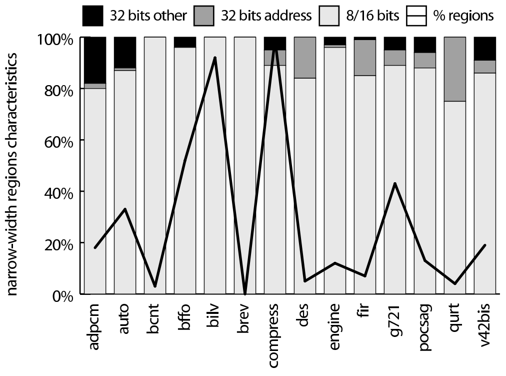


Figure 7: Average narrow-width regions characteristics (bargraph) and regions representativeness in program (linepoint).

ISA extension	Description
MOVACC Reg	ACC ⇒ Reg
MOVREG ACC	Reg ⇒ ACC
LDACC Reg	(ACC) ⇒ Reg
STACC Reg	Reg ⇒ (ACC)
ADDACC Reg	Reg + ACC ⇒ ACC
SUBACC Reg	ACC - Reg ⇒ ACC

Table 1: Basic address instructions.

simple comparison logic, located at the execute stage, detects whether the current operating mode is correct or not. If it appears that the current mode is narrower than the one expected, the current instructions are replayed, i.e. the pipeline is flushed and the correct width is enabled. When an instruction produces a result larger than the current mode, the pipeline is stalled while switching to the correct width. Although this mechanism may relatively impact on performance, its hardware simplicity fits well into the embedded context.

4.5 Handling address instructions

Address instructions, e.g. *load* and *store*, must be handled separately, since they usually require a larger bitwidth to represent memory addresses. We may address this problem by using a dedicated register file for memory addresses, in a way which is reminiscent to a decoupled architecture approach [22]. This feature is already integrated on some modern embedded processors [17]; they may therefore directly benefit from our scheme. For the processors that do not provide support for this feature, we suggest using special purpose registers, e.g. accumulator registers, for hosting and computing memory addresses. Along with the accumulator registers, the ISA must also permit the data transfers between the register file and the accumulators. Table 1 shows a possible subset of basic instructions that must be provided to support this scheme. In the table, the load and store instructions must have their base address residing in an accumulator register. The arithmetic instructions might be needed for computing new addresses.

5. NARROW-WIDTH REGIONS

Having analyzed the distribution of the narrow-width data and the relative bitwidth convergence of the regions, this section discusses the formation of the narrow-width regions.

5.1 Selecting candidates regions

In selecting the candidates regions, we may be forced to leverage the availability of the narrow-width operands against the probability that a bitwidth misprediction occurs at runtime. This might be primarily due to the fact that only a few regions would be able to exhibit narrow-width operands exclusively. This phenomenon can be indeed observed in Figure 2, where no perfect candidates regions can be found. Therefore, these regions may be chosen according to an arbitrary narrow-width operands availability, first ignoring the constraints due to the wide data. We assume for the rest of this study a threshold at 80%, which corresponds, on average, to one instruction out of five that executes with at least one 32-bit operand. Under this consideration, Figure 7 illustrates the average bitwidth profile of a narrow-width operands region as well as their weight in the program. The figure reveals that some applications have perfect narrow-width operands regions (e.g. *bcnt* and *bilv*), although some of them may not count too much in the total program weight, e.g. *bcnt*. Some others, however, include instructions with larger operand's bitwidth. These are labeled in the figure with *32-bit other* and *32-bit address*. The former indicates the fraction of instructions, not counting the memory instructions, having one of their operands with 32-bit. The latter represents the fraction of memory instructions. We may then attempt to build 32-bit-free operands regions out of these regions.

5.2 Regions transformation

It is explicit from the previous section that building a narrow-width operands region implies to deal with the 32-bit operands instructions. This section discusses a technique to efficiently overcome this problem.

5.2.1 Graph partitioning

By assuming that we have a means to deal with address instructions separately, e.g. accumulator registers, the problem to which we are confronted at this stage may be viewed as a graph partitioning problem. Let the graph G denotes the data dependence graph confined to a basic block. A node N of G represents a basic block operation. Two nodes, N and M, of G are connected via an edge e if there exists a *def-use* relationship among them. The graph partitioning problem consists in selecting the set of load/store nodes having one of their operand with 32-bit, in order to replace them with equivalent accumulator-based instructions, while minimizing the cut-size. The cut-size may be viewed as the number of additional instructions needed to move the data between the accumulators and the register file. This latter must be kept small enough in order not to impair the performance and the energy. We use a simple branch-and-bound heuristic to achieve this goal, deciding at each processing step whether or not the cut-size is within an acceptable range. Otherwise, the transformations are simply undone and the region is left unchanged.

