Exploiting Virtual Registers to Reduce Pressure on Real Registers

JUN YAN and WEI ZHANG

Southern Illinois University Carbondale

It is well known that a large fraction of variables are short-lived. This paper proposes a novel approach to exploiting this fact to reduce the register pressure for pipelined processors with data-forwarding network. The idea is that the compiler can allocate virtual registers (i.e., place holders to identify dependences among instructions) to short-lived variables, which do not need to be stored to physical storage locations. As a result, real registers (i.e., physically existed registers) can be reserved for long-lived variables for mitigating the register pressure and decreasing the register spills, leading to performance improvement. In this paper, we develop the architectural and compiler support for exploiting virtual registers for statically scheduled processors. Our experimental results show that virtual registers are very effective at reducing the register spills, which, in many cases, can achieve the performance close to the processor with twice number of real registers. Our results also indicate that, for some applications, using 24 virtual, in addition to 8 real registers, can attain even higher performance than that of 16 real without any virtual registers.

Categories and Subject Descriptors: C.5.3 [Processor Architecture]: Computer System Implementation—*Microcomputers*

General Terms: Design, Performance, Experimentation

Additional Key Words and Phrases: Register file, data forwarding, short-lived variables, virtual register, register allocation

ACM Reference Format:

Yan, J. and Zhang, W. 2008. Exploiting virtual registers to reduce pressure on real registers. *ACM Trans. Architect. Code Optim.* 4, 4, Article 22 (January 2008), 18 pages. DOI = 10.1145/1328195.1328198 <http://doi.acm.org/10.1145/1328195.1328198>

1. INTRODUCTION

The register file is on the top of the memory hierarchy, which provides the fastest way for the processor to access data. Registers are typically managed by the compiler by using sophisticated register allocation algorithms [Muchnick 1997] to ensure that most frequently used variables can be stored in the registers.

Authors' address: Jun Yan and Wei Zhang, Department of Electrical and Computer Engineering, Southern Illinois University Carbondale, Carbondale, Illinois 62901; email: {jun,zhang}@engr.siu.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org. © 2008 ACM 1544-3566/2008/01-ART22 \$5.00 DOI 10.1145/1328195.1328198 <http://doi.acm.org/10.1145/1328195.1328198>

ACM Transactions on Architecture and Code Optimization, Vol. 4, No. 4, Article 22, Publication date: January 2008.

However, if there are more overlapping live variables than the number of registers available, some variables have to be spilled to the memory, which can increase both the execution cycles and energy dissipation. Therefore, a large number of registers, if used effectively, can benefit both performance and energy consumption.

Previous study suggests that 16–32 registers are sufficient to support a high instruction execution rate for scalar programs [Mahlke et al. 1992]. However, recent advances in both compiler and architecture technology call for larger register files. On the compiler side, aggressive code motions can increase variable lifetime. ILP (instruction-level parallelism)-oriented optimizations, such as software pipelining and loop unrolling, also demand more registers for variables that will be used concurrently. Furthermore, scalar replacement (also called register pipelining) [Callahan et al. 1990] can expand the register allocation candidates to array elements, which can further increase the register pressure. On the architecture side, very long instruction word (VLIW), explicitly parallel instruction computing (EPIC), multithreaded or windowed-register architectures typically require many more registers [Yung and Wilhelm 1995]. For instance, Intel IA-64 provides 128 general-purpose, 128 floating-point, 8 branch, 64 predicate, and up to 128 special-purpose application registers [Huck et al. 2000].

On the other hand, large, multiported register files are likely to compromise the clock cycle time of high-performance processors. There have been a number of proposals to circumvent this problem [Farkas et al. 1997; Zyuban and Kogge 2001; Balasubramonian et al. 2001; Postiff et al. 2001; Cruz et al. 2000; Tseng and Asanovic 2003]. One scheme is to partition the microarchitecture into distributed clusters so that each cluster only needs to contain a subset of the register file and functional units [Farkas et al. 1997; Zyuban and Kogge 2001]. While this clustered microarchitecture is scalable to large issue width; it requires very complex control logic to steer the instructions to each cluster and it also needs to cope with the intercluster synchronization and communication [Tseng and Asanovic 2003]. Other approaches retain a centralized microarchitecture, but split the large register file into register caches [Balasubramonian et al. 2001; Postiff et al. 2001] register banks [Balasubramonian et al. 2001; Cruz et al. 2000; Tseng and Asanovic 2003]. While these approaches can reduce the complexity of large, multiported register files, they can also cause other problems that may impact performance, such as register cache misses or register bank contentions. Moreover, all these techniques [Balasubramonian et al. 2001; Postiff et al. 2001; Cruz et al. 2000; Tseng and Asanovic 2003] target the physical registers used for the dynamic register renaming in superscalar processors, which are not visible to the architecture. Thus, these techniques cannot be directly applied to statically scheduled architectures without the dynamic register renaming, such as VLIW, which needs to extend the architecturally visible registers that can be managed by the compiler.

