Empowering a Helper Cluster through Data-Width Aware Instruction Selection Policies

Osman S. Unsal[1], Oguz Ergin[2], Xavier Vera[1], Antonio González[1]

[1]Intel Barcelona Research Center
Intel Labs, Universitat Politècnica de Catalunya
Barcelona, Spain
{osmanx.unsal,xavier.vera,antonio.gonzalez}@intel.com

[2]Department of Computer Engineering
TOBB Univ. of Economics and Technology
Ankara, Turkey
oergin@etu.edu.tr

Abstract

Narrow values that can be represented by less number of bits than the full machine width occur very frequently in programs. On the other hand, clustering mechanisms enable cost- and performance-effective scaling of processor back-end features. Those attributes can be combined synergistically to design special clusters operating on narrow values (a.k.a. Helper Cluster), potentially providing performance benefits.

We complement a 32-bit monolithic processor with a low-complexity 8-bit Helper Cluster. Then, in our main focus, we propose various ideas to select suitable instructions to execute in the data-width based clusters. We add data-width information as another instruction steering decision metric and introduce new data-width based selection algorithms which also consider dependency, inter-cluster communication and load imbalance. Utilizing those techniques, the performance of a wide range of workloads are substantially increased; Helper Cluster achieves an average speedup of 11% for a wide range of 412 apps. When focusing on integer applications, the speedup can be as high as 22% on average.

1. Introduction

As semiconductor technology scales down to deep sub micron range, wire delays are becoming more prominent compared to gate delays. Clustered microarchitectures [8][9][18][19] provide an effective solution to dealing with the problem of wire delays by partitioning processor resources, usually into a frontend (encompassing fetch, decode and rename) and multiple backends (schedule, execute and commit). However, although clustered architectures enable the scaling of instruction issue and execution resources, this comes at an area/complexity cost with some increased overhead compared to a monolithic architecture, such as inter cluster copy instructions (or bypass operations depending on implementation). Those instructions or operations result from data dependencies that exist when a destination and its consumer are mapped to separate clusters.

On the other hand, narrow values (i.e. values that can be represented by less number of bits than the full machine width) occur very frequently in typical programs and provide various microarchitectural optimization opportunities [4][7][14][17]. Here, to reiterate and illustrate the importance of narrow values, we coin a new definition: a consumer is said to be *narrow data-width dependent* if the producer value is narrow. Narrow data-width dependency is a dominant subset of data dependency. As an example consider Figure 1: we plot, for Spec Int 2000 applications, the percentage of register operands that are narrow (defined here as 8 bits) data-width dependent. As can be observed from the figure, there is substantial narrow data-width dependency. Alternatively, we have found that 39.4% of regular ALU instructions require one narrow-width operand, 3.3% require 2 narrow operands producing a wide result and 43.5% require 2 narrow operands producing a narrow result. Traditional clustering approaches based on symmetry do not exploit this potential.

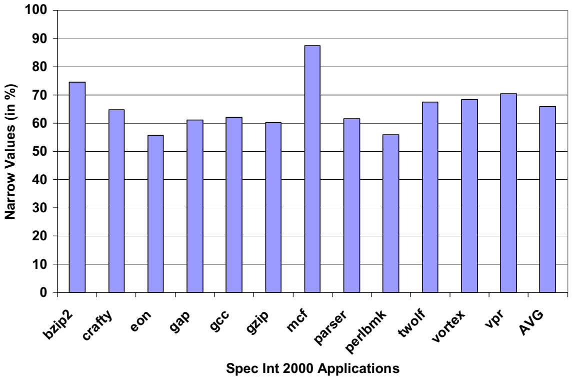

Figure 1- Data-width dependent values for register operands

1-4244-0054-6/06/$20.00 ©2006 IEEE

The best aspects of monolithic and clustered architectures can be potentially combined in an asymmetric clustering scheme based on data-width. However, to fully-realize this potential and achieve substantial speedups, sophisticated data-width dependency sensitive instruction steering schemes must be developed.

Similar to previous work [10], we propose to augment a monolithic processor with a small narrow data-width cluster core, which we term the Helper Cluster. The Helper Cluster is clocked faster than a full 32-bit cluster, potentially achieving a dramatic performance boost in principle. In practice however, the challenge is to develop innovative techniques to steer suitable instructions into the helper cluster to realize this goal. In this work, we demonstrate how to address this challenge to achieve the performance potential of Helper Cluster. Helper Cluster achieves an average speedup of 11% for a wide range of 412 applications. When focusing on integer applications, the speedup can be as high as 22% on average.

The crucial issue in a narrow datapath-width cluster is to balance the workload of the backends, by devising smart instruction steering mechanisms to keep the helper cluster occupied. Two important factors will enable us to fully exploit this performance potential. First, in general, the more the instructions that are executed in the faster narrow backend, the more the performance gain. Second, occasionally a consumer instruction assigned to a backend might need to use a value that is produced in the other cluster. For that purpose, we use the copy instruction scheme as proposed by Gonzalez et al.[6]. In this scheme a special copy instruction is generated by the consumer instruction. This copy instruction is steered to the backend of the producer to gather the value when it is produced and to copy it to the consumers' register file. For a detailed discussion of the implementation details, please refer to [6]. Copy instructions incur an obvious performance overhead, so the number and time penalty of copies should be minimized. Thus, we want to keep in check the number and penalty of copies generated between the clusters. Next, we summarize our steering techniques:

(1) If all the source operands and (if any) the result produced by the instruction is 8-bits (narrow), we steer those instructions to the helper cluster. In order to determine the width of instructions, we use a predictor to predict the width of operands and result.