5.2.2 Code restructuring

The problem that is pointed out in this section arises as soon as we have a narrow-width operands availability of less than 100% within a region. Let us assume that we are dealing with such a candidate region (a basic block). The problem to which we are confronted is to reorder the instructions in that region such that instructions having at least one operand with 32-bit (determined during profiling) are moved around it. The solution to this problem may be better illustrated in Figure 8.

A first operation consists in renaming all the destination operands of the instructions having one of their source operand with 32-bit, that may be used ahead of its computation. In this way, we augment the opportunities of finding more instructions that can be moved around. A second operation consists in computing the sets $MoveUp$ and $MoveDown$ corresponding to the 32-bit instructions that can be moved towards the beginning or the end of the basic block, respectively. Finally, a last operation consists in scheduling the instructions contained in each one of these sets upwards or downwards the underlying basic block, depending on the set to which they belong. Note that a side-effect of this algorithm may eventually cause some instructions to be duplicated if they are scheduled across a control flow graph join point. In addition, this might also lead to augment the pressure on the register file. This latter point can however be avoided if we consider a large register file, e.g. 64 general-purpose registers like that featured in our processor model.

6. EXPERIMENTAL RESULTS

This section discusses the evaluation results of the proposed scheme. We first present a brief overview of our solution. Then, a description of our methodology is exposed. Last, we evaluate and comment our results.

6.1 Solution overview

A synoptic view of our approach can be depicted in Figure 9. It consists of two main phases, a profiling phase and a narrow-width regions formation phase. In the first phase, the program is instrumented and stressed with different input data sets. At each time, statistics about the operand's width are gathered and stored for further utilization. The instrumentation is done by means of SALTO [4], which is a general, compiler-independent tool that makes the manipulation of the assembly code at the CFG level easier. In the second phase, the profiled data collected during the first phase is merged to create a converged profile for each application. From this profile, narrow-width regions candidates are initially identified by SALTO and then processed to create more refined regions. The reconfiguration of the processor datapath as well as the width of the register file is performed at runtime, every time the execution proceeds through a narrow-width region. For this latter to take place, we assume that the widths of the execution datapath and the register file are exposed to the compiler via explicit reconfiguration instructions (see Section 4.1).

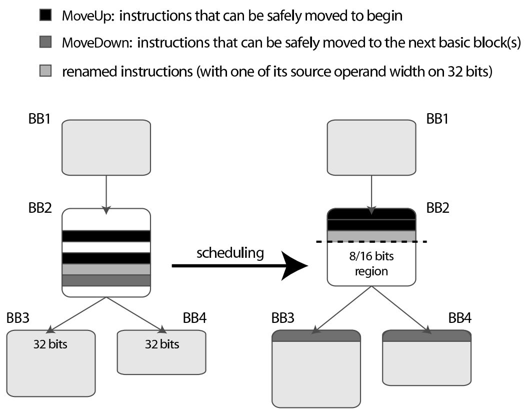

MoveUp: instructions that can be safely moved to begin
MoveDown: instructions that can be safely moved to the next basic block(s)
renamed instructions (with one of its source operand width on 32 bits)
BB1
BB2
scheduling
8/16 bits region
BB3
BB4
32 bits
32 bits

Figure 8: Bitwidth sensitive scheduling example.

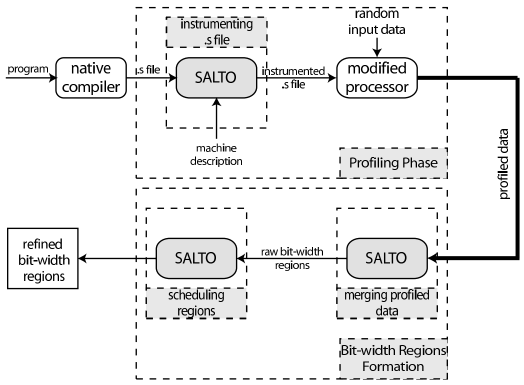


Figure 9: Optimization flow-graph.

