Energy-Efficient Register Caching with Compiler Assistance

TIMOTHY M. JONES and MICHAEL F. P. O'BOYLE
University of Edinburgh
JAUME ABELLA and ANTONIO GONZÁLEZ
Intel Labs Barcelona—UPC
and
OĞUZ ERGIN
TOBB University of Economics and Technology

The register file is a critical component in a modern superscalar processor. It must be large enough to accommodate the results of all in-flight instructions. It must also have enough ports to allow simultaneous issue and writeback of many values each cycle. However, this makes it one of the most energy-consuming structures within the processor with a high access latency. As technology scales, there comes a point where register accesses are the bottleneck to performance and so must be pipelined over several cycles. This increases the pipeline depth, lowering performance.

To overcome these challenges, we propose a novel use of compiler analysis to aid register caching. Adding a register cache allows us to preserve single-cycle register accesses, maintaining performance and reducing energy consumption. We do this by passing information to the processor using free bits in a real ISA, allowing us to cache only the most important registers. Evaluating the register cache over a variety of sizes and associativities and varying the read ports into the cache, our best scheme achieves an energy-delay-squared (EDD) product of 0.81, with a performance increase of 11%. Another configuration saves 13% of register system energy. Using four register cache read ports brings both performance gains and energy savings, consistently outperforming two state-of-the-art hardware approaches.

Categories and Subject Descriptors: C.1.0 [**Processor Architectures**]: General; C.0 [**Computer Systems Organisation**]: General—*Hardware/software interfaces*; D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms: Experimentation, Measurement, Performance

This work has been partially supported by the Royal Academy of Engineering, EPSRC, the Spanish Ministry of Science and Innovation under grant TIN2007-61763, and the Generalitat de Catalunya under grant 2009 SGR 1250.
T. M. Jones, M. F. P. O'Boyle, and O. Ergin are members of HiPEAC (European Network of Excellence on High Performance and Embedded Architecture and Compilation).
Author's address: T. M. Jones, School of Informatics, 1.12 Informatics Forum, 10 Crichton Street, Edinburgh EH8 9AB, UK; email: tjones1@inf.ed.ac.uk.
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 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.
© 2009 ACM 1544-3566/2009/10-ART13 $10.00
DOI 10.1145/1596510.1596511 http://doi.acm.org/10.1145/1596510.1596511

Additional Key Words and Phrases: Low-power design, energy efficiency, compiler, microarchitecture, register file, register cache

ACM Reference Format:

Jones, T. M., O'Boyle, M. F. P., Abella, J., González, A., and Ergin, O. 2009. Energy-efficient register caching with compiler assistance. ACM Trans. Architec. Code Optim. 6, 4, Article 13 (October 2009), 23 pages. DOI = 10.1145/1596510.1596511 http://doi.acm.org/10.1145/1596510.1596511

1. INTRODUCTION

The physical register file is a critical component in an efficient superscalar processor, allowing out-of-order execution by providing storage space for the results of all in-flight instructions. However, as technology scales, current designs of the register file do not scale well [Agostinelli et al. 2005]. Access latencies become unachievable in one clock cycle, so they have to be pipelined, increasing the branch misprediction penalty and register pressure [Tullsen et al. 1996].

Furthermore, the register file is a hotspot and is already one of the most energy-consuming structures within a modern superscalar processor. Any technique that can reduce the register file's energy requirements would have a significant impact on the processor's total power budget.

We propose to reduce the register file's energy consumption through the addition of a register cache. This also allows us to reduce the port requirements of the register file and, therefore, reduce its access latency. Register caching has been proposed by other researchers [Cruz et al. 2000; Balasubramonian et al. 2001; Borch et al. 2002; Butts and Sohi 2004] to reduce bypass complexity in wide-issue superscalars. They involve placing a buffer containing copies of certain registers near the functional units. It is small enough to be accessed in one cycle, removing the need for pipelined register file access. Register reads are directed to the register cache where they can be serviced quickly. Only if the desired register is not in the cache does the main register file have to be accessed. This means that the number of ports into the register file can be reduced, bringing down the access latency and energy required to read or write a value. A conceptual layout of the register file and register cache is shown in Figure 1.

However, one significant downside of many architectural optimisations is that considerable extra logic is required to keep track of the recent past or hold information about the current in-flight instructions. This logic is generally arranged in table format to provide predictions for the future [Butts and Sohi 2004] or to accumulate knowledge for use later in the pipeline [Monreal et al. 2002]. Supplementing the microarchitecture in this way increases the processor's energy budget, offsetting any performance gains achieved.

To overcome this challenge, this article considers the novel use of simple compiler analysis to aid register caching. This maintains single-cycle register accesses and reduces the energy requirements of the register system (register file and register cache). The compiler can quickly and easily generate detailed information about the complex data dependencies present within a program. Instead of simply throwing this information away once the binary has been created, we propose to make it available to the microarchitecture to allow efficient register caching.

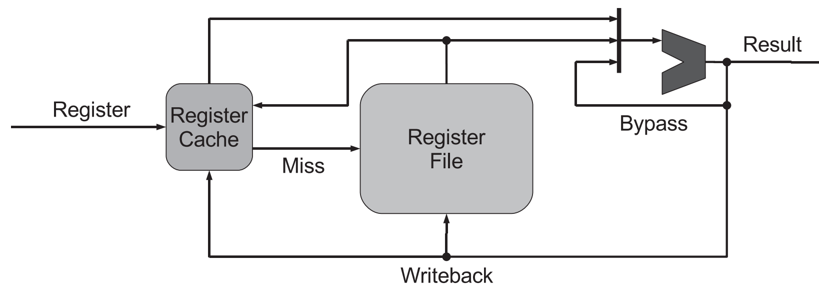


Fig. 1. The conceptual layout of the register system. The register file has only a small number of ports because the majority of reads are serviced by the register cache and bypass network. All generated results are written into both the register file and register cache.

Researchers in the past have used compiler analysis to aid dynamic voltage scaling [Magklis et al. 2003] and early register releasing [Martin et al. 1997; Lo et al. 1999; Jones et al. 2005] based on knowledge of the program's future control or data flow. Our technique, which uses free bits in a real ISA to pass register information to the processor, eliminates the need for the costly extra logic required by state-of-the-art hardware schemes. Performance gains of over 10% can be achieved using a large register cache, leading to an energy-delay-squared (EDD) product of 0.81. On the other hand, energy savings of 13% can be achieved in the register system. In the best case, energy savings can be achieved while maintaining performance increases over the baseline configuration. To summarize our contributions:

—We propose an energy-efficient register-caching scheme that removes the need for costly extra hardware;

—We illustrate a method of incorporating our technique into a real ISA;

—We perform a thorough evaluation of our approach, showing that we outperform two state-of-the-art register-caching schemes [Cruz et al. 2000; Butts and Sohi 2004] across all register cache configurations.

The rest of this article is structured as follows. Section 2 describes previous work on register caching and other related research. The compiler analysis is then described in Section 3. The way in which the processor performs register caching is described in Section 4, then Section 5 details the microarchitecture changes needed to implement our scheme. Section 6 presents our experimental results and, finally, Section 7 concludes.

2. RELATED WORK

Register caching has previously been proposed as an addition to a many-ported or large register file in a wide-issue machine. The aim has been to reduce the complexity of the bypass network, yet still provide back-to-back execution of dependent instructions. Cruz et al. [2000] were some of the first researchers to develop this technique using a multiple-banked register file. Registers are initially read from the register cache, the lower-level register file only being accessed on a miss. In their scheme, returning operands are always written into the lowest level of the hierarchy and are also written into the cache if not

read through bypass. However, entries are only evicted when the cache is full, meaning that some values could be present in the cache even when no longer needed.

Butts and Sohi [2004] introduced a scheme that predicts the number of uses of each register at dispatch based on the defining instruction's PC and some subsequent branch information. This predicted use count is stored with the destination register identifier to determine whether to cache the register after execution. The use count is decremented on each register read. The authors also introduced decoupled indexing to assign a cache set to each destination register.