(2) To increase the number of instructions steered into the helper cluster, we consider instructions with one narrow and one wide source and a wide result, with the upper bits of the wide source and the result being identical. This means that this instructions' execution does not change the upper order bits and that it is effectively a narrow operation that can be done in the helper cluster. We later discuss how such values can be predicted and the associated predictor design.

(3) We propose to steer conditional branches that depend on the flag produced by an instruction that has already been executed in the helper cluster, to the helper cluster. This minimizes the number of copies generated.

(4) We propose copy prefetching which minimizes the time penalty of a copy instruction. The basic idea is to predict when a producer instruction might generate a copy instruction later on, and generate the copy earlier at the producer, instead of at the consumer. When it needs, the consumer instruction will most likely have the value ready and will not block waiting for the value to arrive.

(5) We determine the amount of workload imbalance. If the helper cluster is overloaded, we steer narrow instructions to the wide cluster until the workload balance is restored again. Conversely, if the helper cluster is underloaded, we propose to split up wide, 32-bit instructions into 8-bit chunks and steer them to the helper cluster.

The rest of this paper is organized as follows: In section 2 we introduce the helper cluster microarchitecture. In section 3, we discuss each of the above contributions in detail including implementation issues; this is followed by relevant results in each subsection. In section 4, we survey relevant work and in section 5 present our conclusions.

2. Helper Cluster Microarchitecture

2.1. Helper Cluster Organization

We have conservatively chosen the helper cluster to be 8 bits wide, previous research shows that many 8-bit narrow values exist in typical programs [4][7], and therefore this is a good design point from a performance/complexity tradeoff point of view; we explain this further in the next section. Note that more narrow instructions would be executed in the narrow cluster and therefore the potential performance gains would be even higher if it would be possible to construct a wider than 8-bits. Figure 2 shows a monolithic processor extended with a helper cluster. This clustered microarchitecture executes Intel® IA-32 instructions. The frontend reads the instructions from the upper-level cache, the UL1; translates them into uops (henceforth referred as instructions) and stores them in a Trace Cache, from where they are read, decoded, and steered to any of the backends according to the steering policy. Each backend has its own integer register file, integer instruction queue and ALUs. In addition, the wide backend has floating point instruction queues and FPU's. Floating point functional units usually incur a high area overhead, and operate on wider data. Therefore, for a complexity effective design, the helper cluster has integer functional units only. The area overhead of the helper cluster is very

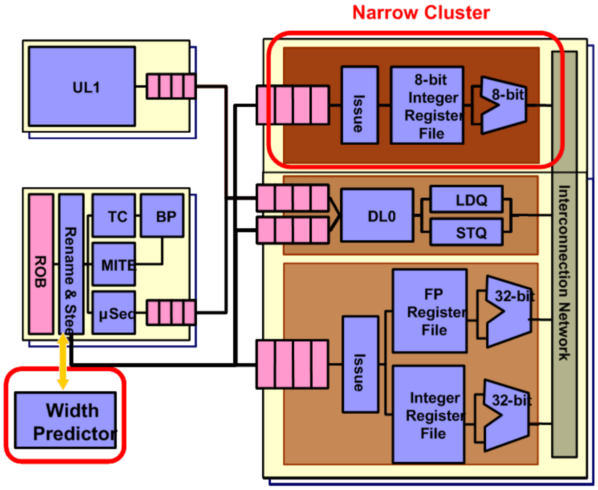

Figure 2 - Processor block diagram with the wide (32-bit), and the helper (8-bit) backends.

small as compared to a full 32-bitwidth cluster since the areas of typical backend structures such as the register files [3] or ALU's [16], scale at least linearly with the data-width. Although there can be more than one helper and/or wide clusters, for ease of presentation we will be henceforth considering two backends; one wide (32-bit) and the other narrow (8-bit).

Narrow values are detected through leading zero (or leading one) detectors. Figure 3 shows the circuit diagrams for 8-bit leading zero and one detectors respectively, which employ dynamic logic for faster operation and larger fan in.

2.2 Helper Cluster Clocking

The Integer Functional Units (ALU's, AGU's) in general, and the adder and the associated bypass loop in particular, are usually the critical path in a microprocessor [22] and determine the limit to which the frequency can be pushed in the backend of the cluster. For example, in Pentium-4 [11], although the issue queue is clocked at the fast 2X "fireball" frequency, the adder can only be clocked at this frequency by adopting a 16-bit staggered organization; therefore a 32-bit add takes one "slow" cycle. The delay of this critical path scales with the datapath width since the adder delay depends on the size of the input operands [15]. Moreover, the adder circuit area also depends on the operation width and that in turn is an important factor in determining the propagation delay of the bypass loop. Typical high-performance ALU latencies are on the order of logN (N being the operand width). Therefore, through considering the ALU and bypass latencies, the 8-bit helper backend can be clocked 2 times faster than the 32-bit backend. This also ensures that the clocks remain synchronized with respect to each other thus avoiding the inter-cluster clock resynchronization costs [21]. In case the critical path is on other structures, we performed experiments with reduced issue queue size and issue width; the results indicate negligible impact on performance.