In this paper, we propose a novel scheme to exploit the short-lived variables and the data-forwarding network to minimize the register pressure. Previous work has found that a large fraction of variables are short-lived in the sense that their live ranges span only a few instructions [Lozano and Gao 1995; Hu

and Martonosi 2000]. In a pipelined processor, data can be forwarded from the output of a functional unit to the inputs of the same or other functional units in the subsequent clock cycles. Therefore, if the live range of a variable is within the length of the data forwarding path, it does not need to be written back to the architectural registers. Moreover, *this variable does not even need to be allocated to an architectural register*, since a place holder is sufficient to identify the data-dependence relationship for forwarding the data to the right instructions. In this paper, the place holders (i.e., tags) in the operand fields of the ISA are called virtual registers. In contrast, the architectural registers that have physical storage places in the register file are called real registers (also called register for simplicity).¹ Since a large fraction of variables are short-lived (see Figure 4 for details), this paper proposes to allocate virtual registers (instead of real registers) to the short variables, whose live ranges are smaller than or equal to the maximum length of the data-forwarding path. As a result, the real registers can be used for other variables more efficiently to reduce the register spills. Our experimental results indicate that virtual registers are very successful in reducing the register spills, leading to higher performance without physically enlarging the register file.

The rest of the paper is organized as follows. We discuss related work in Section 2 and introduce the basics of data forwarding network in Section 3. We elaborate our approaches to exploiting virtual registers for reducing the real register pressure in Section 4. The experimental results are given in Section 5. Finally, we conclude this paper in Section 6.

2. RELATED WORK

There have been several research efforts on exploiting the short-lived variables for performance [Lozano and Gao 1995] and energy optimizations [Hu and Martonosi 2000; Sami et al. 2001]. While the work in [Hu and Martonosi 2000; Sami et al. 2001] focused on reducing the register file energy dissipation, the work in [Lozano and Gao 1995] is closest to this study in that both explore the short-lived variables to improve the register allocation for better performance; however, there are some important distinctions. Lozano and Gao [1995] proposed an architecture extension to avoid the useless commits of the values produced by short-lived variables and a compiler-based approach to avoiding assigning the short-lived variables to the real registers. In their work [Lozano and Gao 1995], the reorder buffer entries are regarded as the extension of real registers, which can be allocated to short-lived variables by the compiler to improve the utilization efficiency of real registers. However, this approach does not apply to VLIW processors, which do not have reorder buffers. In contrast, this paper uses virtual registers that do not physically exist to help enhance the allocation of real registers to long-lived variables. Second, our approach

¹It should be noted that physical registers typically refer to the architecturally invisible registers used for dynamic register renaming in dynamically scheduled processors, although they are also real. This paper, however, focuses on studying a compiler-based approach to reducing the pressure on architecturally visible real registers for statically scheduled processors, which typically do not have architecturally invisible physical registers.

can push Lozano and Gao's scheme further in that in superscalar processors, certain short-live variables do not even need to be allocated to reorder buffer entries, which can use virtual registers instead. Therefore, the number of reads from and writes to the reorder buffer is reduced, which can decrease the stalls due to reorder buffer saturation and also improve the energy efficiency. Another difference is that in [Lozano and Gao 1995], the short-lived variables are also defined as those whose live ranges are smaller than or equal to the size of the reorder buffer (which is typically much larger than three). By comparison, the short-lived variables in this paper refer to the variables whose live ranges are smaller than or equal to the length of the data-forwarding path (i.e., three in our simulated processor, see Section 3 for details).

Yung et al. [1955] proposed register scoreboard and caching to exploit the locality in register values for windowed-register architectures. Yung's approach, however, still needs to allocate real registers to all the variables, although the short-lived variables will stay in the register cache for a short period of time at runtime. In contrast, our approach is based on compiler, which can allocate the limited real registers more efficiently by binding short-lived variables to virtual registers.

Recently, Oehmke et al. [2005] proposed the virtual-context architecture (VCA) to support both multithreading and register windows with significantly fewer registers than a conventional machine. VCA builds on the register renaming logic already found in dynamically scheduled processors, which treats the physical registers as a cache of a much larger memory-mapped logical register space. Therefore, VCA cannot be directly applied to the architectural registers of statically schedule processors (e.g., VLIW) without dynamic register renaming support; otherwise, the hardware simplicity and scalability may be compromised. Moreover, by applying the virtual register-based approach proposed in this paper to the context of VCA, the register pressure can be further mitigated to potentially achieve even better performance of VCA.