6.2 Methodology

Platform. Our experiments were conducted on a RISC-like, 32-bit embedded processor belonging to the Lx family of customizable, multi-cluster VLIW architectures [10]. The processor's implementation used in this study features a six-stages pipeline, 4-issue width processor composed of 4 ALUs, 2 Multipliers, and 1 Load/Store unit with in-order execution, on each cluster. The different pipeline stages model the instruction fetch (IF), the instruction decode (ID), the register read (RR), the first stage execution (EX1), the second stage execution (EX2), and the write-back (WB). There are 3 forwarding paths which are EX1-EX1, EX2-EX1 and EX2-RR. Each cluster provides a set of 64 32-bit general pur-

Benchmark	Description
adpcm	voice encoding/decoding
auto	automotive control code
bcnt	bit count
bffo	find first zero
bilv	shift, and, or operations
brev	bit reverse operations
compress	data compression
des	data encryption
engine	engine control application
fir	integer FIR filter
g721	protocol for voice transmission
pocsag	communication protocol for paging
qurt	root computation of a quadratic equation
v42bis	modem encoding/decoding

Table 2: Benchmarks.

pose registers organized in a monolithic conventional register file. A set of 8 1-bit registers are used as branch condition registers.

Simulation. The Lx platform is provided with an industrial tool-chain, where no visible changes are exposed to the programmer. The tool-chain comprises, among other things, an aggressive ILP compiler, called the Lx compiler, from which we generate an input assembly. The extracted assembly code is processed by SALTO [4] as described in Section 6.1, to instrument the code and construct the narrow-width regions. The instrumented code is used to gather runtime statistics about register file access frequency, instruction's types, and operands bitwidth.

Benchmarks. We evaluated our scheme with applications collected from the Powerstone [20] suite of benchmarks. All the chosen applications were compiled with the Lx native compiler, with the optimization level 3, and then run until completion. Table 2 provides an overview of each benchmark used.

Energy model. In order to have a rough estimate of the energy savings that one may expect to gain with our scheme, we must quantify the energy consumption that is due to the register file on one side, and to the various pipeline stages on the other side. Let us first consider the register file. We model the dynamic energy consumption of a register file, $E_{RF}^{(dyn)}$, as follows:

$$E_{RF}^{(dyn)} = N_{rw} * E_{access} \tag{1}$$

where E_{access} is the average energy consumption on a read/write access, and N_{rw} the number of read/write accesses to the register file. We used a modified version of CACTI [21] for estimating the values of E_{access}, for both a conventional register file, as well as for our *byte-slice* register file architecture.

We employ the expression shown in (2) to quantify the static energy consumption due to the register file.

$$E_{RF}^{(stat)} = N_{cyc} * N_{cell} * P_{Leak} * \frac{1}{f} \tag{2}$$

Parameter	Value
Clock	1 GHz
nb of read/write ports	8/4
E_{access} (monolithic RF)	0.36 nJ
E_{access} (8-bit *byte-slice*)	0.11 nJ
E_{access} (32-bit *byte-slice*)	0.40 nJ
normal leakage power/cell	9.47 pW
drowsy leakage power/cell	2.34 pW

Table 4: Simulation parameters.

In the above expression, N_{cyc} is the number of cycles needed to execute the program, N_{cell} the number of cells contained in the register file, P_{Leak} the leakage power consumption per cell and f the processor's clock speed. The term P_{Leak} is strongly dependent on the technology and may vary with transistor size, width and temperature. Assuming current process technology parameter of 0.18 μm, we estimate P_{Leak} by means of Hotleakage [26], for both the normal and the drowsy modes.

Estimating the energy consumed by the other pipeline stages is a more difficult task, since very few processors vendors communicate detailed results about it. Nevertheless, we rely on power consumption estimates found in some research articles to derive realistic trends that govern the energy consumption of the involved processor's components. For our purpose, we are primarily interested on the energy consumption of the integer ALU and pipeline latches. In [25], the authors published energy results for a generic embedded processor, with a pipeline model very similar to ours. They noted that the obtained energy values were independent of the code being executed. One could deduce from this study that, on average, the register file and the pipeline latches account for ∼64% of the datapath power consumption, with the former representing 28% of the power and the latter 36%. The remaining 36% is due to the datapath multiplexers and the ALU, with the latter contributing for more than 27%.

Since on a narrower bitwidth mode, clock-gating prevents the high-order bytes of a pipeline to be latched, we can expect that the corresponding energy savings will be proportional to the bitwidth mode of the latch. Similarly, we save energy in the ALU structure by preventing its input latches from changing; thus restricting the computation to the low-order bytes of the input latches, yielding a linear reduction in the energy[1]. We summarize all the simulation parameters and the obtained ratios in Table 4 and Table 3.

