A Co-designed HW/SW Approach to General Purpose Program Acceleration Using a Programmable Functional Unit

Abhishek Deb
Universitat Politecnica de Catalunya
C6-221, C. Jordi Girona, 1-3
Barcelona, Spain
abhishek@ac.upc.edu

Josep M. Codina and Antonio González
Intel Research Labs Barcelona
C. Jordi Girona, 1-3
Barcelona, Spain
{josep.m.codina, antonio}@intel.com

Abstract

In this paper, we propose a novel programmable functional unit (PFU) to accelerate general purpose application execution on a modern out-of-order x86 processor in a complexity-effective way. Code is transformed and instructions are generated that run on the PFU using a co-designed virtual machine (Cd-VM). Groups of frequently executed micro-operations (micro-ops) are identified and fused into a macro-op (MOP) by the Cd-VM. The MOPs are executed on PFU.

Results presented in this paper show that this HW/SW co-designed approach produces average speedups in performance of 17% in SPECFP and 10% in SPECINT, and up-to 33%, over modern out-of-order processor. Moreover, we also show that the proposed scheme not only outperforms dynamic vectorization using SIMD accelerators but also outperforms an 8-wide issue out-of-order processor.

1 Introduction

Modern microprocessor designs have shown significant performance improvements over the last decades by exploiting instruction level parallelism (ILP). However, exploiting this ILP came with the addition of complex hardware structures that lead to prohibitive power consumption and design complexity [13]. It is widely known that increasing the width of the processor provides significant speedups as demonstrated in [16]. On the other hand, due to recent advancement in process technology, transistors in a die are fairly abundant. In this scenario, we argue that specialized hardware and accelerators is a promising alternative to harness both the abundance of transistors and the potential of a wider machine. These performance improvements are achieved under a reasonable power budget and design complexity.

Single-instruction multiple-data (SIMD) accelerators are commonly used in current microprocessors to accelerate the execution of multimedia applications. These accelerators perform the same computation on multiple data items using a single instruction. Intel's SSE [1] and AMD's 3DNow![12] extensions are examples of such instructions for the x86 ISA. Although, SIMD accelerators provide significant performance gains in multimedia applications at low cost and energy overheads, they are not good enough for general purpose applications.

More recently, several multiple-instruction multiple data (MIMD) accelerators have been proposed that range from programmable to specialized functions [17, 18, 3, 8]. Due to design complexity and lack of compiler and code generation techniques, in order to leverage the accelerators efficiently, these accelerators have not yet been implemented.

Introducing such hardware accelerators needs to be supported by extending the ISA. Applications need to be recompiled to the new ISA in order to use these hardware accelerators. A co-designed virtual machine (Cd-VM), however, solves this problem by transparently recompiling the application binary to the host ISA. Transmeta Crusoe [11, 5] is a commercial example of a Cd-VM based processor.

In this paper, we propose a novel programmable functional unit (PFU) to accelerate general purpose application execution, in a complexity-effective way. We leverage the fact that reducing the execution latency and increasing the width of the processors leads to a performance improvement. We use a HW/SW co-designed approach to build a out-of-order x86 processor by transparently optimizing applications, and improve performance, without increasing the width of the processor.

In the proposed scheme, the PFU is programmed using a Cd-VM. This software layer dynamically profiles the application code, and identifies frequently executed regions. It then optimizes these regions by fusing a sequence of micro-ops into a macro-operation (MOP). This transformed code is stored in a concealed memory and is executed instead of the original code. We also propose dynamic compilation techniques required in order to achieve this.

Results presented in this paper show that the use of a PFU provides a significant average speedup of 17% in SPECFP and 10% in SPECINT, and speedup of up-to 33% for some benchmarks, over current out-of-order processor. Moreover, we also show that the proposed scheme not only outperforms SIMD accelerators when they are dynamically managed by the Cd-VM, but also outperforms an 8-wide issue out-of-order processor.

The key contributions of this paper are as follows:

- We propose a novel Programmable Functional Unit, along with a novel split-MOP execution model. We also discuss the microarchitectural changes required in order to incorporate the PFU in a complexity-effective manner.
- We describe an effective algorithm to dynamically fuse instructions using a Cd-VM. Our dynamic compilation scheme handles memory and loop-carried dependencies to aggressively reorder the code and generate MOPs, appropriately.