Gonzalez et al. [1998] introduced the concept of virtual-physical registers, which are the names used to keep track of dependences. The virtual physical registers [Gonzalez et al. 1998] and virtual registers proposed in this paper are similar in the sense that both do not use any storage locations. However, there are some significant differences. First, virtual physical registers [Gonzalez et al. 1998] are used to delay the allocation of physical registers (not architectural registers) in dynamically scheduled processors, while virtual registers proposed in this paper can be exploited to mitigate the pressure on architectural registers for any processors, regardless of the dynamic register renaming. Second, virtual physical registers [Gonzalez et al. 1998] are invisible to the compiler, while virtual registers can be explicitly managed by the compiler to reduce the register spills physically. Monreal et al. [2004] studied VP-LAER, which is a rename scheme that can allocate physical registers later and release them earlier than traditional rename schemes. However, VP-LAER targets the dynamic register renaming for superscalar processors, while virtual registers proposed in this paper can reduce the register pressure for VLIW processors without using dynamic register renaming.

Fig. 1. The five-stage pipelined VLIW forwarding architecture, where DEC stands for decoder and LSU represents the Ld/St Unit.

3. BACKGROUND ON DATA-FORWARDING NETWORK

A data-forwarding (also called bypassing) network is a widely used mechanism to reduce the data hazards of pipelined processors [Patterson and Hennessy 2004]. A data-forwarding network has also been used in VLIW processors, such as Philips LIFE processor [Kannan et al. 2002]. In this paper, we study a VLIW processor with a five-stage pipeline provided with data-forwarding logic, as shown in Figure 1.² The pipeline stages are the follows:

- IF: Instruction Fetch from the I cache.
- ID: Instruction Decode and operand fetch from the Register File (RF).
- EX: Execution.
- MEM: Memory access for load/store operations.
- WB: Write-Back results into the RF.

As can be seen from Figure 1, three forwarding paths (i.e., EX-EX, MEM-EX and MEM-ID) provide direct links between pairs of stages through the pipeline registers (i.e., EX/MEM and MEM/WB interstage pipeline registers). Therefore, given a sequence of VLIW instructions $I_1 \dots I_k \dots I_n$, the instruction I_k can read its operands from the following instructions:

- I_{k-1} : through the EX-EX forwarding path (when I_k is in the EX stage).
- I_{k-2} : through the MEM-EX forwarding path (when I_k is in the EX stage).
- I_{k-3} : through the MEM-ID forwarding path (when I_k is in the ID stage).
- I_{k-j} : through the RF where $j > 3$.

3.1 Problem Formulation

Given an instruction pipeline with a data-forwarding network F , we define the length of the data-forwarding path $L(F)$ to be the maximum number of clock cycles between the producer instruction that forwards the value and the consumer instruction that takes this value. Given a variable v , we define its

²For simplicity, we just illustrate the data-forwarding path for a scalar pipeline.

```

tdist = (x - maxc0) * C0_SCALE;
min_dist = tdist*tdist;
tdist = (x - minc0) * C0_SCALE;
max_dist = tdist*tdist;

```

(a)

```

(1) sub $tr1 $x $maxc0;
(2) mul $tdist $tr1 C0_SCALE;
(3) nop;
(4) nop;
(5) mul $min_dist $tdist $tdist;
(6) nop;
(7) nop;
(8) sub $tr2 $x $minc0;
(9) mul $tdist $tr2 C0_SCALE;
(10) nop;
(11) nop;
(12) mul $max_dist $tdist $tdist

```

(b)

Fig. 2. The code segment: (a) the source code; (b) the assembly code.

live range $LR(v)$ to be the number of machine instructions (including nops) in the longest path between its definition points and last-use points, including either the instruction at the definition point. We say a variable v is a short-lived variable if its live range is smaller than or equal to the length of the data-forwarding path, i.e., $LR(v) \leq L(F)$.

Given these definitions, the problem to be studied in this paper can be described as follow:

Problem—Reducing Pressure on Real Registers: Given a pipelined machine M with the length of a data-forwarding path $L(F)$ and a program P , create a register allocation such that the short-lived variables in P will use as few real registers in M as possible to minimize the register spills without violating the program semantics.

4. OUR SOLUTION

To solve the aforementioned problem, we propose to create virtual registers for short-lived variables. A virtual register is simply a tag to identify the data dependences among instructions, which is not corresponding to a physical storage location in the register file. Since the virtual registers can keep track of the data-dependence relations, the value of short-lived variables can be forwarded to the right consumer instructions without violating the program semantics.

The idea of exploiting virtual registers can be explained by the example code shown in Figure 2, which is extracted from the source code of `cjpeg`. Figure 2a gives the source code and Figure 2b gives the corresponding assembly code. The live range of each variable is depicted in Figure 3. As can be seen, this code segment includes three short-lived variables, i.e., `$tr1`, `$tr2`, and `$tdist`, since their live ranges are less than or equal to three. Without virtual registers, at least three real registers are needed. However, by exploiting the virtual registers, only two real registers (in addition to one virtual register) are sufficient for this code segment, as can be seen from Figure 3. Therefore, in case there are only two real registers available for this code segment, utilizing the virtual registers

Fig. 3. Live ranges and the register allocation of the code segment without and with virtual registers, where *pr* stands for physical registers and *vr* represents virtual registers.

for short-lived variables can reduce the number of register spills, potentially leading to higher performance.