6.3 Evaluation results

This section presents the evaluation results of the proposed narrow-width regions formation scheme. We center our discussion around four different aspects: the impact of the recovery mechanism, the code size growth, the dynamic energy reduction, and the leakage energy reduction.

Recovery mechanism. Let us consider a per-region narrow-width prediction rate of r for a total of *nbb* executed basic blocks. Then, assuming a misprediction frequency of

[1]We assume that the carry signal can be prevented from propagating along the higher-bit carries of the ALU. In such case, the energy savings can even be more important than what we have presumed.

components	datapath energy	savings 16-bit	savings 8-bit
latches	36%	18%	9%
ALU	27%	13%	7%

Table 3: Maximal energy savings.

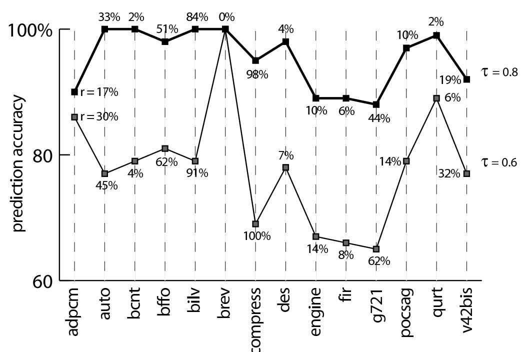


Figure 10: Prediction accuracy.

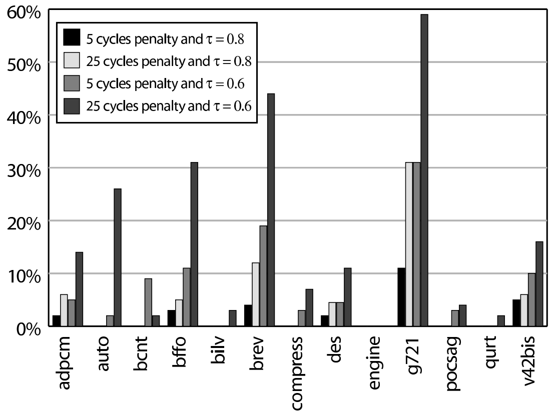


Figure 11: IPC degradation for different values of τ and p.

m, and an associated miss penalty of p, we may express the diminishing returns, $Cost$, of the recovery mechanism as follows:

$$Cost = nbb * r * m * p \qquad (3)$$

In (3), the misprediction penalty p may be viewed as the cost of flushing the pipeline plus the additive cost to recover the correct bitwidth size. Since the misprediction takes place at the execute stage, we may assume a 3 cycles penalty for flushing the pipeline. On the other hand, the cost to recover the correct bitwidth mode may vary with the implementation complexity and the processor design. We assume a 5 cycles recovery penalty for the best case and 25 cycles for the worst case. In Figure 10, we illustrated the impact of varying the narrow-width operand availability τ on the performance. As we increase τ, the speculation rate decreases because few regions may have high narrow-width operand availability. As a consequence, the misprediction frequency is also expected to decrease because the accuracy is sharpened. In contrast, lowering τ increases both r and m. Considering this fact, we plotted in Figure 11 the IPC degradation observed by varying the values of the narrow-width operands availability τ and the misprediction penalty p. On average, most applications experience IPC degradations without consequences on the performance when considering a best case misprediction penalty $p = 5$. In contrast, a worst case misprediction penalty of $p = 25$ can affect the performance by up to 31% for $\tau = 0.8$ and 60% for $\tau = 0.6$. An efficient scheme may therefore strive to keep p as low as possible.

Code size growth. Code size growth is mainly due to the re-encoding of the 32-bit address instructions with equivalent accumulator-based instructions and to the scheduling of the 32-bit instructions around the underlying basic block. Practically, a good cross-block scheduling algorithm may benefit from this code motion to improve the IPC and thereby alleviating the impact of the code size growth. We have not implemented such a tricky cross-block scheduling algorithm. Still, the impact of the code size growth is marginal. Considering a per-region narrow-width operands availability of 80%, we experienced less than 3,1% code size growth, on average, for our benchmarks set.