The rest of the paper is organized as follows. In Section 2, we discuss the implementation of the baseline HW/SW co-designed out-of-order x86 processor. In Section 3, the proposed PFU is described along with its execution model and microarchitecture. Dynamic compilation techniques are discussed in Section 4. A detailed evaluation and analysis of the PFU, its design points and comparison with alternate schemes is presented in Section 5. Finally, related work is reviewed in Section 6 and we conclude in Section 7.

2 Baseline Processor

In this paper, we consider a HW/SW co-designed processor. The co-designed virtual machine (Cd-VM), virtualizes the hardware underneath and offers an x86 ISA to the operating system and applications. The hardware implements an internal ISA and provides support for efficient execution of the software. In this section, we will detail the main characteristics of the baseline processor implemented in a HW/SW paradigm.

Our baseline processor is based upon a state-of-art modern out-of-order processor. A Cd-VM is added on top of it to execute superblocks [10]. Hardware Profiling (block and edge)[4] identifies frequently executed code. Once a basic block is hot, superblock formation begins by following the most frequently taken edges. The Cd-VM optimizes superblocks and stores them in code cache.

Jump TLB (JTLB)[15] holds mapping from source program counter (PC) to target PC for each superblock and is accessed in parallel to the I-Cache. Code residing in the code cache is in micro-op ISA. The proposed out-of-order processor is able to fetch both non-optimized x86 ISA and micro-ops from the I-cache, using dual-mode decoders [8].

Early exit branches of superblocks are converted to asserts to ensure atomic execution similar to [14]. Therefore, in addition to x86-level exceptions and interrupts, superblock execution rolls back whenever an assert fails. On a rollback the whole of the superblock is discarded and unoptimized code is fetched.

3 Proposed Microarchitecture

3.1 Split-Mop Model

A MOP like any other instruction requires inputs and outputs to execute. However, the input and output parts of MOP are split into several micro-ops using a split-MOP model. Our split-MOP model consists of following micro-ops : (1) a set of loads to provide inputs from memory (ld-set), (2) a set of register moves to provide inputs from register file (mv-set), (3) a computation macro-op (CMOP), and (4) a store set (st-set). Figure 1 shows an example of split-MOP. Note, that irf0, irf1, irf2 in Figure 1b indicates the IRF registers.

```
ld rcx = [rax, 8]
add rdx = rcx, rax
add rsp = rbx, 1
sub rbp = rsp, rdx
st [rbp, 4], rax
```

(a) Micro-ops before fusion

```
ld irf0, rcx = [rax,8] : ld-set
mov irf1, irf2 = rax, rbx : mv-set
cmop rdx, rsp, rbp
 st [rbp, 4], rax
```

irf* : Internal Register File Tag

(b) Macro-op after fusion

Figure 1: Split-MOP Model

3.1.1 Computation Macro-Op (CMOP)

CMOP consists of a unique identifier, and destination registers. Transient registers are not reflected in CMOP's destination register. CMOP does not contain any source operands, because, it reads input values from the IRF, but it writes directly to the physical register file.

The encoded data corresponding to each fused micro-op in a CMOP is known as a configuration. Configurations are appended to the superblock and stored in the code cache. These configurations are located using the unique identifier encoded in a CMOP. The unique identifier is used as an index into a configuration TLB (CTLB).

ld	Issue	Read	Eag	Eat	Emd	WB		
mov	Issue	Read	Mov					
cmop				Issue	Read	Execute	WB	
st					Issue	Read	Eag	Eat

Figure 2: Execution Pipeline

3.1.2 Execution Model

The execution pipeline of the split-MOP as described in Figure 1 is illustrated in Figure 2. The four execution pipelines correspond to load, mov, CMOP and store respectively of 1b. The pipeline stages Eag, Eat and Emd stands for address generation, address translation and memory disambiguation respectively.

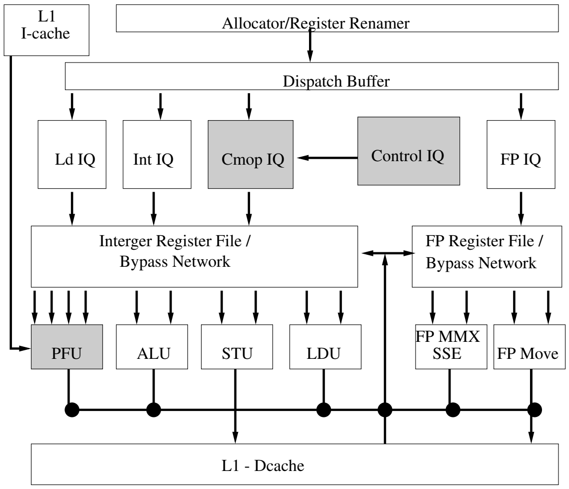


Figure 3: Modified Microarchitectural Block Diagram

3.2 PFU Microarchitecture

The microarchitecture that supports split-MOP execution is discussed in this section. Figure 3 shows a block-level microarchitecture diagram, and the added components are shown in gray. The added components are a PFU, 2 separate issue queues to facilitate PFU execution.