Clearly, the success of our approach depends on how many variables are actually short-lived. If only a small fraction of variables are short-lived, its impact on the real register allocation and the reduction of register spills will be insignificant. Since in our five-stage pipelined-forwarding architecture, as depicted in Figure 1, the length of the forwarding path is three; only the variables whose live ranges are equal to or less than three will be identified as short-lived, according to our definition of short-lived variables in Subsection 3.1. Figure 4 shows the percentage of short-lived variables for the selected benchmarks from Mediabench [Lee et al. 1997] and SPEC 2000 [spe 2006] (more details of our evaluation methodology can be found in Section 5). As we can see, except *rasta*, all the applications have more than 50% of short-lived variables. On average, 66.2% of variables are short-lived, indicating good opportunities to exploit virtual registers for mitigating real register pressure.

4.1 Architectural Support

To support the exploitation of virtual registers, we need slight extension of the underlying hardware. The assumed architectural support is depicted in Figure 5b. Compared with original ISA shown in Figure 5a, 1 bit (called the virtual bit) needs to be added into the ISA for each destination and source operand field. By default, the leftmost virtual bit is set to be 0; the remaining 5 bits of the operand field will be used to specify a real register. When the virtual bit is set to 1, a virtual register (i.e., the tag consisting of all the 6 bits) will be used to identify the data-dependence relations associated with short-lived variables. The decoder also needs to be extended to recognize the additional virtual bits. Clearly, adding 1 virtual bit to each of the three operand fields of a 32-bit instruction format will increase the instruction width by 9.3%, which

Fig. 4. Percentage of short-lived variables.

Fig. 5. The architecture support for virtual registers: (a) the original instruction format; (b) the extended instruction format.

also requires the corresponding increase of the instruction memory size. This overhead can be partly mitigated by exploiting unused instruction encoding bits; however, this will restrict the proposed approach only to the subset of ISA operations with unused encoding bits in the instruction format. With the increasing use of 64-bit architectures for high-end microprocessors, this overhead is also expected to be reduced (i.e., 4.7% for adding the same number of virtual registers as the real registers without doubling the physical size of the register file). Moreover, for embedded processors that provide multiple ISAs, such as ARM7TDMI [arm 1995], the wider ISA can be exploited to support virtual registers for better performance, while the number of real registers is kept small and fixed.

In addition, we add a Rs Inhibit bit and a Rt Inhibit bit to the register file, which, when enabled (i.e., 1), will inhibit reading the Rs or Rt register, respectively. As can be seen from Figure 5b, the value in the virtual bit of the

Rs field and the Rt field will be passed to the Rs and the Rs Inhibit bits directly. Therefore, when a virtual register is used as the source registers, it will not be read from the register file. Instead, the source operands will be forwarded from the interstage pipeline registers through the forwarding network. Similarly, the virtual bit of the Rd field is connected with a NOT gate, whose output is then passed to the RegWrite Enable signal that is typically already existed in the register file to enable (1) or disable (0) the register write operations. Therefore, if the virtual bit of the Rd field is 1, i.e., a virtual register is used as the destination register, the RegWrite Enable signal will be set to 0, thus disabling writing the value of a virtual register to the register file. However, it should be noted, that we only show the extension of a single RISC-like operation in this figure. Obviously, a VLIW instruction consisting of multiple operations can be extended in the same manner by adding inhibit bits to each operation.

Since the virtual registers can be encoded in the machine instructions, we do not need to extend the ISA for compilers to handle virtual registers. For instance, in our base processor with 32 real registers, the virtual register number will vary from 32 to 63 (i.e., 100000_2 to 111111_2). Thus if a virtual register is needed for a short-lived variable, the compiler can allocate an available register number between 32 and 63 to it, which will set the virtual bit of the corresponding operand field to 1 in the binary code. With our architectural extension, this virtual bit will ensure that the virtual register will not be read from or written to the register file, while real register operations are not impacted.

4.2 Compiler Support

With the architectural support for virtual registers, the compiler's job is to modify the register allocation to exploit these virtual registers for short-lived variables while freeing real registers for long-lived variables, as shown in Figure 6. Conceptually, the compiler first performs analysis to identify short-lived variables (called the short-liveness analysis in this paper), which are dependent on the length of the data-forwarding path of the processor. Then, the compiler attempts to allocate virtual registers to short-lived variables and to allocate real registers to long-lived variables. In case that there is no virtual variable available, real registers will also be used for the short-lived variables, i.e., these short-lived variables are treated like long-lived variables. In case there is no real register available, a variable must be either spilled or split. The strategy to be used is controlled by the parameter in the configuration file. This process is then repeated until all the variables have been bound to virtual or real registers, or spilled to memory.