Both of these schemes [Cruz et al. 2000] and [Butts and Sohi 2004] are considered state-of-the-art approaches, reducing the latency of the register file either by sacrificing a small amount of performance or increasing the register system's energy budget, as demonstrated in Section 6.3.

Another scheme for register caching developed by Balasubramonian et al. [2001] keeps track of register consumers, moving registers to a lower level when all have started execution. In their two-level register file registers are allocated from the first level and moved down to the second when all consumers have started execution. Borch et al. [2002] introduced a register cache at each cluster of functional units from where operands can be read at instruction issue. Zeng and Ghose [2006] developed a register cache consisting of two structures: a FIFO queue for values used over a short duration and a set-associative cache for registers required for a longer period of time. Hu and Martonosi [2000] proposed the Value Aging Buffer, which evicts only the oldest entries each cycle. Finally, Postiff et al. [2001] presented an implementation of a large logical register file combined with a smaller physical register file. However, these schemes require extra hardware tables or additional fields in existing structures, increasing energy requirements.

There have been attempts to simplify the register file by reducing the number of ports it contains. Park et al. [2002] add a bypass hint to each operand in the issue queue to predict whether the value will be read on the bypass network and thus remove the need for a register file read. Tseng and Asanović [2003] present a design for a register file split into banks with only two read ports each. Kim and Mudge [2003] add new buffer structures to hold operands and, finally, Kucuk et al. [2002] present a reorder buffer with no ports for reading operands. However, all these schemes require extra pipeline stages or retention latches, or rely on the majority of values being encoded in fewer than 64 bits.

Other researchers have proposed schemes to reduce register idle time both before results are generated [González et al. 1998] and after the last user has read its value [Lo et al. 1999; Monreal et al. 2002; Ergin et al. 2004, 2006; Jones et al. 2005]. Others have used the width of data to optimize the register file [Aggarwal and Franklin 2003; Ergin et al. 2004; González et al. 2004; Kondo and Nakamura 2005]. All these approaches are orthogonal to the schemes we propose in this article.

This article proposes to use compiler-inferred knowledge about the number of consumers of each register to reduce register file energy and maintain single-cycle read access. We do this by augmenting it with an effective and low-energy register cache which allows us to reduce its read port requirements. Knowing

the number of users enables us to make efficient use of the cache without the need for expensive extra hardware.

3. COMPILER ANALYSIS

This article proposes the use of the compiler to help reduce the register file's energy requirements through register caching. Our scheme relies on the ability to determine the number of consumers that need to read each register value. This section describes how we calculate this within the compiler and pass this information to the processor using free bits in the ISA. Section 4 then describes how this consumer count is used to provide efficient register caching.

Our algorithm uses simple data flow and liveness analysis. The first part iterates over the control flow graph (CFG) to determine the number of uses of each register definition. This is described in Section 3.1, and an example given in Section 3.2. In a second stage, this information is encoded in the free bits available in the ISA. Section 3.3 describes the ISA changes and how the information is encoded in a special no-op when there is not enough space in the particular instruction encoding.

3.1 Counting Register Uses

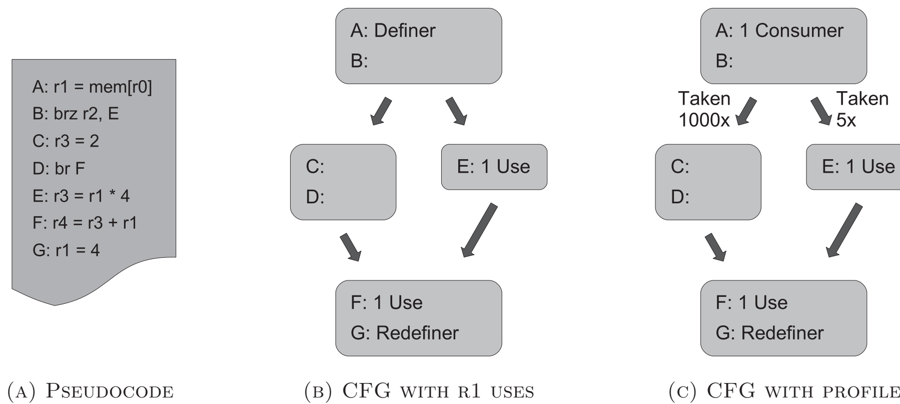
Initially, we estimated the number of register uses statically but, as we show in Section 6.3, this can lead to significant performance degradation in some benchmarks. Therefore, we ran each benchmark using training input and gathered profile information recording the frequency each basic block in the CFG was executed. Using this information the compiler iterates postorder over the CFG gathering the number of uses in or after each node's most frequently executed successor. Each instruction using a register increases that register's consumer count by one. At each register definition, the number of uses is encoded into free bits in the instruction, as described in Section 3.3. For registers with a consumer count greater than six, the value seven is encoded, in effect making seven mean seven-or-more consumers.

3.2 Example

An example of the compiler analysis to count consumers is shown in Figure 2. Figure 2(a) shows some pseudocode defining *r1* in *A*, using it in *E* and *F*, then having it redefined by *G*. The CFG is created in Figure 2(b) and each edge annotated with profile information in Figure 2(c), which is the number of times each branch was taken. The uses are counted along the most frequently executed path and summarized at the defining instruction, *A*. In this example, the path through *C* and *D* is most often taken so there is one use of *r1*, and this is encoded into the instruction format for *A*.

3.3 ISA Impact

Section 6.3 shows that the only efficient way of passing consumer information over to the processor is with ISA modifications. The compiler, therefore, encodes this information into free bits available in the defining instruction's encoding in the ISA. Three bits are needed in each instruction to encode a number of uses



(a) Pseudocode (b) CFG with r1 uses (c) CFG with profile

Fig. 2. An example of the compiler analysis to count the consumers of a register. In (a) seven instructions are shown where B is a conditional branch and D is an unconditional branch. The control flow graph (CFG) in (b) is constructed and each edge annotated with the number of times it was taken when profiling in (c). We count the consumers of r1 backward from G and encode 1 into A as the consumer count.

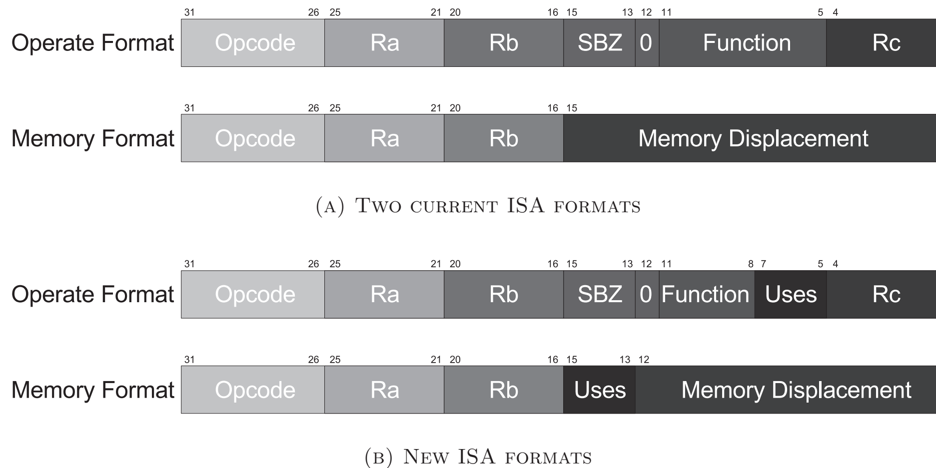
from zero through to seven inclusive. Any register having more than seven uses simply has the value seven encoded.