3.2.1 Programmable Functional Unit

We propose a PFU which has two major components: 1) Distributed Internal Register File (IRF), and (2) a grid of Processing Elements (PE). Data flows from one row to the following in the grid of PEs as shown in Figure 4, an organization similar to [3]. Note, that there are no latches between the PEs of two different rows. The effects of varying the grid size and PFU execution latency is studied and discussed in Section 5. The inputs required by each micro-op in the grid of PEs is provided by the IRF.

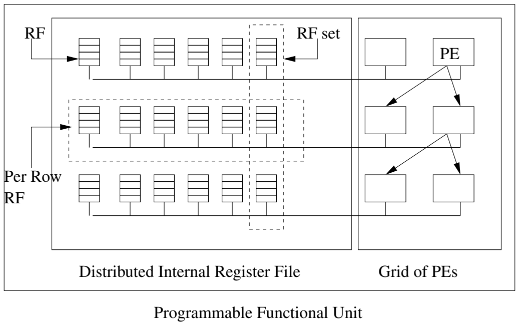

RF
RF set
PE
Per Row
RF
Distributed Internal Register File
Grid of PEs
Programmable Functional Unit

Figure 4: Programmable Functional Unit overview

3.2.2 Distributed Internal Register File

The proposed PFU with six (2 columns, 3 rows) PEs requires up to twelve read ports to execute all the micro-ops of CMOP simultaneously. Providing so many read ports to the physical register file is certainly not complexity-effective. Hence, in order to deal with this, we propose a separate register file contained in the PFU.

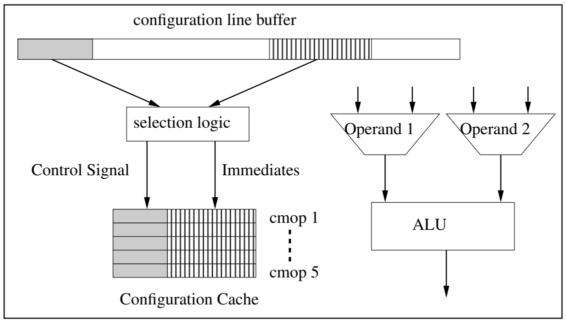


Figure 5: Processing Element

This internal register file is distributed in order to provide sufficient bandwidth, as shown in Figure 4. IRF contains multiple register file sets, each of which is allocated to a MOP in the dispatch stage. A register file set contains replicated copies of register file, one copy corresponding to each row. Each register file has 4 entries and has 4 read ports and 4 write ports. Recollect, that the CMOP writes to the physical register file. Hence, the write ports on IRF are used by the ld-set and mv-set only.

There are 5 different register file sets, so inputs for 5 different MOPs can be stored at the same time. Hence, the total size of this IRF is 60 (number of entries per RF*number of rows*number of RF sets = 4*3*5). Dispatch stalls in case a register file set cannot be allocated to the MOP. It is obvious from such a distributed organization that a PE could access only the register file of the row that it belongs to and to that of the MOP that is currently being executed. The values in the IRF are discarded only when CMOP is successfully executed.

3.2.3 PE and Configuration Cache

Figure 5 provides a deeper look into the PE. Each PE contains 1) an ALU, which is connected to the ALUs of following rows, and 2) a configuration cache. Configuration cache holds configuration of 5 CMOPs in a distributed manner. The configuration contains pre-decoded control signals of all the fused micro-ops pertaining to the CMOP as shown in the Figure 6. Configurations are 32 bytes long and is equal to half of L1 I-cache line size. The lower 16 bytes contains opcode and source operand information of all the fused micro-ops, and the upper 16 bytes holds the immediate operand values.