Dynamic energy reduction. The upper part of Figure 12 shows the breakdown of the dynamic energy savings obtained for each component of the processor's datapath. Some applications such as *bcnt*, *brev* and *qurt* show no benefit from using our scheme. This is mainly because not enough static narrow-width regions have been uncovered at code generation time. Our future works in this direction therefore seek at improving our approach by addressing the detection of these narrow-width regions at lower levels, e.g. at post-link-time where, hopefully, more optimization opportunities may be given. The lower part of Figure 12 shows the overall datapath energy savings realized with our scheme. We can indeed observe that an average energy savings of 17% can be obtained with the remaining applications. On some modern embedded processors such as the *M.Core* [20], for instance, the datapath dynamic energy contributes to as much as 42% of the total processor's power consumption. Achieving a 17% energy reduction can therefore provide a substantial energy gain.

Leakage energy reduction. The *byte-slice* register file architecture we proposed also permits to tackle the static energy consumption. This is mainly due to the fact that when executing on a narrower datapath-width, the upper byte-slices of the register file are put into in low-power mode. Figure 13 illustrates the static energy savings observed in the register file when using our scheme. For the vast ma-

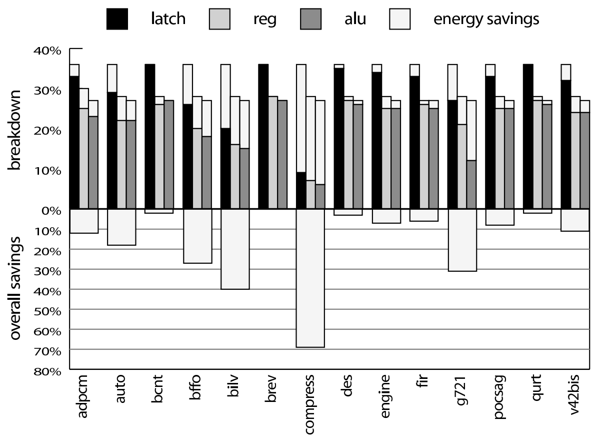


Figure 12: Breakdown of the datapath dynamic energy savings and overall gain.

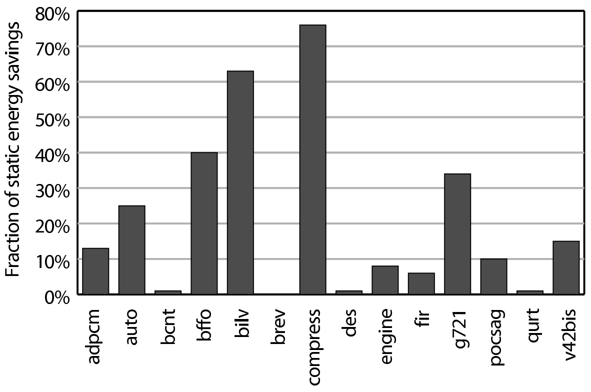


Figure 13: Register file static energy savings.

jority of the applications, an average of 22% reduction of the static energy is realized, with a peak energy savings of roughly 80% for the *compress* benchmark.

7. CONCLUSIONS

Operand-gating has been recently proposed as a means to dynamically exploiting the availability of narrow-width data elements in programs. Implementations of operand-gating have principally relied on the hardware to drive the gating decision. From a software point of view, some solutions have also emerged that take benefit of narrow-width operands to save energy. However, software-only solutions suffer a lot from not considering runtime information. Hence, they must often be very conservative about the ranges of bitwidth values that a data may take during its execution.

In this paper, we have proposed a speculative software management scheme to overcome the difficulties encountered by software-only solutions. Central to our approach is the ability to expose dynamic narrow-width operands to the compiler; this in order to allow it to speculatively accommodate the execution of static narrow-width regions on a narrower datapath-width. For this purpose, we have introduced a novel register file organization, the *byte-slice* register file, that permits the width of the register file to be dynamically reconfigured; and a simple and efficient exception management mechanism to handle width mispredictions. Our evaluation results have indeed demonstrated the efficacy of our approach in managing the energy consumption at the software level. We showed that up to 17% of the datapath dynamic energy and 22% of the register file static energy can be saved, while only a negligible IPC degradation is observed for most applications.

8. REFERENCES

[1] Ayala, J.L., López, V.M., Veidenbaum, A., and López C.A. Energy Aware Register File Implementation through Instruction Predecode. In *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors*, June 2003.

[2] Bahar, R.I., and Manne, S. Power and Energy Reduction Via Pipeline Balancing. In *Proceedings of the 28th International Symposium on Computer Architecture*, June 2001.