In order to make use of virtual registers, the first step is to perform compiler analysis to identify the short-lived variables, which is called the short-liveness analysis in this paper. This analysis consists of three steps: (1) perform standard dataflow analysis to identify def-use chains [Muchnick 1997]; (2) for each variable v , calculate its live range $LR(v)$ based on our definition in Subsection 3.1; (3) identify v as a short-lived variable if $LR(v) \leq 3$; otherwise, v is a long-lived variable. This short-liveness analysis is implemented by using the Rebel intermediate representation of the Elcor compiler [tri 2006].

Fig. 6. The overview of the virtual/real register allocation for both short-lived and long-lived variables.

Our virtual/real register allocation algorithm built upon the region-based register allocation [Kim 2001] of Elcor compiler, which is based on Chow and Hennessy’s priority-based coloring algorithm [Chow and Hennessy 1984]. The framework of this algorithm [Kim 2001] is shown in Figure 7, which includes five major steps: (1) build the interference graph; (2) simplify the graph; (3) prioritize each live range according to the heuristic function [Chow and Hennessy 1984]; (4) color the live range with the highest priority; and repeat from (4) whenever there remains an uncolored live range; and (5) if there is a live range that cannot be colored, reduce the interference of the live range by splitting it into two or more smaller live ranges and repeat from (4). More information about this register-allocation algorithm can be found in Kim’s [2001] thesis.

To minimize the modification on the existing region-based register allocation algorithm, we incorporated our short-liveness analysis and short-lived variable allocation in the *binding()* function, which is part of the step (4) in Figure 7. The pseudocode of this function is provided in Figure 8. In this algorithm, for each variable of a region, if it is a short-lived variable, the compiler tries to allocate a virtual register to it. Otherwise, a real register or spilling has to be used for this variable. The long-lived variables will follow the existing register allocation process without touching virtual registers.

4.3 Precise Interrupt

Exploiting virtual registers in the pipelines can cause problems to the precise interrupt (or exception) [Smith and Pleszkun 1988]. Since virtual registers do not have corresponding physical storages, the values that are associated with virtual registers will be lost after interrupts. Therefore, if an instruction needs to use a virtual register as a source operand, while the value associated with this virtual register is produced by an instruction before the faulting instruction or by the faulting instruction itself; the dependent instruction will not be able to fetch the data correctly. To solve this problem, we propose to store the values in the pipeline registers as part of the processor state when dealing with interrupts or exceptions. We also assume the architecture support for

Fig. 7. The framework of the region-based register allocation.

Algorithm Virtual_Real_Register Allocation (var, Region)

```

vr // virtual register bank
pr // physical register bank
{
  For each var do
    if (short_liveness_analysis(var) == True) then
      if Try(vr, var) = unBlocked then
        Binding(vr, var, status)
      else if Try(pr, var) = unBlocked then
        Binding(pr, var, status)
      else
        Spilling(var, status)
      end if
    end if
  }
}

```

Fig. 8. The algorithm of binding().

NUAL-freeze (nonuniform assumed latency) operations [Schlansker and Rau 2000], which appear not to advance when virtual time is frozen, such as during cache misses or interruptions. The NUAL-freeze operations can be supported by adding snapshot hardware, such as replay buffers [Rudd 1997]. With the support of NUAL-freeze semantics, the operations after exception handling will be executed according to its original virtual time scheduled by the compiler. Since the contents of the pipeline registers are also saved and restored at exactly the same virtual time as if the interruption has never occurred, after the exception handling, the instructions after the faulting instruction can receive the value forwarded from the right pipeline registers based on the original virtual timing.

Table I. Default Parameters Used in Our Simulations

Parameter	Value
Register File	32 registers, 4 read ports and 4 write ports
L1 instruction cache	16 KB direct-mapped cache with 8 sub-banks
L1 instruction cache latency	1 cycle
L1 data cache	32 KB 2-way cache
L1 data cache latency	1 cycle
L1 cache line size	32 bytes
Unified L2 cache	512 KB 4-way cache
L2 cache latency	10 cycles
L2 cache line size	64 bytes
Memory latency	100 cycles
Clock speed	1 GHz

In particular, virtual registers can also be used for short-lived variables, where a load instruction is between the producing instructions and consumer instructions. In case there is a cache miss that is estimated as a hit by the compiler, the pipeline will be stalled by the interlock employed in VLIW processor to ensure correct semantics, i.e., the virtual time is frozen. Since we assume that the operations of the VLIW processor are NUAL-freeze operations, the instruction producing the value is not allowed to produce the value immediately (although it is independent of the load); since the whole pipeline is stalled until the data is returned from the cache. Therefore, when the pipeline is resumed after the load returns, the producer instruction will produce the value and forward the value to the consumer instruction(s) correctly as usual.