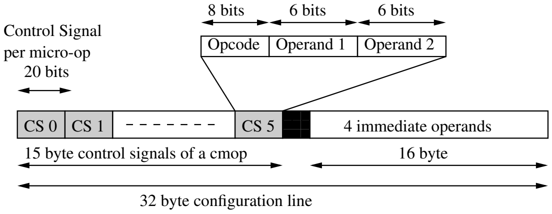


Figure 6: Configuration Line

A direct access is made to the L1 I-cache to read in the configuration line corresponding to a CMOP. The configuration line is then distributed to all the PEs. Each PE contains also a line buffer to store the configuration line. The PE then selects the appropriate micro-op control signal and immediate operand, if any, from the configuration line buffer. The control signals are stored in the configuration cache.

3.2.4 Bypass Network

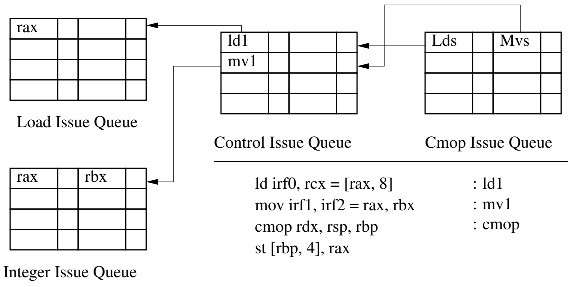
To support back-to-back execution, all the 6 PEs should receive source operands from the bypass. The PEs, however, receive inputs only from the 2 load units (LDUs) and 2 ALUs. In the evaluation Section 5, we however show that not all the 6 PEs need the source operands to be bypassed. For a 2x3 grid a bypass network to 4 PEs is more than sufficient.

On the other hand, a significant fraction of execution cycle of an ALU in a modern out-of-order processor is consumed by the destination operand forwarding [7, 13]. Hence, in order to support a PFU that collapses three ALUs and execute with low latency, we remove the forwarding logic from PFU to other ALUs, and dedicate this fraction of execution cycle completely to execution. Our studies indicate that such a constraint has negligible impact on performance.

3.2.5 Pipeline Stages

- **Rename** The width of a typical out-of-order processor determines the number of micro-ops that could be renamed. For instance, a 4-wide machine could rename up to four micro-ops per cycle. However, in MOP model we constrain renaming to the number of registers and not to the micro-ops. So, if a CMOP has four destination registers then only the CMOP is renamed in that cycle. However, if a CMOP requires two destination registers, two other micro-ops can be renamed in the same cycle.

- **Dispatch** Loads of the ld-set go to the traditional Issue queue, and an entry in the control issue queue is allocated for each ld-set. The control issue queue entry contains issue queue tags of all the loads in the ld-set. The same holds true for all the moves in the mv-set. This hierarchical issue queue model is described below, and illustrated in Figure 7.



```
ld irf0, rcx = [rax, 8]        : ld1
mov irf1, irf2 = rax, rbx      : mv1
cmop rdx, rsp, rbp             : cmop
st [rbp, 4], rax
```

Figure 7: Hierarchical Issue Queue Model

The labels against each issue queue entry indicates the issue queue tag associated with the entry. For instance, Ld1 is the tag associated with the first load in the load set. Control issue queue entry corresponding to tag Lds depends upon both the Ld1 and Ld2 issue queue entries. Similarly, CMOP depends upon the Lds and Mvs control issue queue entry tags. Ld1 and Ld2 issue queue tags are broadcast to the control issue queue and Lds issue queue tag is broadcasts to CMOP issue queue, where CMOPs are held. CMOP's dependence with ld-set and mv-set entry is built at runtime using information encoded in the CMOPs. Such a model ensures that CMOP issues only when both the ld-set and the mv-set have issued, without having the need of explicit source operand encoding in a CMOP.

4 Code Generation

Co-designed virtual machine (Cd-VM) plays an important role in dynamically optimizing the code for an efficient use of the Programmable Functional Unit (PFU). Hardware assisted block and edge profiling [4] is performed while running the application. Once a particular basic block becomes hot, the Cd-VM is invoked. The Cd-VM then creates a superblock, optimizes the code in the superblock, and finally stores the generated code in the code cache. Figure 8 shows the flow chart of the whole process.