In Section 6, we have evaluated two schemes. The first is an idealistic approach that assumes every instruction has the necessary 3 bits free in order to encode the number of register uses. The second scheme implements such a scheme in an existing ISA, where we chose that of the Alpha. In this ISA, many instructions already have 3 unused bits in their encoding, such as the operate format that contains unused function codes [Compaq 1998]. For those that do not, such as the memory format, we shorten the immediate or offset field by the required 3 bits. However, the compiler first analyzes the value encoded to see if this shortening is possible. If so, then the number of uses is encoded as normal. If not, then a special no-op containing the actual immediate or offset is placed before the instruction. The processor dispatches the no-op and subsequent instruction atomically, stripping out the value contained in the no-op and attaching this to the following instruction with little overhead. In our experiments, we find that only 6% of the static and 2% of the dynamic instructions require this special no-op.

Figure 3 shows how operate and memory format instructions are altered to accommodate consumer or use counts. In the operate format, the uses can be encoded into free function codes. However, in the memory format instructions, the displacement field has to be shortened and uses encoded into the new free space. Although this implementation is specific to the Alpha, other researchers have found that free bits are common in other ISAs, such as MIPS [Hines et al. 2005].

3.4 Summary

Our compiler pass to enable efficient register caching is based on simple data flow and liveness analysis. We count the number of consumers of each register



(a) Two current ISA formats

(b) New ISA formats

Fig. 3. Two current instruction formats in the Alpha ISA for operate and memory instructions and their proposed alterations. There are enough free function codes in the operate format to encode the number of uses. For memory format instructions, we shorten the displacement field and encode the uses here. A special no-op is provided to allow larger values that no longer fit.

and encode this in free bits from the instruction's encoding. Should there be no free bits to use, we can include a special no-op to help pass this information to the processor. The next section explains how the processor caches registers, then Section 5 describes the microarchitectural changes needed for register caching.

4. REGISTER CACHING

This article proposes simple mechanisms to decide when to place copies of registers in the register cache and when to evict them. As described in Section 3, the compiler calculates the number of consumers for each register which is used to determine whether to cache its value or not. This consumer count is updated by the microarchitecture as each instruction reads the data. Section 4.1 discusses the insertion of registers into the register cache, Section 4.2 describes what happens on a read, and Section 4.3 describes the eviction scheme when the register cache is full. Register cache miss handling is then described in Section 4.4, with Section 4.5 detailing the handling of context switching. Finally, Section 4.6 summarizes our register-caching scheme.

4.1 Inserting Registers

Using the compiler analysis described in Section 3, each destination register is tagged with the number of consumers that will read its value. The processor extracts this count when the defining instruction is dispatched. It is held in the issue queue with the destination register identifier and is used when the instruction writes back to decide whether or not to write the value into the register cache as well as writing into the register file.

The event diagram shown in Figure 4 is used by the processor whenever an operand returns from the functional units to decide whether the value should be

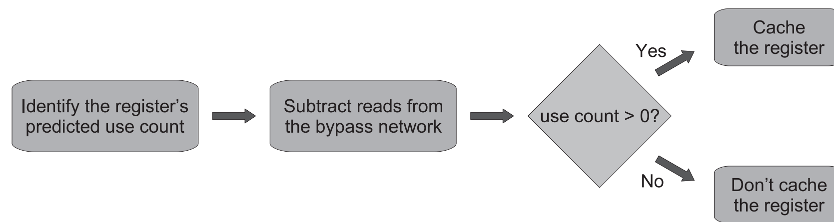


Fig. 4. The event diagram used by the microarchitecture to decide whether to cache a value.

cached or not. First, the compiler's predicted use count is determined, then the number of bypass network reads is subtracted. This gives the predicted number of outstanding consumers that still need to read the register. Registers that now have a predicted consumer count of zero are not cached because there are no predicted future uses of the value. Placing it in the cache would needlessly use an entry, possibly evicting a required value. Using this simple heuristic the microarchitecture can effectively decide whether to cache a returning register.

4.2 Reading from the Register Cache

The compiler provides a prediction of the number of consumers that will read each register, and this consumer count is stored in the register cache along with the copy of the register. Whenever a consumer reads the register cache entry, the consumer count is decremented, providing a count of the remaining consumers. If the read is via the bypass network, it is subtracted from this consumer count before the register is cached.

However, some registers have a very large number of consumers that cannot be directly encoded in the 3 bits allocated for the consumer count. In our scheme, therefore, registers with seven or more consumers (called high-use registers) are given the same consumer count of 7. Since the exact number of such high-usage registers is not known, we do not decrement this value on each read, otherwise the register may be prematurely evicted.

Updating the consumer count when registers are read means that branch mispredictions impact the accuracy of the count. If a mispredicted instruction is squashed after it has decremented one or more consumer counts, these counts will be lower than they should be after the pipeline has recovered from the misprediction. However, this will have only a negligible impact performance, especially given that the branch prediction accuracy of modern processors is extremely high, making this type of event a rarity.

4.3 Evicting Registers

There are just two situations in which registers are evicted from the register cache:

1. Full Register Cache. When the register cache is full, evictions are achieved by considering the remaining consumer count for each cached register in the set from which an eviction should occur. That with the fewest future uses is chosen for eviction and, in the event of a tie, LRU is used. However, as

shown in Section 6, the best configurations are only two-way set-associative, making LRU implementation trivial. If greater associativity is desired, pseudo-LRU could be employed instead, such as that described by Intel Corporation [1998], or even simply a random policy since the eviction choice between entries with the same number of remaining consumers will have little impact on performance. In our experiments with a fully associative, 16-entry register cache, 89% of the time, an empty entry could be found or one corresponding to a register with zero future consumers.

2. Physical Register Released. Register cache entries are marked invalid when the physical register they correspond to is released. This occurs when the instruction redefining the logical register commits, and so this also instigates the release of the register cache entry if it is still valid. To perform this, all tags are checked in the relevant set in the register cache, and if a match occurs, then that entry is marked invalid. This tag check at commit incurs a small energy overhead, which is accounted for in all of our experiments.

4.4 Register Cache Misses—Replaying Instructions

Unfortunately, without a prohibitively large register cache, some register reads will inevitably miss in the cache and have to be serviced by the register file. However, at the time where the miss is detected, subsequent instructions, dependent on that which experienced a miss, may have already issued. This is because the register cache miss will not be detected until the cycle before execution, whereas to provide back-to-back execution of dependent instructions, subsequent instructions must issue in the cycle immediately after their producer (assuming a one-cycle execution latency). These later, dependent instructions must be stalled while the producer reads its source operands from the register file.

One option to deal with this situation would be to simply stall the dependent instructions until the producer had read the register file. However, that would require additional hardware to track the dependents and stall them only, while leaving independent instructions alone. The second option, which we chose, is to simply stop all instructions issuing after that which experienced a miss and allow them to reissue on a future cycle. Although this replaying affects dependent and independent instructions alike, it also makes the additional hardware more simple and is also employed in Butts and Sohi [2004].

4.5 Context Switching

There are two options for dealing with context switching in our register-caching scheme. The first is to include the contents of the register cache in the state that is saved. However, this would increase the overheads of context switching. The second alternative is to invalidate the register cache on a context switch. This would have no effect on the correctness of the program but would incur a very minor performance hit. We have assumed the use of this latter scheme for our register cache in this article.

In terms of dispatch, if there is a special no-op in the instruction stream, the processor needs to ensure that the execution of the no-op and subsequent

instruction are atomic. If a context switch occurs immediately after the no-op, it simply ignores it and restarts fetch again at the no-op once control returns to that process. This is much like the atomic execution of macroinstructions in the x86 ISA. Macroinstructions are split into a number of microinstructions at decode. If execution is interrupted for any reason, instructions are flushed at the macroinstruction level.

4.6 Summary

This section has described how register caching is implemented within the processor. Registers are inserted into the register cache at writeback if they have at least one consumer still needing to read their value. Registers are evicted from the cache when all consumers have read the value or if another register needs to be cached. On a register cache miss, the pipeline is stalled to allow the register to be read from the main register file. Section 5 now describes the microarchitectural changes needed to implement this scheme, and Section 6 presents our experimental results.

5. MICROARCHITECTURE

The registers in our out-of-order superscalar processor are organized into a centralized architectural register file. In addition to this, we provide a register cache to hold copies of some physical registers, as in Figure 1.