5. EVALUATION

5.1 Experimental Setup

We use simulation to evaluate the proposed virtual/real register allocation algorithm on a VLIW processor by using Trimaran v3.7 [tri 2006]. Trimaran is composed of a front-end compiler IMPACT, a back-end compiler Elcor, an extensible intermediate representation (IR) Rebel, and a cycle-level VLIW/EPIC simulator that is configurable by modifying the machine description file [tri 2006]. The virtual/real register allocation algorithm was implemented in Elcor by modifying the existing region-based coloring algorithm [Kim 2001]. The machine description file was configured to simulate VLIW processors with various number of real and virtual registers. The VLIW processor simulated in our experiments consists of two IALUs (integer ALUs), two FPALUs (floating-point ALUs), one LD/ST (load/store), unit and one branch unit. Other system parameters used for our default setting are provided in Table I. The trimaran front-end compiler Impact used optimization level 4 (O4). The back-end compiler Elcor performed machine independent optimizations, including dead code elimination, forward copy propagation, local copy propagation, common subexpression elimination, and loop-invariant code removal, as well as machine-dependent optimizations, such as instruction scheduling and register allocation [tri 2006]. The basic block scheduling algorithm was used as the default scheduling algorithm.

Fig. 9. Normalized execution cycles by using 64 real registers and by using 32 virtual, in addition to 32 real registers, which are normalized with the execution cycles by using 32 real registers.

The benchmark set includes applications drawn from Mediabench [Lee et al. 1997] and the SPEC 2000³ [spe 2006].

5.2 The Effect of Exploiting Virtual Registers

Our first experiment studies the impact of the proposed virtual/real register allocation algorithm on performance of the VLIW processor with 32 real and 32 virtual registers. The percentage of performance improvement (in terms of the total number of execution cycles) relative to the base processor (i.e., 32 physical register with region-based register allocation [Kim 2001]) is illustrated in Figure 9. As can be seen, the exploitation of 32 virtual registers have various effects on performance for different applications. Five of the mediabench applications, i.e., *cordic*, *cjpeg*, *djpeg*, *des* and *unepic*, result in significant performance increases (i.e., reduction of the execution cycles); which vary from 14.5% for *cordic* up to 28.8% for *cjpeg*. On the other hand, two benchmarks, i.e., *g721decode*, and *g721encode* are not impacted by introducing additional virtual registers at all. The rest of the benchmarks have relatively small performance improvement, ranging from 0.7% for *256.bzip2* to 3.1% for *mpeg2dec*.

The diverse effects of virtual/real register allocation on different benchmarks relate to two factors: (1) The number of register spillings that can be reduced by exploiting virtual registers, and (2) the ratio of the number of cycles because of reduced number of register spills relative to the overall number of cycles (also called *ratio of spill cycles* in this paper). In Table II, we give the the number of dynamic register spills with and without using virtual registers, as well as the *ratio of spill cycles*. By comparing the performance improvement in Figure 9 and the data in Table II, we can see that the performance increase of the virtual/physical register allocation is closely related to the *ratio of spill*

³Note that we have included only five SPEC 2000 benchmarks, since the rest of the SPEC 2000 benchmarks cannot be compiled and run correctly in our current experimental framework, based on Trimaran.

Table II. Number of Dynamic Register Spills with 32 Real and 32 Real Registers Plus 32 Virtual Registers and Their Impacts on Performance

Benchmarks	# of Dynamic spills (32)	# of Dynamic spills (32/32)	Ratio of spill cycles
cordic	740152	160184	0.193
cjpeg	7231164	1726496	0.277
djpeg	3543668	2255126	0.222
des	2606502	2151806	0.129
g721decode	28907704	28907704	0.000
g721encode	36873756	36873756	0.000
mpeg2dec	306527313	270691519	0.004
pegwitdec	23727580	22779676	0.020
pegwitenc	35394588	34176536	0.016
rasta	641475	415217	0.046
unepic	2399608	34110	0.225
164.gzip	68118530	23955030	0.019
176.gcc	1454936117	1419677177	0.006
181.mcf	663352446	143701636	0.018
255.vortex	4496129415	3933562637	0.032
256.bzip2	540385384	340472832	0.014
AVERAGE	479469712	391346340	0.076

cycles. More specifically, those five benchmarks (i.e., `cordic`, `cjpeg`, `djpeg`, `des` and `unepic`) that achieve the best performance results also have the highest *ratio of spill cycles*, indicating that using virtual registers is an effective way to boosting performance for applications that suffer greatly from register spills. In contrast, the *ratio of spill cycles* of both `g721decode` and `g721encode` are 0, since these two applications do not have any register spill, given 32 real registers. Obviously, in this scenario, there is no need to exploit additional virtual registers and, hence, there is no room for performance increase by optimizing register allocation. Other benchmarks that also suffer from a limited number of register spills can only be moderately improved by the virtual/real register allocation. Overall, we find that exploiting virtual registers is effective to enhance performance when the register spilling is a severe problem (i.e., the register pressure is high).