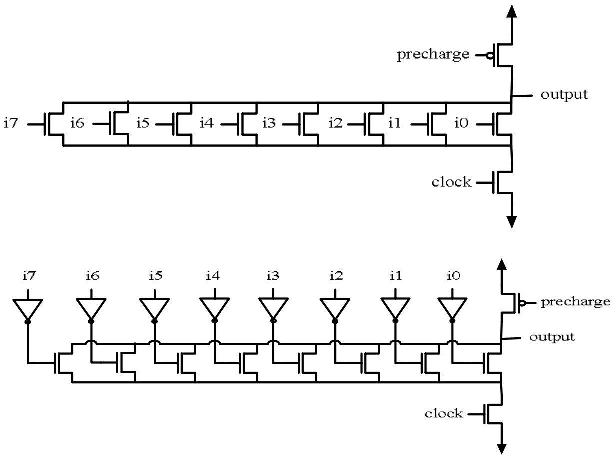

Figure 3 - Consecutive Zero (a) and One (b) Detection Circuits.

3. Evaluation of Steering Techniques

3.1 Simulation Methodology

Results provided in this section were collected from an IA-32 trace-driven simulator modeling an Intel Pentium® 4 - like processor. Due to simulation time for detailed analysis, we use 12 traces generated from SPEC Integer 2000 benchmarks; for final analysis we discuss the results over a wider range of workloads. Each trace is composed of 100 million instructions. To skip the initialization section, we split each benchmark into 10 equal slices and start executing from the fourth slice. The performance results are given with respect to a baseline monolithic processor which has the same resources as the frontend and the wide backend of the cluster. As discussed in the previous section, the Helper Cluster augments the baseline with the narrow cluster. Table 1 lists the parameters used in the experiments. For calculating power/energy, we utilize an in-house wattch-like [2] power simulator, modified to take into account the helper cluster power, including the 8-bit datapath and the clock network as well as the width predictors.

Trace Cache(TC)	32Kuops,4w
Level-1 DCache (DL0)	32KB,8w,3cycle,2R/Wport
Level-2 Cache (UL1)	4MB,16w,13cycle,1R/Wport
Integer Execution	32 entry scheduler, 3 issue
Fp Execution	32 entry scheduler, 3 issue
Commit Width	6 instructions
Main Memory	450 cycles

Table 1 - The monolithic baseline processor parameters

3.2 All source operands and output narrow (8-8-8)

If we predict that all the operands and the output are narrow (8 bits), then we steer that instruction into the helper cluster. The predictor uses a simple table-based tagless scheme. The table is indexed by the PC; the predictor occupies 1 bit per entry, the width predictor logic predicts the width of a result of an instruction by storing a single bit to remember the last generated width. We have experimented with various table sizes, a size of 256 entries was found to be a good compromise between complexity and performance, therefore this size was selected for the final design. Please see Figure 4 for an embodiment of the width predictor showing the modifications on the datapath structures.

Regarding the source widths, width information is stored inside a field in the rename table called width table (which is 1-bit wide) and is updated with correct outcome later while updating the width predictor to improve the prediction accuracy for the source operands. When a new instruction is decoded, the width prediction for the destination register is read out. For the source operand width, the actual width is read if the producer instruction has already written back the result; if not, the prediction is read. If the instruction has any immediate fields we obtain the actual width. If all source operands and output of an instruction need values of 8 bits or less, the instruction is sent to the helper cluster.

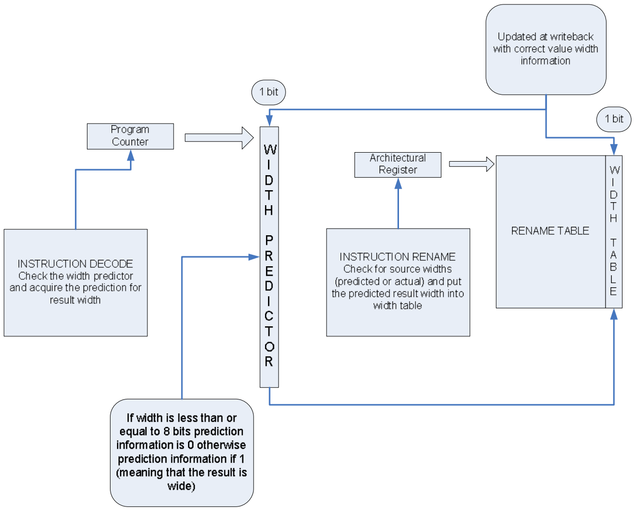
Updated at writeback with correct value width information
1 bit
1 bit
Program Counter
W I D T H P R E D I C T O R
Architectural Register
RENAME TABLE
W I D T H T A B L E
INSTRUCTION DECODE Check the width predictor and acquire the prediction for result width
INSTRUCTION RENAME Check for source widths (predicted or actual) and put the predicted result width into width table
If width is less than or equal to 8 bits prediction information is 0 otherwise prediction information if 1 (meaning that the result is wide)