The register file does not change in structure from the baseline. However, fewer ports are needed to read values (lowering its access time and energy requirements) since many accesses are satisfied by the register cache. The register file and register cache are arranged, as shown in Figure 1. Both are attached to the functional units with the bypass network but the register file only supplies a value when the miss signal is raised from the register cache. The few values that are read from the register file are written back into the register cache again, if needed. Operands are always written directly into the register file once generated by the functional units, and optionally cached in the register cache.

To support our register-caching scheme, the consumer count must be extracted from each instruction at dispatch and kept with the destination register identifier until writeback. We extend each issue queue entry by three bits to hold this count, which increases the issue queue energy consumption by a negligible amount. The register cache also keeps a copy of this after writeback, but once a value is evicted, the count is lost. Registers written back into the register cache after a miss get a consumer count of zero.

6. RESULTS

This section describes the results obtained for our register-caching schemes. Section 6.1 introduces our experimental infrastructure, and Section 6.2 presents an initial evaluation of our compiler-assisted approach, showing that schemes that do not employ profile information or do not use free ISA bits perform poorly. Section 6.3 then considers reducing the number of read ports required by the register file, since the majority of reads are satisfied by the bypass

Table I. Processor Configuration

Parameter	Configuration	Parameter	Configuration
Machine width	4 instructions	ROB size	96 entries
Branch predictor	16K gshare	LSQ size	48 entries
BTB	2,048 entries, 4-way	Issue queue	32 entries
L1 Icache	32KB 4-way 32B line 1 cycle hit	Int register file FP register file	128 entries 128 entries
L1 Dcache	32KB 4-way 32B line 3 cycle hit	Int FUs	3 ALU (1 cycle), 2 Mul (3 cycles)
Unified L2 cache	2MB 8-way 64B line, 14 cycles hit, 250 miss	FP FUs	2 ALU (2 cycles), 1 MultDiv (4/12 cycles)

network or register cache. Section 6.4 evaluates different register cache sizes and associativities, comparing with two state-of-the-art hardware schemes proposed by Butts and Sohi [2004] and Cruz et al. [2000]. Section 6.5 considers the optimum number of register cache read ports to reduce energy requirements and the EDD product. Finally, Section 6.6 considers using a register cache on different processor configurations. Throughout this section, we use the term register system to mean the register file and register cache for the compiler and Cruz's schemes, the register file, register cache, use-prediction table, and training table for Butts' technique.

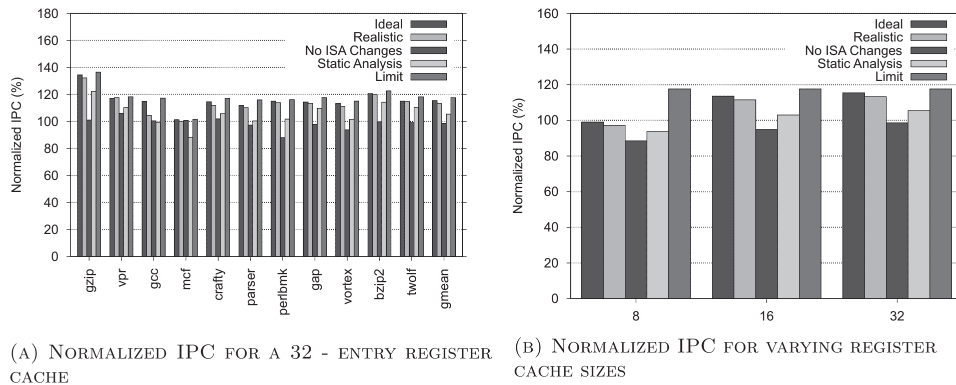
6.1 Compiler, Simulator, and Benchmarks

We wrote our compiler analysis as a pass in MachineSUIF [Smith and Holloway 2000] version 2.02.07.15 and ran our experiments using SimpleScalar [Burger and Austin 1997] version 3.0d and Wattch [Brooks et al. 2000] version 1.02, with register system energy values (both dynamic and static) derived from Cacti [Tarjan et al. 2006] version 4.1. The configuration of the processor we implemented is shown in Table I, which is similar to the Intel Core microarchitecture [Intel 2007; Sandpile 2007]. This is a balanced microarchitecture, representative of modern superscalar processors and allows four instructions to dispatch and issue each cycle, with modestly sized instruction and data caches.

We used the Spec2000 integer benchmark suite for our experiments. We did not use *eon* or the floating point benchmarks because MachineSUIF cannot directly compile them. We trained each benchmark on a complete run with its *training* inputs for profiling. For evaluation, we ran each benchmark with its *reference* input, using SimPoint [Sherwood et al. 2002] to accurately represent each program with an interval size of 10 million instructions and a maximum of 30 clusters per program.

To evaluate our register-caching schemes, we augmented the baseline configuration with a register cache. We experimented with the number of register file ports, size and associativity of the register cache, and the number of read ports it needs to fully explore the design space. Where results are shown in later sections, the configuration of the register cache and register file are described in detail.

For our baseline scheme, we chose a register file with a two-cycle read latency, one level of bypass and no register cache. Our register-caching schemes have



(A) Normalized IPC for a 32 - entry register cache

(B) Normalized IPC for varying register cache sizes

Fig. 5. Normalized IPC when adding a fully associative register cache that exploits compiler analysis to the register file.

a one-cycle register file and only one level of bypass since the register cache is present to service subsequent reads. This is similar to Butts and Sohi [2004] who provide meaningful comparisons with existing research.

6.2 Initial Evaluation

For our initial evaluation of the compiler-assisted approach, we simply added a register cache to the baseline processor. In this section, all cache configurations are fully associative with eight read and four write ports. We considered four different compiler schemes, called *Ideal*, *Realistic*, *No ISA Changes*, and *Static Analysis*. The first assumes there is space in the ISA to encode all consumer counts. The second is implemented using the Alpha ISA, as explained in Section 3.3. The third scheme uses the logical register number to encode the number of uses, similar to the scheme in Jones et al. [2005] and, as, such requires no ISA changes. The final approach does not use any profile information, relying solely on static compiler analysis to determine register-use counts. In addition to this, we present the results of using an idealized register file without a register cache that allows one-cycle register reads, labeled *Limit*. Although unrealistic in practice, this provides an upper limit on the performance achievable.

6.2.1 *Ideal and Realistic.* Figure 5 shows the performance increases for each of the four compiler-assisted register-caching schemes and *Limit* compared to the baseline. Figure 5(a) shows a breakdown of the performance increases for each benchmark when using a register cache containing 32 entries. The addition of this register cache significantly improves performance with most benchmarks running over 10% faster for the *Ideal* scheme. The best improvement comes for *gzip*, which speeds up by 34%. This is due to this benchmark being able to read almost all source registers from the bypass network or register cache in just one cycle, only having to read the register file 0.7% of the time. On the other hand, *mcf* experiences little speed-up, being only 1% faster than the baseline

when using the *Ideal* approach. This benchmark experiences a high number of L2 cache misses, which limits the performance gains achievable using just a register cache.

On the whole, both the *Ideal* and *Realistic* schemes produce good performance improvements across all benchmarks (apart from *mcf*, discussed earlier). The addition of the special no-ops in the *Realistic* scheme only marginally affect performance for the majority of benchmarks. The most affected is *gcc* whose performance gain drops from 15% to just 5%. This is because *gcc* contains many tightly coupled dependencies that are adversely affected by the addition of special no-ops (explained in Section 3.3). These no-ops prevent the full dispatch width being exploited each cycle (because they take up space in the fetch queue), which causes significant performance loss as dependent instructions fail to be issued in consecutive cycles.

It is interesting to note that *vpr* performs better in the *Realistic* compiler scheme compared with the *Ideal* version. Here, the special no-ops take up dispatch resources, but this has a beneficial effect, preventing later instructions from entering the issue queue too early where they are then issued early and cause values to be evicted from the register cache before all consumers have read them. By dispatching certain instructions a cycle later, some registers are present in the register cache slightly longer and hence all consumers can read them before they are evicted. This is an unlikely benefit of implementing the *Realistic* compiler scheme.