Figure 9 also gives the execution cycles of 64 registers, which are normalized with that of 32 registers. As can be seen, the number of execution cycles is reduced by increasing the number of real registers from 32 to 64, except for `g721decode` and `g721encode`, for which 32 real registers are adequate. We also find that the execution cycles of 32 real registers plus 32 virtual registers are very close to those of 64 real registers for all the benchmarks. On average, while using 64 real registers can reduce the execution cycles by 9.4%, as compared to that of 32 real register only; exploiting 32 virtual registers in addition to 32 real registers can reduce the number of execution cycles by 8.3%. It should be noted that these results are under the assumption that both processors (i.e., with 64 or 32 real registers) have the same frequency. Since the clock cycle time is typically a function of the size of the register file [Farkas et al. 1998]; if we take this into consideration, the scheme with 32 real plus 32 virtual registers may achieve even higher performance than the scheme with 64 real registers, yet having a slower clock frequency.

Fig. 10. Normalized execution cycles by using 32 real registers and by using 16 virtual, in addition to 16 real registers, which are normalized with the execution cycles by using 16 real registers.

5.3 Sensitivity Results

We have also studied the performance of the VLIW processor with 16 real registers, while all other architectural parameters are fixed. We compare the execution cycles of 16 real with that of 32 real registers, as well as that of 16 virtual registers in addition to 16 real registers, which are shown in Figure 10. As can be seen, increasing the number of real registers from 16 to 32 can lead to significant performance increase. On average, the number of execution cycles is decreased by 17.9%. Also, using 16 virtual plus 16 real registers can also achieve much better performance than using 16 real registers only. As can be seen from Figure 10, except a few benchmarks, such as 256.bzip2, the performance of 16 virtual plus 16 real registers is also close to that of 32 real registers. Compared with the results in Figure 9, however, we can see that the performance gap between 16 virtual plus 16 real registers and 32 registers becomes slightly larger than the performance gap between 32 virtual plus 32 real registers and 64 registers. The reason is twofold. First, with 16 real registers, there are more number of register spills (as compared to that of 32 real registers). Second, 16 virtual registers cannot mitigate the register spills as effectively as 32 virtual registers. As can be seen, 16 additional virtual registers still decrease the number of execution cycles by 12.5%, on average, compared with using 16 real registers only. Note that this performance improvement is achieved without increasing the number of real registers or compromising the clock frequency, indicating the effectiveness of the proposed virtual/real register allocation approach.

In this paper, we also investigate the sensitivity of various virtual/real registers combinations, while keeping the total number of registers (including both virtual and real registers) fixed. Since the simulation is very time consuming (especially for SPEC 2000 benchmarks), we select two benchmarks (cjpeg, djpeg) from Mediabench and two (164.gzip and 181.mcf) from SPEC 2000 for this study. We fix the total number of real/virtual registers to be 32, while varying the combination of real/virtual registers to be 8/24, 16/16, and 32/0. Correspondingly, each operand field of these three schemes include 5 bits

Fig. 11. Normalized execution cycles by using 8 real/24 virtual registers, 16 real/16 virtual registers, and 32 real registers, which are normalized with the execution cycles of 16 real registers.

in total, while the number of virtual bits decreases from 2 to 1, and to 0, respectively. The performance results of these three configurations are given in Figure 11, which are normalized with respect to the execution cycles of 16 real registers. As we can expect, using more real registers can always lead to higher performance when the total number of real/virtual registers is fixed, simply because real registers can be allocated to both short-lived and long-lived variables. However, an interesting result is that the scheme of 8 real/24 virtual registers can even achieve better performance than the base scheme of 16 real/0 virtual registers, for 3 out of the 4 benchmarks (i.e., the only exception is 164.gzip). This result indicates that by exploiting virtual registers intelligently, a processor with fewer number of real registers can even achieve better performance than a processor with a greater number of real registers, yet no virtual register, even though we assume that the clock frequency of the latter is not negatively affected. Therefore, the virtual/real register allocation can be very useful for processors that are constrained by the number of real registers to boost performance by minimizing the register spills.

6. CONCLUSION

This paper proposes a novel scheme to reduce register pressure by exploiting the data-forwarding mechanisms widely used in pipelined processors. We propose to add virtual bits to the operand fields of the ISA to create virtual registers that do not have physical storage locations in the register file. These virtual registers can be used to keep track of the data-dependence relations among instructions, making them suitable for being associated with short-lived variables without requesting physical storages in the register file. With the architectural extension to support virtual registers, we modify the region-based register allocation algorithm [Kim 2001] to exploit the virtual registers for the short-lived variables. Our experimental results indicate that virtual registers are very effective at reducing the register spills, which, in many cases, can achieve the

performance close to the processor with twice number of real registers. Also, our results demonstrate that, for some applications, exploiting 24 virtual registers with 8 real registers can attain even higher performance than that of 16 real registers.

While this work focuses on studying virtual registers for statically scheduled processors, we plan to apply virtual registers to dynamically scheduled processors to reduce both the register spills and the pressure on register renaming. In addition, we would like to study the effectiveness of virtual registers in windowed-register or multithreaded architectures, where the register pressure is high, while using a large number of real registers can be prohibitive.

ACKNOWLEDGMENTS

This work was funded in part by NSF grants 0613244 and 0613633. We would like to thank the anonymous referees for the detailed comments that helped us improve the paper.