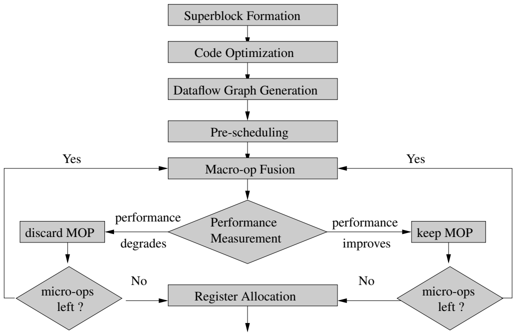


Figure 8: Code Generation Flow Chart

Few of the code generations steps, such as superblock formation, code optimization, dataflow graph generation and register allocation, are standard steps in most dynamic compiling systems. Hence, we will discuss only the steps that we have introduced in the code generation process.

4.1 Pre-Scheduling

The pre-scheduling step helps in aggressively reordering micro-ops, including load and stores, and thus, it creates more opportunities for the algorithm that combines micro-ops into MOPs.

4.2 Macro-op Fusion

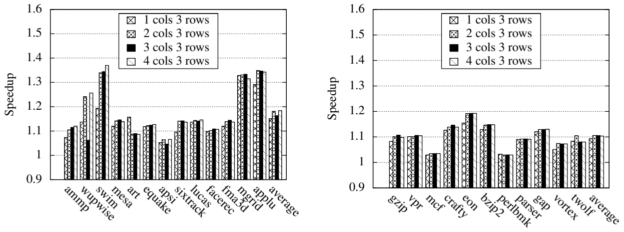
After pre-scheduling the code, the Cd-VM traverses the dataflow graph in the pre-scheduled program order to select appropriate micro-ops to be combined into complex macro-ops. While traversing the code, micro-ops are fused into the CMOP. In particular, a micro-op is placed such that its predecessors are always placed in rows above it. The proposed schemes tries to fit a MOP with as many micro-ops as possible.

Once a MOP is formed, the performance of the actual state of the generation is estimated. The performance function model is based on a scheduling step that estimates the number of cycles of the code prior to the inclusion of the current MOP and compares it with the code after the MOP is included. In particular, every time a MOP is formed, the MOP and the remaining code is scheduled. If the current MOP degrades performance, the MOP is discarded. The algorithm then reiterates over the remaining micro-ops and exits when all the micro-ops have been considered.

5 Performance Evaluation

5.1 Experiment Methodology

We have implemented the proposed PFU model including the Cd-VM in a modified version of PTLSim[19]. The simulated processor is a modern 4-way out-of-order processor. Table 1 provides detailed information of the microarchitecture of the simulated processor and the proposed PFU configuration.



(a) SPECFP

(b) SPECINT

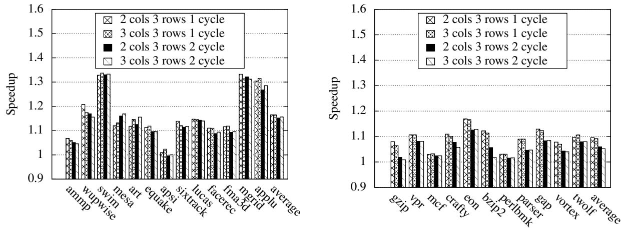
Figure 9: Impact of grid size

Common Parameters	
I-Cache	16 KB, 4-way, 64 B line, 2 cycle access
Branch Predictor	Combined Predictor 64 K 16-bit history 2-level, 64 K bi-modal, 1K RAS
Fetch Width	4 micro-ops / x86 instructions up-to 16 bytes long
Issue Width	4 (2 loads, 2FP, 2 INT, 2 st, 1 CMOP)
L1 Data Cache	32 KB, 4-way, 64 B line, 2 cycles
L2 Cache	256 KB, 16-way, 64 B line, 6 cycles
L3 Cache	4 MB, 32-way, 64 B line, 14 cycles
Main Memory	154 cycles
Out-of-order Parameters	
Rename	8 source, 4 destination operands
Issue Queue	16 entry FP, Int, Mem, CMOP and Control
Functional Units	2 LDU, 2 ALU, 2 AGU, 2 FPU, 1 PFU
Register File	128 entry INT RF, 128-entry FP RF, 4 write ports each
ROB	128 entry
LSQ	80 entry (48 loads + 32 stores)
Load Fill Request Queue	8 entry
Miss Buffer	8 entry
PFU Parameters	
Grid size	2 columns, 3 rows
Internal Register File	5 RF sets, 3 RFs per set, 4 entries per RF, 4 read and 4 writer ports per RF
Configuration Cache	5 (7 byte) entries per PE
Execution Latency	1 or 2 cycles

Table 1: Baseline processor configuration

We have evaluated the proposed scheme using the SPEC2000 benchmark suite. These benchmarks have been compiled with gcc version 4.1.3 using -O3. Using the developed infrastructure, we have simulated the benchmarks for 100 million x86 instructions after the initialization and a cache warm-up period of 2 million x86 instructions.