Figure 4- Data width predictor

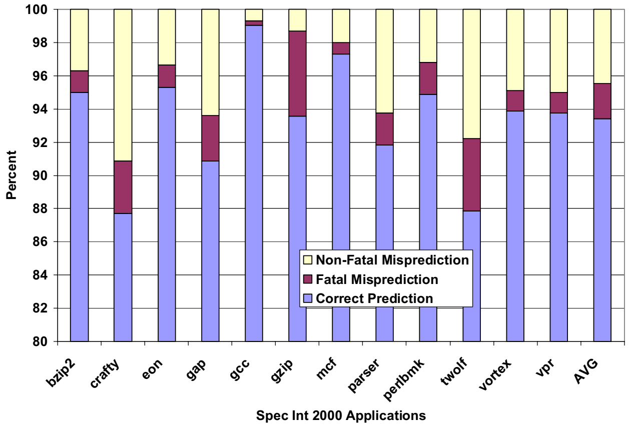

Figure 5 – Width prediction accuracy

The width prediction accuracy is around 93.5% on average, decreasing the percentage of the cases that need recovery because of a misprediction, see Figure 5. Note that recovery is only necessary in the case of a misprediction for an instruction that has been steered to the narrow backend (we term this a fatal misprediction); a misprediction for an instruction that has been steered to the wide backend (which could be otherwise executed in the narrow backend) is a missed opportunity and does not require recovery. In case of a misprediction requiring recovery, the impacted instructions have to be squashed in the narrow backend and resteered into the wide backend. We adopt a flushing scheme which squashes all instructions starting from the mispredicted one. Although simple, this scheme has a high performance overhead in the case of a misprediction. Therefore, to decrease the misprediction rate, we augment the predictor with a 2-bit per-entry confidence interval estimator. We only take the decision to steer the predicted narrow instruction to the helper cluster if the prediction is with high-confidence. This fine-tuning decreased the misprediction requiring recovery to 0.83% from 2.11%. As a result, 15% of the instructions are steered to the helper cluster, with a 6.2% performance boost compared to the baseline; see Figures 6 and 7. Examining the figures, note that this scheme generates a relatively large number of copy instructions since the generated narrow value is likely to be used in the wide cluster for addressing or indexing purposes. Also note that the application with the worst performance (bzip2) has a very high copy/narrow instruction ratio, while the application with the best performance gain (gcc) has a low copy/narrow instruction ratio. The results here may seem to contrast with the results in Figure 1, where we established that on average 65% of the consumers are

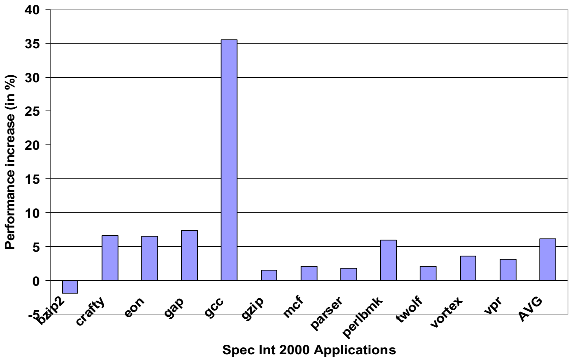

Figure 6 Performance of 8_8_8 scheme

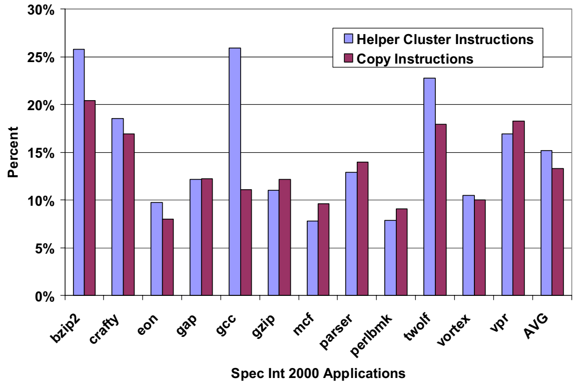

Figure 7 Percentage of instructions steered to the helper cluster, and of inter-cluster copies

narrow-width dependent on the producer. However, we need to consider that *all* the input operands (which can be more than 2 in the IA-32 internal machine state) and the result value of a particular instruction must to be narrow to use this steering mechanism. It is clear that this particular combination occurs less frequently. The observations above motivate us to develop and explore a mechanism to steer more instructions into the helper cluster while trying to minimize copies. We introduce such a mechanism based on dependencies in the next section.

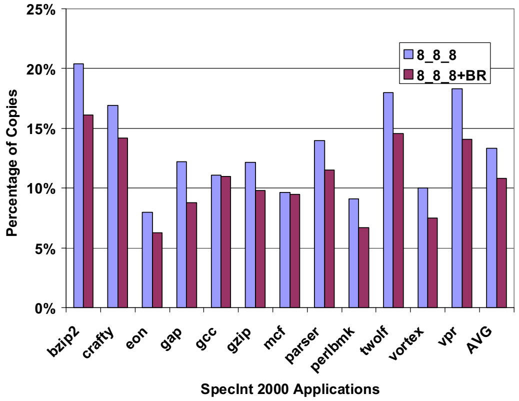

Figure 8 – Decrease in copy percentage due to use of BR scheme.

3.3 Branches Dependent on Narrow-Value Condition (BR)

We can leverage another data-width sensitive characteristic to steer more instructions to the helper cluster and decrease generated copies. The idea is based on the fact that many conditional branches, such as those at loop boundaries, depend on a narrow value. In other words, the conditional branch (that checks the flags register) usually depends on an arithmetic instruction (which produces the write to the flags register) that is narrow. Now we discuss how to steer those conditional branches to the helper cluster. We propose to move some of the conditional branch address resolution to the frontend; this enables more branches that depend on narrow values to be steered to the narrow backend. Many conditional branches calculate their target address through the addition of an immediate operand value with the value kept in the register of the code segment and the instruction pointer (EIP register). This addition can be done in the frontend since the value of the code segment register does not change and the EIP register contains the offset within the code segment. Those conditional branches can be easily tagged since they have a unique operand. In consort with this framework, the steering scheme works as follows: if the instruction that last wrote to the flags register has been steered to the narrow backend, we can also steer the dependent conditional branch to the narrow backend. If this branch were steered to the wide backend instead, a copy would be generated to fetch the flags register from the narrow backend; we avoid this overhead, increase helper cluster instructions and decrease the number of copies. Adding this new technique, we steer 19.5% of the instructions to the narrow cluster, with 10.8% copy percentage, which yields a performance boost of 9%. As seen in Figure 8, BR effectively leverages dependency and data-width information to simultaneously steer more instructions into the helper cluster while decreasing the copy percentage.