REFERENCES

- Arm7tdmi data sheet. 1995. In *ARM Ltd.*
- BALASUBRAMONIAN, R., DWARKADAS, S., AND ALBONESI, D. 2001. Reducing the complexity of the register file in dynamic superscalar processors. In *Proc. of MICRO*.
- CALLAHAN, D., CARR, S., AND KENNEDY, K. 1990. Improving register allocation for subscripted variables. In *Proc. of PLDI*.
- CHOW, F. C. AND HENNESSY, J. L. 1984. Register allocation by priority-based coloring. In *Proc. of the ACM SIGPLAN Symposium on Compiler Construction*.
- CRUZ, J. L., GONZALEZ, A., VALERO, M., AND TOPHAM, N. P. 2000. Multiple-banked register file architectures. In *Proc. of ISCA*.
- FARKAS, K. I., CHOW, P., JOUPPI, N. P., AND VRANESIC, Z. G. 1997. The multicluster architecture: reducing cycle time through partitioning. In *MICRO-30*.
- FARKAS, K. I., JOUPPI, N. P., AND CHOW, P. 1998. Register file design considerations in dynamically scheduled processors. In *Proc. of the Fourth IEEE International Symposium on High Performance Computer Architecture*.
- GONZALEZ, A., GONZALEZ, J., AND VALERO, M. 1998. Virtual-physical registers. In *Proc. of HPCA*.
- HU, Z. AND MARTONOSI, M. 2000. Reducing register file power consumption by exploiting value lifetime characteristics. In *Workshop on Complexity-Effective Design*.
- HUCK, J., MORRIS, D., ROSS, J., KNIES, A., MULDER, H., AND ZAHIR, R. 2000. Introducing the ia-64 architecture. In *Micro, Volume 20, Issue 5*, 12–23.
- KANNAN, N., CHIDAMBARAM, P., SUBRAMANIAN, S., AND PARTHASARATHI, R. 2002. Compiler assisted data forwarding in vliw/epic architectures. In *Proc. of the HiPC*.
- KIM, H. 2001. Region-based register allocation for epic architecture. In a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, Department of Computer Science, New York University.
- LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *Proc. of the International Symposium on Microarchitecture*.
- LOZANO, L. A. AND GAO, G. R. 1995. Exploiting short-lived variables in superscalar processors. In *Proc. of MICRO*.
- MAHLKE, S. A., CHEN, W. Y., CHANG, P. P., AND HWU, W. W. 1992. Scalar program performance on multi-instruction-issue processors with a limited number of registers. In *Proc. of the 25th Hawaii International Conference on System Sciences*.
- MONREAL, T., VINALS, V., GONZALEZ, J., GONZALEZ, A., AND VALERO, M. October 2004. Late allocation and early release of physical registers. In *IEEE Transactions on Computers* 53, 10, 1244–1259.

- MUCHNICK, S. S. 1997. Advanced compiler design and implementation. In *Morgan Kaufmann Publishers*.
- OEHMKE, D. W., BINKERT, N. L., MUDGE, T., AND REINHARDT, S. K. 2005. How to fake 1000 registers. In *Proc. of MICRO*.
- PATTERSON, D. A. AND HENNESSY, J. L. 2004. Computer Organization and Design. Morgan Kaufmann San Francisco, CA.
- POSTIFF, M., GREENE, D., RAASCH, S., AND MUDGE, T. N. 2001. Integrating superscalar processor components to implement register caching. In *Proc. of 2001 International Conference on Supercomputing*.
- RUDD, K. W. October 1997. Efficient exception handling techniques for high-performance processor architectures. In *Technical report CSL-TR-97-732*. Coordinated Science Laboratory, Stanford University.
- SAMI, M., SCIUTO, D., SILVANO, C., ZACCARIA, V., AND ZAFALON, R. 2001. Exploiting data forwarding to reduce the power budget of vliw embedded processors. In *Proc. of DATE*.
- SCHLANSKER, M. S. AND RAU, B. R. February 2000. Epic: an architecture for instruction-level parallel processors. In *HPL Technical Report*.
- SMITH, J. E. AND PLESZKUN, A. R. May 1988. Implementing precise interrupts in pipelined processors. In *IEEE Transactions on Computers*, 37, 5.
- SPEC HOMEPAGE. 2006. In <http://www.spec.org>.
- TRIMARAN HOMEPAGE. 2006. In <http://www.trimaran.org>.
- TSENG, J. H. AND ASANOVIC, K. 2003. Banked multiported register files for high-frequency superscalar microprocessors. In *Proc. of ISCA*.
- YUNG, R. AND WILHELM, N. C. 1995. Caching processor general registers. In *Proc. of the International Conference on Computer Design*.
- ZYUBAN, V. V. AND KOGGE, P. 2001. Inherently low-power high-performance superscalar architectures. In *IEEE Trans. on Computers*.

Received November 2006; revised March 2007; accepted April 2007