In this evaluation we have studied the performance of the proposed PFU using different PFU configurations: varying the grid size, and constraining the number of PEs that could receive data from the bypass network. We have also studied the impact of code optimization heuristics on performance.



(a) SPECFP

(b) SPECINT

Figure 10: Impact of varying PFU latency

5.2 Impact of Microarchitectural Constraints

PFU as described earlier is a grid of PEs. We try to vary the number of columns in the grid from 1 to 4, but keeping the number of rows fixed to 3 as shown in Figure 9. In SPECINT the 2x3 grid is the best performing configuration. For SPECFP, the 2x3 grid provides performance close to that provided by a 4x3 grid. Therefore, we choose 2x3 grid configuration for PFU.

Note, however, that in some cases increasing the number of columns in the grid (e.g. wupwise, applu, twolf) results in a lower performance improvement. The main reason for this is the fact that the heuristic for fusion, fuses as many micro-ops possible. Thus, in some cases, micro-Ops that are independent are also fused,and so delay in input for one micro-op delays the execution of the CMOP.

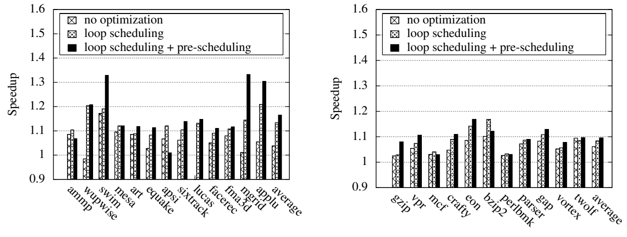
To support back-to-back execution of CMOP, register operands should be bypassed to each of the PEs of PFU. This bypass however is needed only from the ALUs in which the mv-set executes and from the LDUs where the loads execute. A fully connected bypass network, where data is bypassed to all the PEs of PFU is not complexity-effective. Our simulation results suggests a design point with both the PE in the first row and 1 PE each in the second and the third row provide performance within 0.5% of a configuration where all the PEs are connected to the bypass.

On the other hand, PFU execution latency also is another important factor that contributes to performance gain. After all, fusing a chain of micro-ops and executing them in fewer cycles have been shown to provide benefit [3, 8, 17, 18]. Interlock collapsing ALUs [8] collapses two ALUs and execute in a single cycle. Moreover, as mentioned in Section 3.2.4, we do not introduce forwarding logic from PFU to other ALUs is removed. Based on the above observations, we consider two PFU execution latencies of 1 cycle or 2 cycles.

Figure 10 shows the effect of execution latency on a 2x3 and 3x3 grid configuration. SPECINT is particularly sensitive to this increase in latency. Mesa, however, shows a reverse trend, recollect that our code fusion algorithm tracks the performance of fusion by scheduling the generated instructions. If the fused MOP degrades the performance it is discarded. This results in few MOPs being generated, which provides better performance than aggressively fused MOP.

5.3 Impact of Code Optimization Schemes

In a HW/SW co-designed processor Cd-VM plays an important role in improving performance as well. This impact in performance due to code optimization is clearly visible in all the benchmarks as shown in Figure 11. In many benchmarks, such as lucas, fusion without optimization actually slows down the application. Recollect that our MOP-fusion algorithm tries to fit in as much micro-ops as possible. As a result of which artificial dependencies are created. This results in delaying the critical path of the application and causing a slowdown.



(a) SPECFP

(b) SPECINT

Figure 11: Impact of Code Optimization

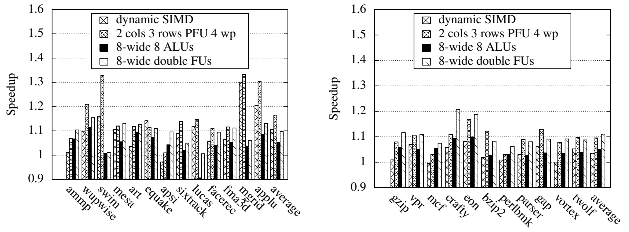
As mentioned in Section 4, performance of fusion is evaluated using an objective function based on a scheduler. However, the scope of this monitoring is limited to the superblock. Getting a global scope for all superblocks is a cumbersome task, however, in case of superblocks that are loops its fairly possible. Hence, in order to get a semi-global scope in case of superblocks with loops multiple iterations of the superblocks are scheduled. The impact of this optimization can be seen from Figure 11 by the bar with tag loop scheduling. Notice the gain in performance in wupwise, mgrid and applu which have many superblocks that are loops.