3.4 Load Replication (LR)

Since there is a single Memory Order Buffer (MOB), registers could be allocated in both clusters for loads; thus minimizing copies. This is beneficial in cases where the load has to be done in one backend (e.g., in 32-bit cluster because the address resolution might require a 32bit add), and the result used in the other (e.g., in 8-bit cluster because the loaded value is 8-bit wide). Load replication

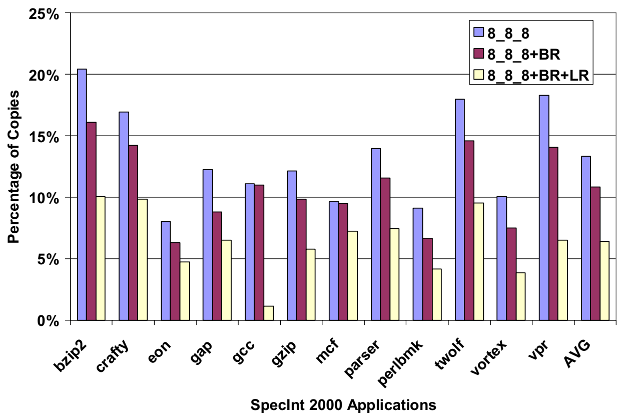

Figure 9 - Minimization in copy percentage due to use of LR scheme.

can also help when too many copies are generated between the 32-bit and helper clusters. Copies are avoided by duplicating loads for such cases as an 8-bit value being used in the 32-bit cluster later on.

Multiple load replication schemes can be considered. One such scheme would replicate the register value on the helper cluster for loads that are steered to the 32-bit cluster if they are predicted to load an 8-bit value. Another, more complexity-effective scheme would duplicate 8-bit loads on the wide cluster; here we adopt this scheme. Load replication decreases copies to 6.4% from 10.8%, see Figure 9 .

3.5 Carry Width Prediction (CR)

To further increase the percentage of instructions that are steered into the helper cluster, we exploit the instructions whose operation only works with the lower order 8-bits of the inputs and there is no carry propagation beyond the lower order 8 bits. We term this scheme CR. As an example consider load instructions whose address is calculated by adding a small offset to the large base address with only the lower order bits of the large base address changed. Figure 10 illustrates this case for a load address calculation on an Address Generation Unit (AGU). Here, contents of the base register R2 is added to the offset register R3 to get the address. In this case the address calculation can be done in the helper cluster. We have done an analysis of such instructions (with two sources, one 8-bit and the other 32-bit wide, and with one 32-wide result) to determine the percentage in which the carry is not propagated. The results, shown in Figure 11 for loads and certain arithmetic instructions such as add or subtract indicate that CR has substantial potential.

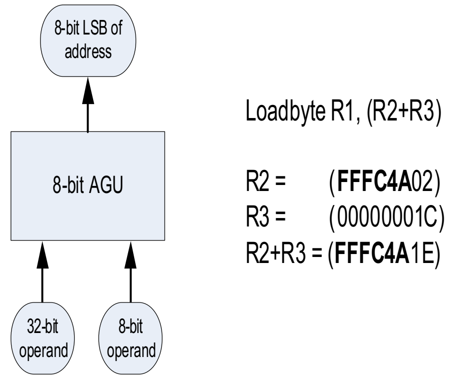

Figure 10 - Example for Carry Not Propagated.

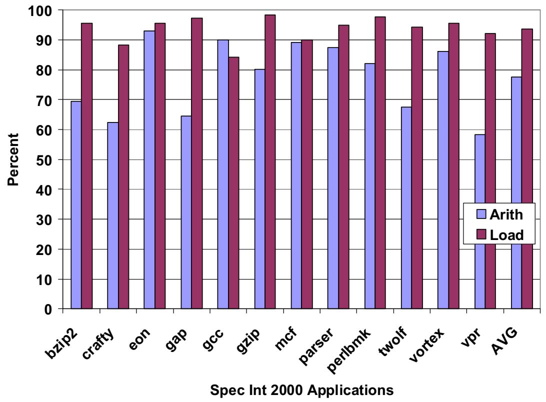

Figure 11 – For instructions with two sources, (8-bit and 32-bit), and with one result (32-bit); the percentage that the carry is not propagated

For such cases, we extend the data value width predictor to predict when carry does not propagate beyond the 8 bits. An instruction is eligible to be considered as "carry will not be propagated", if its result is predicted to be 32 bits and only one of its sources is either predicted to be or is actually 32-bits wide. This is obtained from the width predictor. Along with this information, an additional bit in the width predictor has to be used to indicate if the last occurrence of this instruction had operated with only 8 bits. This bit is set at writeback if the previous preconditions are satisfied. The new instruction is steered to the narrow backend if the predictor indicates that previous occurrence did not propagate a carry beyond 8 bits. Similar to the 8-8-8 case, a 2-bit confidence estimator is used to decrease the fatal misprediction rate. Since we utilize the carry signal to catch fatal mispredictions, such arithmetic operations as divide and multiply are not eligible to be considered for this technique. To reconstruct the 32-bit value, the rename table entry associated with the newly allocated destination physical register is appended with an extra tag that points to the register holding the upper 24-bits of the produced value in the wide cluster. The register deallocation mechanism must be modified as