Both *Ideal* and *Realistic* are close to *Limit* over all benchmarks. This shows that register caching is a useful technique to mitigate the latency of register file accessing, without requiring an increase in bypass complexity.

6.2.2 *No ISA Changes and Static Analysis.* It is disappointing to see that the other two compiler schemes do not perform well. The *No ISA Changes* approach causes performance drops for half the benchmarks (especially a significant 12% drop for *perlbmk*). This is because there are eight values representing register uses that need encoding and the compiler is often too restricted in its choice of destination registers to be able to select one that represents the correct value. On the other hand, the *Static Analysis* scheme only causes a performance drop in *mcf* (of 12%), but, on average, its speed-ups are a modest 5%, showing that profile information is essential to gain the full performance increases available from register caching.

6.2.3 *Summary.* This section has shown that a register-caching scheme with compiler assistance is beneficial to the baseline architecture in terms of performance. We have evaluated four different compiler schemes, and Figure 5(b) shows the average performance increases for three register cache sizes. As expected, the larger the register cache the greater the performance increases. Also, on average, the *Realistic* scheme achieves slightly less of a performance increase than the *Ideal* version. Furthermore, for a 32-entry and 16-entry register cache, performance of the *Ideal* and *Realistic* schemes is close to *Limit*, that represents a baseline processor with an unrealistic, idealized single-cycle register file.

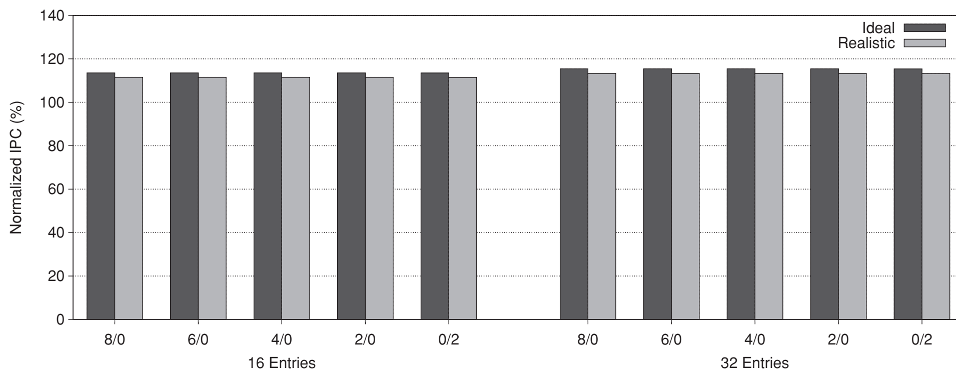


Fig. 6. Normalized IPC when adding a fully associative register cache to the register file. Reducing the number of read ports to the register file does not affect IPC.

In general, we have found that we need to use profiling and modify the ISA to allow the register use information to be accurately passed from compiler to processor. Hence, we only consider the *Ideal* and *Realistic* schemes further. In addition, the performance of the 8-entry register cache is disappointing, so for the rest of this article, we consider only a 16-entry and 32-entry register cache.

In order to achieve register system energy savings, a smaller register file (in terms of area) must be used and different configurations of the register cache employed. The next section considers reducing the register file read ports to reduce its access time and energy costs. Subsequent sections then consider differing configurations of the register cache to produce an energy-efficient register-caching scheme.

6.3 Reducing Ports to the Register File

This section shows that reducing the number of read ports into the register file has little impact on performance because the majority of reads will be satisfied by the bypass network or register cache. All writes must still go to the register file, since the register cache only keeps a copy of some data. In this section, as before, all register caches are fully associative with eight read and four write ports.

The performance impact of reducing the number of read ports to the register file for varying register cache sizes is shown in Figure 6. On the x-axis, each group of bars is labeled with "number of read ports/number of read-write ports" corresponding to the configuration of the register file. It is interesting to see that performance hardly suffers as the number of register file read ports is reduced for both register cache sizes. This shows that almost all register reads can be satisfied by the bypass network or register cache and that just two read-write ports to the register file will suffice. In this configuration, we also reduce the number of write ports correspondingly. For the 16-entry cache, the performance increase is reduced by less than 1% for the *Realistic* version when using two read-write ports. Hence, we chose to use a register file with no read, two write, and two read-write ports.

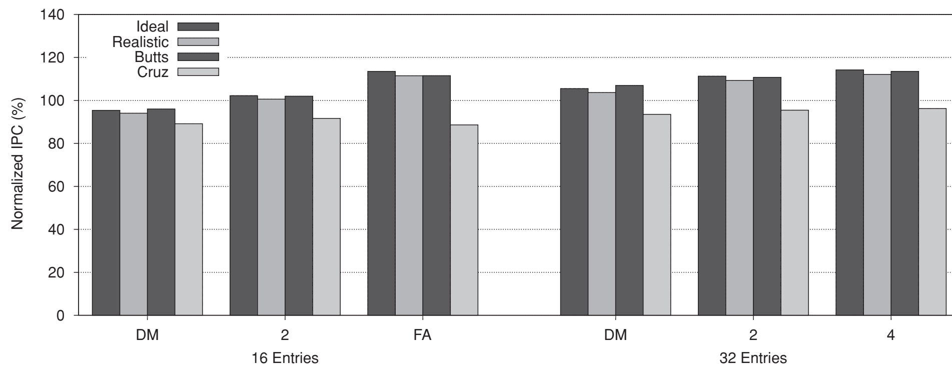


Fig. 7. The impact of varying the associativity of the register cache on performance for two schemes that exploit compiler analysis and two state-of-the-art hardware techniques.

This section has considered reducing the ports to the register file, since the majority of them are not needed when using a register cache. This reduces the register file's dynamic energy consumption to just 12% of the baseline amount and its static energy consumption to just 38% of the original. However, taking into account the register cache's energy contribution, the whole register file architecture experiences significant increases in dynamic and static energy due to the fully associative design of the register cache. Therefore, the next section considers reducing the energy requirements of the register cache by making it set-associative, or even direct-mapped.

6.4 Register Cache Associativity

This section considers a realistic implementation of the register cache by altering its associativity and evaluating the results in terms of performance, energy and EDD values. We also compare our results with schemes proposed by Butts and Sohi [2004] and Cruz et al. [2000]. These techniques are considered state-of-the-art in register caching. We chose Butts' technique (labeled *Butts* in all graphs), since it achieves high performance increases and Cruz's approach (labeled *Cruz*) because it has a low-complexity overhead.

Figures 7, 8, and 9 show the results of varying the associativity of the register cache for two different cache sizes. In this section, the register cache has eight read ports and four write ports, whereas the register file has the best configuration found in Section 6.3 of no read ports, two write ports, and two read-write ports.

6.4.1 *IPC.* In Figure 7, the average performance of each configuration is shown, normalized to the baseline. As expected, increasing the associativity of the register cache increases the overall performance. For all configurations, the *Ideal* compiler scheme has a similar performance to *Butts*, with *Realistic* achieving slightly smaller performance gains. In comparison, the *Cruz* scheme performs poorly, never reaching the baseline's IPC. This is because the authors assumed that on a register cache miss, not all instructions need to be replayed (see Section 4.4), whereas implementation on our processor, which replays all