As discussed earlier in Section 4, fusion is preceded by pre-scheduling. Superblocks are aggressively reordered in the pre-scheduling phase and a new program order is obtained. The impact of pre-scheduling step can also be seen from Figure 11 in the third bar with tag loop scheduling + pre-scheduling. This aggressive re-ordering can have negative impact when some store instructions are pushed down as evident in ammp, apsi and bzip2.

5.4 Comparison with alternate designs

As mentioned earlier chaining of FUs has been shown to provide performance benefits[3, 8]. Also at the same time vector functional units increases performance[1, 12]. Our proposal tries to leverage these two facts in a more efficient way with little change over the baseline architecture.

First, we dynamically create SIMD instructions using similar code generation heuristics as discussed earlier. The SIMD instructions considered are SSE2 and hence act upon 4 32-bit operands in parallel. The impact of SIMD units on performance is shown in Figure 12 by the bar with dynamic SIMD tag.



(a) SPECFP

(b) SPECINT

Figure 12: Comparison with SIMD and 8-way out-of-order

Wider machines are known to exploit the inherent ILP and provide performance benefits. Code-optimization heuristics like code reordering as well exploit this inherent ILP by generating code with data parallel instructions being executed together. Hence, we compare our PFU based proposal to two variants of 8-wide machine.

A 2x3 PFU grid adds 6 ALUs to the baseline processor. Hence, in the first 8-wide issue variant as shown in Figure 12, by the bar 8-wide ALU, we add 6 more ALUs to the baseline processor and increase the issue width from 4 to 8. Clearly, a HW/SW co-designed approach outperforms an 8-wide issue machine. Note that, in this 8-wide issue variant the bypass is provided from all the FUs to all the newly added ALUs. Whereas in our model we provide a limited bypass as evaluated in the section 5.2.

We also consider a more aggressive design of 8-wide issue machine, in which all the FUs are doubled, including memory and complex FUs eg: FPDIV. Such an 8-wide issue machine performs better in case of SPECINT and is evident in Figure 12 by the 8-wide double FUs bar. Adding more store and load units has a stronger impact in performance in SPECINT. However, our proposal outperforms in case of SPECFP without doubling any memory or complex FUs.

5.5 Overhead of Co-designed Virtual Machine

To measure the overhead of Super Block Translation we used estimates provided in [9, 6]. DAISY [6] reported an estimate of nearly 4000 source instructions to optimize a single source instruction. They however, mention that this a very conservative estimate and quote a reasonable estimate to be 1000 instructions. S. Hu [9] measured the overhead to be 1000 x86 instructions per x86 instruction.

We used this esmtimate to measure the startup overhead of Superblock translation and optimization. This overhead turned out to be very low less than 1 percent in our case. Note that, the overhead related to cold code translation is removed due to the use of dual-mode decoders in the frontend.

6 Related Work

6.1 Dynamic Optimization

Performing dynamic optimization in a user transparent manner is not new. Several hardware approaches have been considered [14]. The main difference between our scheme and theirs is that a co-designed approach uses a software layer to perform these optimizations. This has several advantages including flexibility to perform different optimizations and analysis with reduced hardware complexity.

The concept of a co-designed VM is also used by Transmeta Crusoe [11, 5] processors to execute x86 code on top of a VLIW processor, and by IBM DAISY [6] to execute PowerPC code on top of a VLIW processor. There are two main differences between these schemes and the one described in our proposal. First, we eliminate start-up overheads by eliminating the interpretation/translation step of a typical co-designed VM. Secondly, prior projects assume a VLIW target processor, while we target an x86 processor out-of-order design.

6.2 Accelerators

Many different types of fine-grained accelerators [3, 18, 17, 8] have been proposed to improve performance in a complexity-effective way. Some of them [3, 17] are designed requiring changes to the ISA and they are programmed using a static code generation scheme. This, in contrast to our approach demands an effort in forward compatibility. Applications evolve and so, accelerators may require some changes from generation to generation. Therefore, adding instructions at ISA level may pose important constraints for future processor generations.