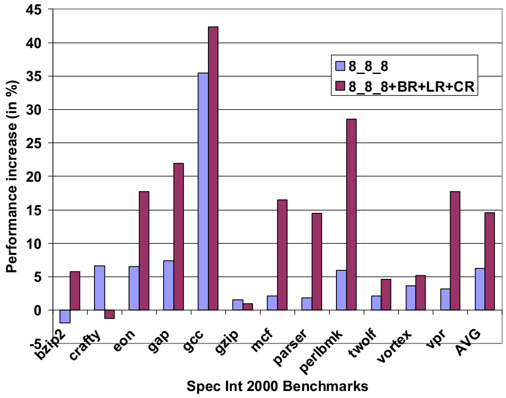

Figure 12 – Performance of Carry Not Propagated (CR) scheme

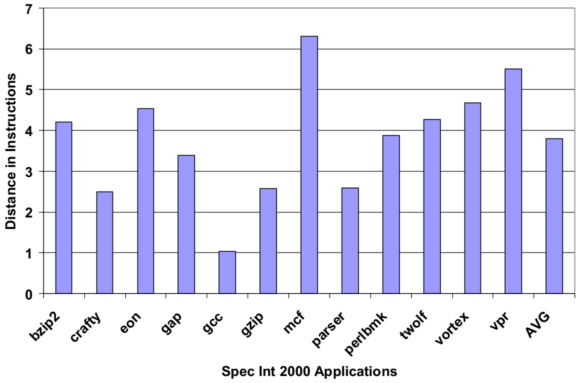

Figure 13 – The average producer-consumer distance for IA-32.

well: a counter is associated with the wide physical register. The counter is incremented when an 8-32-32 condition is detected. The decrement occurs when the renamer of the destination register of the 8-32-32 instruction deallocates the definition. The 32-bit register is then deallocated only when its renamer commits and the counter associated with it is zero. This check does not incur any additional delay since it requires a simple zero check which can be done in parallel with the renamer commit check. Adding the CR mechanism, the results indicate that 47.5% percent of the instructions are executed in the helper cluster with a 15.7% copy percentage; the performance improvement compared to the baseline is 14.5%, see Figure 12.

3.6 Copy Prefetching (CP)

BR and LR schemes aimed to decrease the percentage of copies; however the copy instructions also have an associated time penalty, we address this penalty next. To decrease the copy time penalty, we propose Copy Prefetching (CP) for narrow backends. Prefetching the copy at the producer may lead to a performance gain, since the value needed by the consumer is prefetched to the consumers' backend, thus reducing the stalling time of the consumer. This gain depends on the distance between the producer and the consumer instructions. If this distance is very small, then the effectiveness of copy prefetching is diminished. If the distance is very large, the prefetched copy instructions will waste backend resources while waiting for the consumer. As seen in Figure 13, the IA-32 architecture has good producer-consumer distance characteristic for prefetching.

As discussed before, the implemented CP mechanism generates a copy at the producer if the CP predictor predicts that a copy might be generated later on. The predictor, which is last value based, is orthogonal to the previous predictors and can be constructed by adding a bit to the base width predictor. If a producer instruction incurs a copy later on, its CP prediction bit is set at writeback. This triggers a prefetch at the next iteration of this instruction. Although the CP predictor can be used to predict copies in both directions, this tends to increase the copy percentage. Therefore, we found that hybrid policies can further boost performance so we use the CP predictor to predict narrow-to-wide copies. To predict wide-to-narrow copies, we use the existing result width predictor for identifying narrow values produced in the wide backend (this could be the result of a load-byte instruction that is executed in the 32 bit backend); and then prefetching them to the narrow backend since those values will most likely be copied into the narrow backend. Our studies indicate that the CP predictor has an accuracy of 90% percent. While increasing the copy percentage to 21.4%, CP increases the performance gain to 16.7%.

3.7 Instruction Splitting for Imbalance Reduction (IR)

Although 47.5% of instructions are steered to the helper cluster, it might be the case that the helper cluster might be under/over utilized. Intuitively speaking, one is tempted to declare that the helper cluster is underutilized. This intuition is based on the fact that the performance-optimal ratio (in the absence of copies and dependences) is 66% of the instructions being executed in the 2X faster helper cluster. However, we have to test whether this intuition holds. One standard technique to measure this imbalance is the NREADY metric [18][19]. According to this metric, the workload imbalance at a given instant of time is defined as the total number of ready instructions that cannot issue, but could have issued in the other cluster. If the helper cluster is underutilized there is comparatively more wide-to-narrow imbalance; if the helper cluster is overutilized the narrow-to-wide imbalance dominates. The results indicate that with the current steering schemes, there is little narrow-to-wide imbalance (about 2%), but a significant wide-to-narrow imbalance (about 22%) exists.

The above conclusion establishes that the helper cluster is underutilized and more wide instructions could be steered to the narrow cluster. On the surface, it may seem that splitting up a wide instruction into four and executing them on the 2X faster narrow cluster may not be very advantageous. However, note that by splitting-up wide instructions when there is wide-to-narrow load imbalance and by steering the split up instructions to the temporarily underutilized helper cluster, we can achieve considerable speedups. To that end, we developed a complete design which splits up “wide instructions” into 4 multiple “narrow instructions” in the decode stage. Those narrow instructions are the same in every respect with the wide replica, except that they use 8-bit register sources and destinations so that they can execute in the helper cluster. Therefore, if the wide instruction has a destination register, the four split-up narrow instructions allocate four register entries at the rename stage. Another required modification is that each split narrow instruction is made dependent on each other in a chain fashion from the instruction that calculates the least-significant byte to the one that processes the most significant byte. This ensures that the split narrow instructions are executed back-to-back in the correct order. It is very likely that the result is used as a source operand in the wide cluster later on; therefore the full 32-bit register value is prefetched by dispatching four 8-bit copy instructions to the wide cluster.