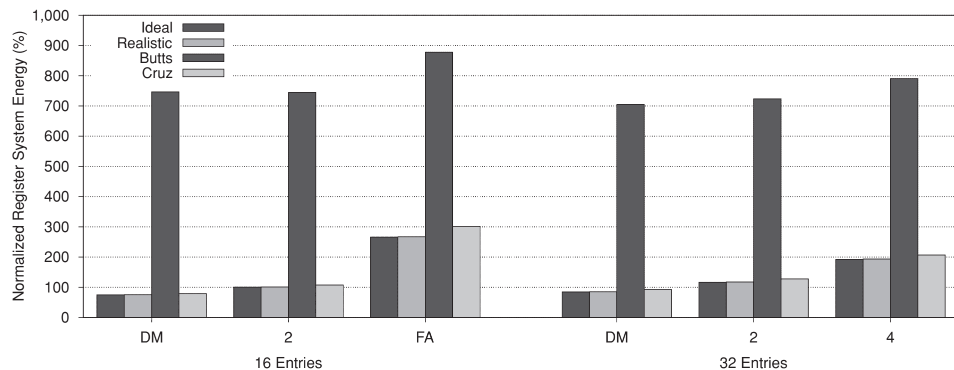


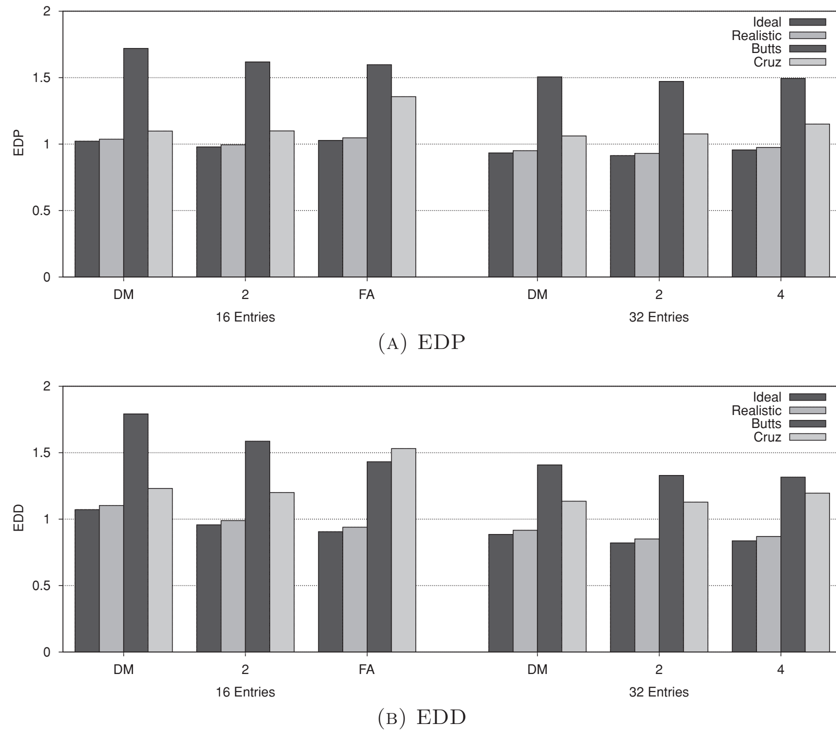
Fig. 8. The impact of varying the associativity of the register cache on register system energy for two schemes that exploit compiler analysis and two state-of-the-art hardware techniques.

instructions following the miss, means they cannot make such effective use of their caching heuristic. Thus, the addition of a register cache in the style of *Cruz* actually harms performance.

The greatest performance gains come from using a four-way, 32-entry register cache. The direct-mapped, 16-entry register cache is the only version in which *Ideal*, *Realistic*, and *Butts* experience performance losses. It is interesting to note that using a fully associative 16-entry register cache provides greater speed-ups than a direct-mapped or two-way, 32-entry version. This shows that a small register cache, configured correctly, can be as useful an addition as a large cache.

6.4.2 *Energy*. Next, we consider the energy impact of using each register cache configuration. Figure 8 shows the register system energy (combined register file and register cache, both dynamic and static energies) normalized to the baseline. It is immediately clear that the state-of-the-art hardware scheme, *Butts*, would present real problems in its implementation, since all configurations of the register cache raise the register system energy by at least 700%. This is due to the overhead of the use-predictor it employs, which is accessed on every register rename to provide a prediction and on every instruction commit to train it. On the other hand, the *Cruz* scheme consumes a similar amount of energy to the compiler-assisted approaches, although always slightly more. This is due to the increase in static energy caused by longer execution.

In comparison, some of the low-associativity caches, when used with the compiler analysis to aid cache management, actually reduce the register system energy consumption. In fact, for both register cache sizes, the direct-mapped setups using our schemes consume less energy than the baseline (75% for 16-entries, 85% for 32 entries with the *Realistic* approach). In addition, the 16-entry, two-way configuration consumes exactly the same as the baseline and the 32-entry, two-way cache consumes only 17% more energy than the baseline. However, the fully-associative register cache is extremely power-hungry and significantly raises the register system's energy consumption.



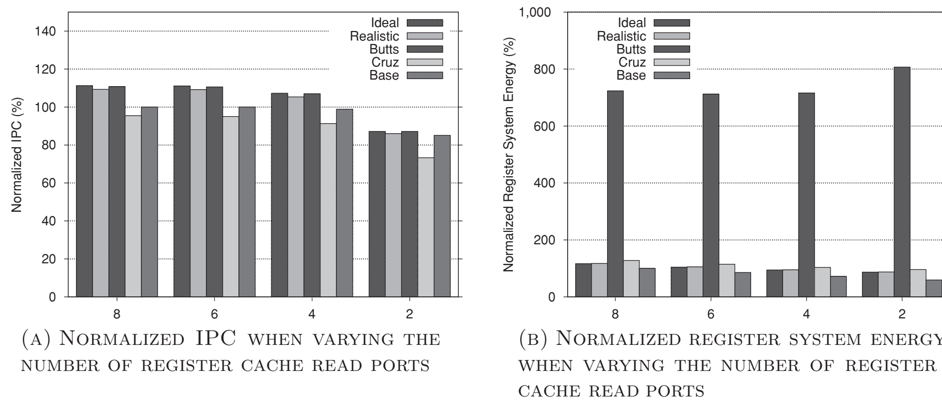
(A) EDP

(B) EDD

Fig. 9. The impact of varying the associativity of the register cache on EDP and EDD for two schemes that exploit compiler analysis and two state-of-the-art hardware techniques.

6.4.3 *EDP and EDD*. Finally, we consider the impact of both performance and energy together using the energy-delay product (EDP) and the EDD product for each cache configuration. These are important metrics in microarchitecture design because they indicate how efficient the processor is at converting energy into speed of operation, the lower the value the better. EDP considers energy and delay to be equally important, whereas EDD, by using the square of the delay, places more emphasis on increasing performance than saving energy, as discussed by Brooks et al. [2000]. For these metrics, we need the trade-off between performance and total processor energy, rather than just register system energy. We conservatively assume that the register system accounts for 10% of the total processor energy budget. A register system accounting for a higher fraction (e.g., in Folegnani and González [2001]) would increase our benefits and be detrimental to the EDP and EDD of *Butts*.

Despite this emphasis on performance, Figure 9 shows that the *Butts* and *Cruz* schemes are not realistic implementations of register caching, since their EDP and EDD values are always above 1, mostly being around 1.3 for EDD, but reaching 1.8 in the most extreme case. The compiler schemes are, in contrast, generally below 1, showing that energy consumed is effectively converted into performance. Although two configurations of the register cache have an EDP above 1 (the direct-mapped and fully associative, 16-entry



(a) Normalized IPC when varying the number of register cache read ports

(b) Normalized register system energy when varying the number of register cache read ports

Fig. 10. The impact of varying the number of register cache read ports on performance and register system energy for a two-way register cache with 32 entries. Using six read ports with the compiler schemes has no impact on performance. Using only two read ports brings energy savings of 13%.