Dynamic and transparent management of these accelerators have been studied in the past. Most of the works [3, 18] have focused on managing accelerators with hardware-based dynamic optimizers. A more limited amount of work has considered the use of co-designed VM for assigning instructions to an accelerator. However, their work focused on using a 3-1 ALU [8]. In contrast, in this paper we focus on exploiting the benefits of a larger and powerful accelerator over larger regions.

On the other hand a recent work have considered the benefits of using a co-design virtual machine to deal with the changes on the SIMD vector ISA from generation to generation [2]. In contrast, in this paper we proposed a new PFU and we show that this scheme outperforms SIMD-based accelerators. As far as we know this is the first work where these alternatives are compared.

7 Conclusions

A HW/SW co-designed approach is a complexity-effective way of providing high performance for general purpose application. A Cd-VM reorders a superblock and fuses a sequence of micro-ops to generate a MOP. The fusion is done by taking into account the existing microarchitectural resources and performance of fusion is monitored. A split-MOP model allows inputs to be provided both from memory and conventional register file.

A novel PFU executes the CMOP design is proposed which exploits both ILP and chaining to gain performance. We obtain average speedups of 17% in SPECFP and 10% in SPECINT. We also obtain speedups of up-to 33 % is some benchmarks.

We measure the impact of various microarchitectural constraints on performance. We also demonstrate that, by introducing some code generation scheme performance is improved. We also show that our split-MOP model outperforms not only a dynamic SIMD machine but also 8-wide issue machine. Hence, we conclude that a new generation of out-of-order processors can be co-designed for higher performance in a complexity-effective way.

References

[1] SSE extension : Intel IA 64 and IA-32 Architectures Software Developer's Manual, 1997.

[2] N. Clark, A. Hormati, S. Yehia, S. Mahlke, and K. Flautner. Liquid simd: Abstracting simd hardware using lightweight dynamic mapping. In *IEEE Intl. Symp. on High-Performance Computer Architecture*, 2007.

[3] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner. Application-Specific Processing on a General-Purpose Core via Transparent Instruction set customization. In *IEEE Intl. Symp. on Microarchitecture*, 2004.

[4] T. Conte, A. Patel, and J. Cox. Using branch handling hardware to support profile-driven optimization. In *IEEE Intl. Symp. on Microarchitecture*, 1994.

[5] J. Dehnert, B. Grant, J. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing Software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *IEEE Intl. Symp. on Code Generation and Optimization*, 2003.

[6] K. Ebcioglu and E. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *IEEE Intl. Symp. on Computer Architecture*, 1997.

[7] E. Fetzer, M. Gibson, A. Klein, N. Calick, C. Zhu, E. Busta, and B. Mohammad. A fully bypassed six-integer datapath and register file on the itanium-2 microprocessor. In *IEEE Intl. Journal of Solid-State Circuits*, 2002.

[8] S. Hu, I. Kim, M. Lipasti, and J. Smith. An approach for implementing efficient superscalar CISC processors. In *IEEE Intl. Symp. on High-Performance Computer Architecture*, 2006.

[9] S. Hu and J. E. Smith. Reducing startup time in co-designed virtual machines. In *In Proc. of the 33rd Annual International Symposium on Computer Architecture*, 2006.

[10] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *THE JOURNAL OF SUPERCOMPUTING*, 1993.

[11] A. Klaiber. The technology behind Crusoe Processors, 2000.

[12] S. Oberman, G. Favor, and F. Weber. AMD 3Dnow! Technology: Architecture and implementations. *IEEE/MICRO*, 1999.

[13] S. Palacharla, N. Jouppi, and J. Smith. Complexity-effective superscalar processors. In *IEEE Intl. Symp. on Computer Architecture*, 1997.

[14] S. Patel and S. Lumetta. rePLay: A Hardware Framework for Dynamic Optimization. *IEEE Transactions on Computers*, 2001.

[15] J. Smith and R. Nair. *Virtual Machines: A Versatile Platform for Systems and Processes*. Elsevier Inc., 2005.

[16] F. Tseng and Y. Patt. Achieving Out-of-Order Performance with Almost In-Order Complexity. In *IEEE Intl. Symp. on Computer Architecture*, 2008.

[17] Z. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *IEEE Intl. Symp. on Computer Architecture*, 2000.

[18] S. Yehia and O. Temam. From sequences of dependent instructions to functions: An approach for improving performance without ILP or speculation. In *IEEE Intl. Symp. on Computer Architecture*, 2004.

[19] M. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *IEEE Intl. Symp. on Performance Analysis of Systems and Software*, 2007.