Whenever wide-to-narrow imbalance exists (as indicated by the discrepancy of the issue queue occupancy rates of the clusters); we use the above scheme to split up instructions and steer them to the narrow cluster. We achieve a speedup of 22.1% with 72.4% of instructions steered to the narrow cluster while the wide-to-narrow imbalance decreases to 2.3% from 22%.

A fine tuning could be applied the above heuristic by splitting up instructions when imbalance exists *and* the instruction has no destination register. This heuristic achieves a balance between imbalance reduction and communication costs: the wide-to-narrow imbalance increases to 5.1% from 2.3%, however the copy instructions incurred drops to 24.4% from 36.9%. A speedup of 21.3% is achieved with 63.6% of instructions steered to the narrow cluster. A further future extension to the above idea is a helper cluster that operates with a looser granularity: complete blocks of wide instructions are split up and sent in their entirety to the narrow cluster, thus minimizing copies while decreasing imbalance.

Finally, we have done an energy-delay^2 comparison of the monolithic baseline with helper cluster in its most resource aggressive configuration (i.e., the configuration in this section), the results indicate that helper cluster is 5.1% more energy-delay efficient than the baseline.

3.8. Wrap-up

We conclude the section with a study comparing the baseline with the best performing steering, the IR mechanism. We selected a comprehensive category of workloads and we simulated 10 million consecutive IA-32 instructions for each benchmark. The details are given in Table 2.

Workloads	#traces	Description/Examples
Encoder (enc)	62	Audio/video encode
SpecFP2K (sfp)	41	Spec FP’s
Kernels (kernels)	52	VectorAdd, FIRs
Multimedia (mm)	85	WMedia, photoshop
Office (office)	75	Excel, word, ppt
Productivity (prod)	45	Internet content
Workstation (ws)	49	VectorAdd, FIRs

Table 2 – The various categories of workloads used in the study

The results in Figure 14 show that Helper Cluster consistently increases performance, with workloads with comparatively regular control flow (such as multimedia) and many arithmetic operations (kernels, sfp) benefiting more than office or productivity applications.

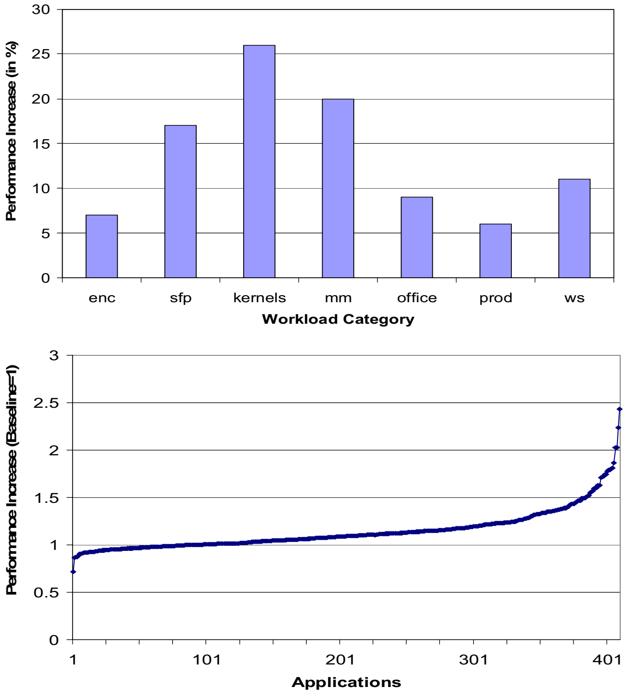

Figure 14 – Helper Cluster performance for various workloads

4. Related Work

Making use of narrow values has been proposed before in the context of monolithic, non-clustered systems. Canal, González and Smith observed that typical applications use many data values that are narrow and studied compression schemes to encode narrow values in the context of an in-order processor to save power [4]. Packing multiple narrow values into wide function units is discussed in [1]. That approach assumes that the value widths are known at instruction issue time. On the other hand, [13] argued that the operand widths cannot be known at issue time and proposed using a width predictor in order to identify instructions with narrow operands. A similar width prediction mechanism was proposed by Nakra et al., [17] for a VLIW-style machine. Some techniques propose optimizations for power efficiency [23], where the presence of zero bytes was exploited for reducing the cache energy consumption. In [14] and [20], narrow width operands were exploited to reduce the power requirements of a value predictor. A software-controlled operand gating is proposed in [5], where the ISA is extended to include the opcodes that specify operand widths. In [12], Lipasti et al. introduced a technique for reducing register file pressure that exploits significance compression [23]. In their technique, narrow width results are stored in the rename table entry itself. Packing multiple narrow values into wide registers was proposed in [7]. Those approaches examine narrow values in the context of non-clustered systems.

A very recent work [10] has proposed considering narrow values in the context of clustering. Instead of adding a narrow cluster to a monolithic processor, the authors start with a homogeneous cluster and shirk one of the 64-bit clusters to 20-bits. Targeting the Alpha architecture, more than 80% of the instructions are executed in this 20-bit narrow cluster. They utilize an inter-cluster bypass scheme to forward values across clusters and propose a replicated register file. A history-based prediction scheme is used to predict narrow instructions which are then steered to execute in the narrow cluster. In case a predicted narrow instruction turns out to require wide cluster resources, they propose a replay mechanism to recover from this misprediction. To deal with the data invariant-portion of load/store address calculations a special address register file is proposed which is shared across clusters. In comparison, we propose and evaluate a suite of steering mechanisms which consider inter-cluster load balancing and producer-value prefetching mechanisms on an Intel® IA-32 clustered architecture. Regarding complexity, both approaches present different challenges: On one hand, our microarchitecture does not require the register file to be replicated, although we study smart mechanisms for load value replication. We also avoid the synchronization and complexity issues associated with resources that are shared across clusters such as the address register file. This centralized register file can be challenging to implement, especially if low-latency operation is required. In the case of mispredictions we adopt a flushing mechanism; and use a confidence-interval based scheme; instead of using replay-based schemes which can be costly. On the other hand, we use a copy instruction scheme to communicate values across clusters. This scheme requires the addition of a special copy microinstruction, as well as requiring its own scheduling resources.