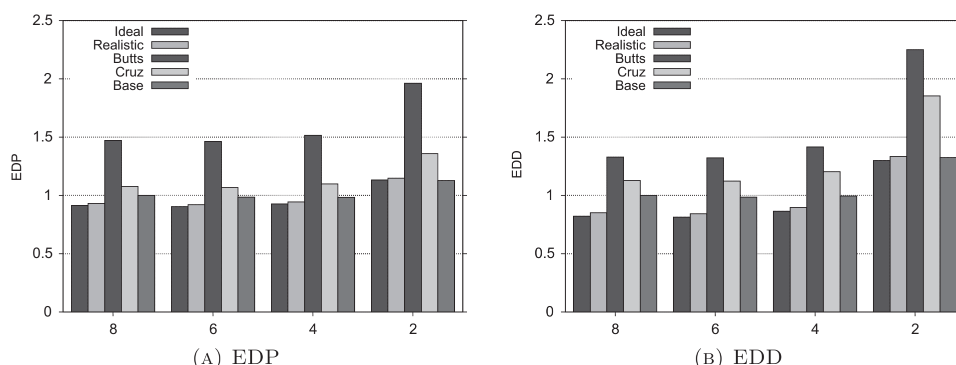
versions) and 1 has an EDD value above 1 (the direct-mapped, 16-entry version), other configurations are below this value. The two-way, 32-entry register cache achieves the lowest EDP and EDD values of 0.92 and 0.82 for *Ideal*, respectively, 0.93 and 0.85 for the *Realistic* scheme.

6.4.4 *Summary*. This section has shown that varying the size and associativity of the register cache using compiler analysis can bring large benefits in terms of register system energy, EDP and EDD values compared with the baseline. State-of-the-art hardware approaches fail to reduce the energy required in the register system, even though they can bring good performance gains, meaning that they are detrimental to EDD. In the next section, we continue to improve the energy-efficiency of the register cache by considering the reduction of read ports into it.

6.5 Reducing Register Cache Read Ports

This section considers reducing read ports once again; however, this time it is the number of read ports required by the register cache, showing that we can achieve energy savings and maintain performance increases over the baseline architecture. We start with the best configuration of the register cache, as determined by the previous section. We find that the lowest EDD value is obtained by the two-way set-associative, 32-entry cache, so focus on this configuration. We adjust the number of read ports to this version of the register cache and deal with port contention in the issue logic at the same time as functional unit availability.

The performance and register system energy for varying register cache read port configurations are shown in Figure 10, normalized to the baseline. Also included, with the title *Base*, are the results for the baseline configuration when the number of register file read ports is also reduced to the same value. This provides a fair comparison with the baseline, whose energy consumption will decrease as the number of read ports is reduced.



(A) EDP (B) EDD

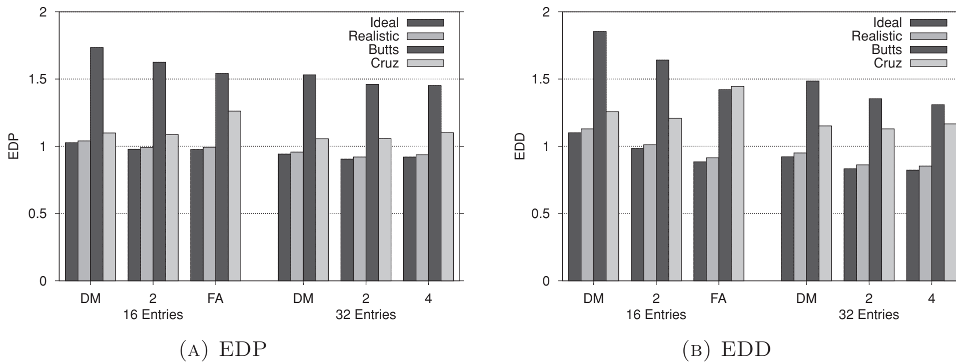
Fig. 11. The impact of varying the number of register cache read ports on EDP and EDD for a two-way register cache with 32 entries. The best EDD value of 0.81 is achieved with six read ports using the compiler schemes.

Performance does not vary much when reducing the number of read ports from eight down to four. This is because it is quite rare that four instructions issue in a cycle, each with two source registers that need to be read. The number of register cache read ports only starts to affect performance when four operands need to be read at issue (1% of the time) but since this is so infrequent, there is only a 4% difference between the six-ported and four-ported register cache. Using only two read ports harms performance considerably, meaning a drop of about 13% for the *Ideal* scheme. However, the two compiler schemes always have better performance than the baseline with the corresponding number of register file read ports.

As the number of read ports to the register cache decreases, so does the energy consumption of the register system. With four read ports, the energy of the *Ideal* and *Realistic* schemes in the register system 94% of the baseline, with a 7% performance increase over the baseline. For *Butts* on the same configuration, the energy consumption is 716%, and for *Cruz*, it is 3% larger than the baseline. Simply reducing the number of register file read ports for the *Base* scheme brings similar energy savings to the register-caching approaches. However, this is at the expense of reduced performance.

Combining the performance and energy gives EDP and EDD values, allowing us to choose the best configuration, as shown in Figure 11. The lowest EDP value is 0.90, obtained with six read ports and the *Ideal* scheme. For the *Realistic* approach, the same configuration achieves 0.92. These configurations also obtain the lowest EDD value of 0.81 for *Ideal* and 0.84 for *Realistic*. Neither *Butts* nor *Cruz* manage to achieve an EDP or EDD value below 1 for any configuration. For the *Base* approach, the performance reductions generally cancel any energy savings. The lowest EDP achieved is 0.98 and the smallest EDD value obtained is 0.99.

6.5.1 *Summary.* Depending on the metric designed for, using the compiler-directed register-caching schemes can bring benefits. For performance, the six-read-ported register cache gains 11% over the baseline and the lowest EDD



(A) EDP (B) EDD

Fig. 12. The impact on EDP and EDD of varying the register cache associativity of two different register cache sizes on a processor with 256 registers.

value of 0.81. For energy, the best configuration is with just two read ports, giving 13% register system energy savings. For both performance gains and energy savings, using four read ports saves 6% of energy and gains 7% performance.

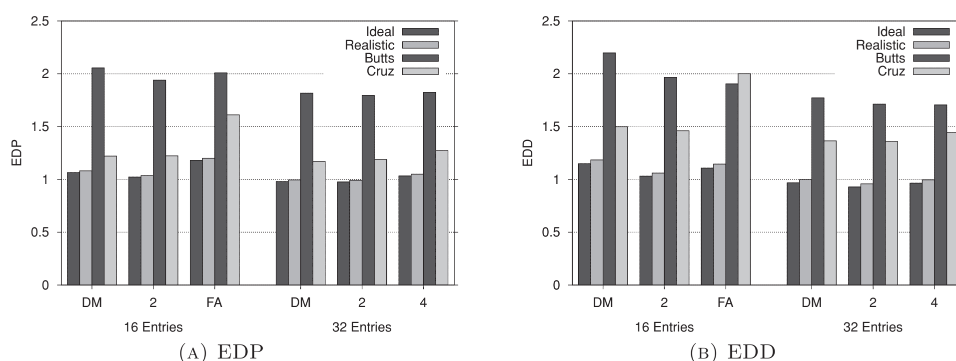
6.6 Application to Other Processors

Our final analysis considers the applicability of our register-caching approach to two different processors, namely one containing 256 registers and another with a two-instruction–wide pipeline. We show that our approach can be beneficial to both processors, achieving performance increases and energy savings. In this section, the register file has no read ports but two read-write ports.

6.6.1 *Larger Register File.* Figure 12 shows the impact on EDP and EDD of the two compiler schemes, *Butts* and *Cruz* when increasing the number of registers from 128 to 256. Here, we have also increased the ROB, IQ, and LSQ proportionally and compared to a baseline of the same configuration but without a register cache. The register cache has eight read ports and four write ports. It is interesting to see that for the compiler schemes, only the direct-mapped, 16-entry register cache performs poorly. All configurations of the 32-entry cache achieve EDP and EDD values of under 1, with the two-way configuration achieving 0.90 (EDP) and 0.83 (EDD) for the *Ideal* scheme. This corresponds to a performance increase of 9% and energy savings of 18%. In contrast, *Butts* and *Cruz* perform poorly across the board, achieving EDP and EDD values of over 1 for all configurations.

6.6.2 *Narrower Pipeline.* Turning our attention to a processor with a narrower pipeline, Figure 13 shows the results of applying our compiler approaches, *Butts* and *Cruz* when the processor has a pipeline width of 2. The rest of the processor configuration is exactly the same as in Table I, and the register cache has three read ports and two write ports.