[3] Balasubramonian, R., Dwarkadas, S., Albonesi, D. Reducing the Complexity of the Register File in Dynamic Superscalar Processor. In *Proceedings of the 34th International Symposium on Microarchitecture*, December 2001.

[4] Bodin, F., Rohou, E., and Seznec, A. SALTO: System for Assembly-Language Transformation and Optimization. In *Proceedings of the Sixth Workshop on Compilers for Parallel Computers*, December 1996.

[5] Brooks, D., and Martonosi, M. Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, January 1999.

[6] Canal, R., Gonzales, A., and Smith, J.E. Very Low Power Pipelines Using Significance Compression. In *Proceedings of the 33th International Symposium on Microarchitecture*, December 2000.

[7] Canal, R., Gonzales, A., and Smith, J.E. Software-Controlled Operand-Gating. In *Proceedings of the International Symposium on Code Generation and Optimization*, March 2004.

[8] Cao, Y., and Yasuura, H. Low-Energy Design using Datapath Width Optimization for Embedded Processor-based Systems. *IPSJ Journal*, 43(5):1348–1356, May 2002.

[9] Drach, N., and Sebot, J. SIMD ISA Extensions: Tradeoff between Power Consumption and Performance on a Superscalar Processor. In *Proceedings of the Kool Chips Workshop*, December 2000.

[10] Faraboschi, P., Brown, G., Fisher, J.A., Desoli, G., and Homewood, F. Lx: A Technology Platform for Customizable VLIW Embedded Processing. In *Proceedings of the 27th International. Symposium on Computer Architecture*, June 2000.

[11] Flautner, K., Sung Kim, N., Martin, S., Blaauw, D., and Mudge, T. Drowsy Caches: Simple Techniques for Reducing Leakage Power. In *Proceedings of the 29th*

International Symposium on Computer Architecture, May 2002.

[12] Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R.B. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the 4th IEEE International Workshop on Workload Characterization*, pages 3–14, December 2001.

[13] Larsen, S., and Amarasinghe, S. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2000.

[14] Loh, G. Exploiting Data-Width Locality to Increase Superscalar Execution Bandwidth. In *Proceedings of the 35th International Symposium on Microarchitecture*, November 2002.

[15] Mahlke, S., Ravindran, R., Schlansker, M., Schreiber, R., and Sherwood, T. Bitwidth Cognizant Architecture Synthesis of Custom Hardware Accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(11), November 2001.

[16] Manne, S., Klauser, A., and Grunwald, D. Pipeline Gating: Speculation Control for Energy Reduction. In *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

[17] Moreno, J.H., et al. An Innovative Low-Power High-Performance Programmable Signal Processor for Digital Communications. *IBM Journal of Research and Development*, 47(2-3):299–326, March/May 2003.

[18] Nakra, T., Childers, B.R., and Soffa, M.L. Width-Sensitive Scheduling for Resource-Constrained VLIW Processors. In *Proceedings of the 3th ACM Workshop on Feedback-Directed and Dynamic Optimization*, December 2000.

[19] Pokam, G., Bihan, S., Simonnet, J., and Bodin, F. SWARP: A Retargetable Preprocessor for Multimedia Instructions. *Concurrency and Computation: Practice and Experience*, 16(2-3):303–318, February/March 2004.

[20] Scott, J., Hwang Lee, L., Arends, J., and Moyer, W. Designing the Low-Power M.CORE Architecture. In *Proceedings of Power Driven Microarchitecture*, June 1998.

[21] Shivakumar, P., and Jouppi, N. CACTI 3.0: An Integrated Cache Timing Power, and Area Model. Technical report, DEC Western research Lab, 2002.

[22] Smith, I.E., et al. The ZS-I Central Processor. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–204, October 1987.

[23] Stephenson, M., Babb, J., and Amarasinghe, S. Bitwidth Analysis with Application to Silicon Compilation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2000.

[24] Tseng, J.H., and Asanovic, K. Banked Multiported Register Files for High-Frequency Superscalar Microprocessors. In *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003.

[25] Vijaykrishnan, N., Kandemir, M., Irwin, M.J., Kim, H.S., and Ye, W. Energy-driven Integrated Hardware-Software Optimizations using SimplePower. In *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

[26] Zhang, Y., Parikh, D., Sankaranarayanan, K., Skadron, K., and Stan, M. Hotleakage: A Temperature-aware Model of Subthreshold and Gate Leakage for Architects. Technical Report CS-2003-05, University of Virginia, Department of Computer Science, March 2003.