5. Concluding Remarks

We propose data-width aware instruction steering, splitting and copy prefetching mechanisms for achieving substantial performance gains through the addition of a low complexity 8-bit helper cluster operating on narrow data widths. The techniques exploit data-width dependencies (a subset of data dependency) for 22% average performance improvement for Spec Int 2000 applications. Beyond the five new data-width aware techniques proposed in this paper; to the best of our knowledge, this is the first work that proposes copy instruction prefetching or instruction split-up for clustered microarchitectures. Proposed extensions to this work include designing a simple core working with narrow data and operands on a CMP.

References

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

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

[3] Bunchua S., “Fully Distributed Register Files for Heterogeneous Clustered Microarchitectures”, *Ph.D. Thesis, Georgia Tech.,* July 2004.

[4] Canal R., González A., Smith J.E., “Very Low Power Pipelines Using Significance Compression”, in *Proceedings of 33th International Symposium on Microarchitecture (MICRO-33)*, Portland, USA, December 2000.

[5] Canal, R., González, A., Smith, J., “Software-Controlled Operand Gating”, in *Proc. of the Intl. Symp. On Code Generation and Optimization*, 2004.

[6] Canal R., Parcerisa J. M., González A., "A Cost-Effective Clustered Architecture", Proc. of the 1999 International

Conference on Parallel Architectures and Compilation Techniques (PACT-99), October 1999.

[7] Ergin, O., Balkan, D., Ghose, K. and Ponomarev, D., "Register Packing: Exploiting Narrow-Width Operands for Reducing Register File Pressure", in *Proceedings of 37th International Symposium on Microarchitecture (MICRO-37)*, December 2004.

[8] Farkas K.I., Chow P., Jouppi N.P., Vranesic Z., "The Multicluster Architecture: Reducing Cycle Time through Partitioning", in *Proceedings of the 30th Iinternational Symposium on Microarchitecture (MICRO-30)*, Dec. 1997.

[9] Gwennap L., "Digital 21264 Sets New Standard", *Microprocessor Report, 10(14),* Oct. 1996.

[10] González R., Cristal A., Pericas M., Valero M., Veidenbaum A., "An Asymmetric Clustered Processor Based on Value Content", in *Proceedings of the 19th ACM International Conference on Supercomputing (ICS-2005),* June 2005.

[11] Hinton, G., Upton, M., Sager, D., Boggs, D., Carmean, D., Roussel, P., Chappell, T. I., Fletcher, T. D., Milshtein, M. S., Sprague, M., Samaan, S. and Murray, R., "A 0.18-μm CMOS IA-32 Processor With a 4-GHz Integer Execution Unit", *IEEE Journal of Solid State Circuits*, Vol. 36, No. 11, November 2001, pp.1617-1627.

[12] Lipasti, M., et.al., "Physical Register Inlining", in *Proc. of the Int'l. Symp. On Computer Architecture* (ISCA), 2004.

[13] Loh, G., "Exploiting Data-Width Locality to Increase Superscalar Execution Bandwidth", in *Proc. of the International Symposium on Microarchitecture*, 2002.

[14] Loh, G., "Width Prediction for Reducing Value Predictor Size and Power", in First Value Pred. Wksp, ISCA 2003.

[15] Lu S.-L., "Speeding Up Processing with Approximation Circuits", in *IEEE Computer*, March 2004.

[16] Miyaoka Y., et al., "Area/Delay Estimation for Digital Signal Processor Cores", in *ASP-DAC '01: Proceedings of the 2001 Conference on Asia South Pacific Design Automation*, 2001.

[17] Nakra, T., et.al., "Width Sensitive Scheduling for Resource Constrained VLIW Processors", *Workshop on Feedback Directed and Dynamic Optimizations*, 2001.

[18] Parcerisa J. M., González A., "Reducing Wire Delay Penalty through Value Prediction", in *Proceedings of the 33th International Symposium on Microarchitecture (MICRO'00)*, December 2000.

[19] Parcerisa J. M., Sauquillo J., González A., Duato J., "Efficient Interconnects for Clustered Microarchitectures", in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'02)*, September 2002.

[20] Sato, T., Arita, I., "Table Size Reduction for Data Value Predictors by Exploiting Narrow Width Values", in *Proc. of the International Conference on Supercomputing*, 2000.

[21] Semeraro, G., Magklis, G., Balasubramonian, R., Albonesi, D. H., Dwarkadas, S., and Scott, M. L., "Energy-Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling", In *Proceedings of the Eighth international Symposium on High-Performance Computer Architecture (Hpca'02),* Feb. 2002.

[22] Sprangle E., Carmean D., "Increasing Processor Performance by Implementing Deeper Pipelines", in *Proceedings of the International Conference on Computer Architecture, June 2002.*

[23] Villa, L., Zhang, M. and Asanovic, K., "Dynamic Zero Compression for Cache Energy Reduction", in Micro-33, Dec. 2000.