As Figure 13 shows, our technique can apply equally well to a machine that is less complex than before. Using a 32-entry register cache, we can achieve an EDP value of 0.98 and an EDD value of 0.97 for the direct-mapped version, corresponding to a minor performance increase of 2% and register system



(A) EDP (B) EDD

Fig. 13. The impact on EDP and EDD of varying the register cache associativity of two different register cache sizes on a processor with a pipeline width of 2.

energy savings of 3%. However, as before, *Butts* and *Cruz* are detrimental to the processor, resulting in EDP and EDD values of over 1 for all configurations. These graphs show that our register-caching schemes are applicable across a variety of processor configurations where they can be used to achieve performance gains and maintain energy savings.

7. CONCLUSIONS

This article has presented a novel approach to aid register caching. Through the use of compiler-inferred knowledge about the number of consumers of each register, we have proposed an efficient register cache that increases performance, decreases energy consumption, and decreases the EDP and EDD values of the processor. Our simple compiler analysis determines the number of consumers of each register and embeds this information in the instruction encoding, using free bits in the ISA. We have evaluated two schemes: an idealistic version that assumes all instructions have the necessary free bits and a realistic approach evaluated on the Alpha ISA where special no-ops are used if the instruction cannot be tagged.

We first showed the benefits of register caching, then explored the design space of the register system, reducing read ports to the register file, evaluating the size and associativity of the register cache, then finally reducing the number of read ports to the register cache too.

Using a 32-entry, two-way register cache with four read ports, we show that our realistic-caching scheme outperforms two state-of-the-art hardware techniques, bringing performance gains of 7%, energy savings of 6%, and an EDD value of 0.86. In comparison, the two hardware approaches have EDD values of over 1, experiencing either performance losses or significant increases in energy over the baseline.

REFERENCES

Aggarwal, A. and Franklin, M. 2003. Energy efficient asymmetrically ported register files. In *Proceedings of the 21st International Conference on Computer Design (ICCD'03)*. IEEE, Los Alamitos, CA.

AGOSTINELLI, M., HICKS, J., XU, J., WOOLERY, B., MISTRY, K., ZHANG, K., JACOBS, S., JOPLING, J., YANG, W., ET AL. 2005. Erratic fluctuations of SRAM cache Vmin at the 90nm process technology node. In *Proceedings of the IEEE Electron Devices Meeting*. IEEE, Los Alamitos, CA.

BALASUBRAMONIAN, R., DWARKADAS, S., AND ALBONESI, D. H. 2001. Reducing the complexity of the register file in dynamic superscalar processors. In *Proceedings of the 34th International Symposium on Microarchitecture (MICRO'01)*. IEEE, Los Alamitos, CA.

BORCH, E., MANNE, S., EMER, J., AND TUNE, E. 2002. Loose loops sink chips. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, Los Alamitos, CA.

BROOKS, D., TIWARI, V., AND MARTONOSI, M. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA'01)*. ACM, New York.

BROOKS, D. M., BOSE, P., SCHUSTER, S. E., JACOBSON, H., KUDVA, P. N., BUYUKTOSUNOGLU, A., WELLMAN, J. D., ZYUBAN, V., GUPTA, M., AND COOK, P. W. 2000. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro 20*, 6.

BURGER, D. AND AUSTIN, T. 1997. The simplescalar tool set, version 2.0. Tech. rep. TR1342, University of Wisconsin-Madison.

BUTTS, J. A. AND SOHI, G. S. 2004. Use-based register caching with decoupled indexing. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA'04)*. ACM, New York.

COMPAQ 1998. Alpha Architecture Handbook.
http://www.compaq.com/cpq-alphaserver/technology/literature/alphaahb.pdf

CRUZ, J.-L., GONZÁLEZ, A., VALERO, M., AND TOPHAM, N. P. 2000. Multiple-banked register file architectures. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA'00)*. ACM, New York.

ERGIN, O., BALKAN, D., GHOSE, K., AND PONOMAREV, D. 2004. Register packing: Exploiting narrow-width operands for reducing register file pressure. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO'04)*. IEEE, Los Alamitos, CA.

ERGIN, O., BALKAN, D., PONOMAREV, D., AND GHOSE, K. 2004. Increasing processor performance through early register release. In *Proceedings of the 22nd International Conference on Computer Design (ICCD'04)*. IEEE, Los Alamitos, CA.

ERGIN, O., BALKAN, D., PONOMAREV, D., AND GHOSE, K. 2006. Early register de-allocation mechanisms using check-pointed register files. *IEEE Trans. Comput. 55*.

FOLEGNANI, D. AND GONZÁLEZ, A. 2001. Energy-effective issue logic. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA'01)*. ACM, New York.

GONZÁLEZ, A., GONZÁLEZ, J., AND VALERO, M. 1998. Virtual-physical registers. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA'98)*. IEEE, Los Alamitos, CA.

GONZÁLEZ, R., CRISTAL, A., ORTEGA, D., VEIDENBAUM, A., AND VALERO, M. 2004. A content aware integer register file organization. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA'01)*. ACM, New York.

HINES, S., GREEN, J., TYSON, G., AND WHALLEY, D. 2005. Improving program efficiency by packing instructions into registers. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05)*. ACM, New York.

HU, Z. AND MARTONOSI, M. 2000. Reducing register file power consumption by exploiting value lifetime. In *Proceedings of the Workshop on Complexity Effective Design (WCED) in Conjunction with the 27th International Symposium on Computer Architecture (ISCA'00)*. ACM, New York.

INTEL. 2007. Intel Core Microarchitecture.
http://www.intel.com/technology/architecture-silicon/core/index.htm

INTEL CORP. 1998. Embedded Pentium Processor Family Developer's Manual.
http://developer. intel.com/design/intarch/MANUALS/241428.htm

JONES, T. M., O'BOYLE, M. F. P., ABELLA, J., GONZÁLEZ, A., AND ERGIN, O. 2005. Compiler directed early register release. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. ACM, New York.

KIM, N. S. AND MUDGE, T. 2003. The microarchitecture of a low power register file. In *Proceedings of the International Symposium on Low-Power Electronics and Design (ISLPED'03).* ACM, New York.

KONDO, M. AND NAKAMURA, H. 2005. A small, fast and low-power register file by bit-partitioning. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05).* IEEE, Los Alamitos, CA.

KUCUK, G., PONOMAREV, D., AND GHOSE, K. 2002. Low-complexity reorder buffer architecture. In *Proceedings of the 16th International Conference on Super-Computing (ICS'02).* ACM, New York.

LO, J. L., PAREKH, S. S., EGGERS, S. J., LEVY, H. M., AND TULLSEN, D. M. 1999. Software-directed register de-allocation for simultaneous multithreaded processors. *IEEE Trans. Paral. Distrib. Syst. 10*, 9.

MAGKLIS, G., SCOTT, M. L., SEMERARO, G., ALBONESI, D. H., AND DROPSHO, S. 2003. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA'03).* ACM, New York.

MARTIN, M. M., ROTH, A., AND FISCHER, C. N. 1997. Exploiting dead value information. In *Proceedings of the 30th International Symposium on Microarchitecture (MICRO).* IEEE, Los Alamitos, CA.

MONREAL, T., VIÑALS, V., GONZÁLEZ, A., AND VALERO, M. 2002. Hardware schemes for early register release. In *Proceedings of the International Conference on Parallel Processing (ICPP'02).* IEEE, Los Alamitos, CA.

PARK, I., POWELL, M. D., AND VIJAYKUMAR, T. N. 2002. Reducing register ports for higher speed and lower energy. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO'02).* IEEE, Los Alamitos, CA.

POSTIFF, M., GREENE, D., RAASCH, S., AND MUDGE, T. 2001. Integrating superscalar processor components to implement register caching. In *Proceedings of the 15th International Conference on Super-Computing (ICS'01).* ACM, New York.

SANDPILE. 2007. Ia-32 implementation. Intel Core. http://www.sandpile.org/impl/core.htm.

SHERWOOD, T., PERELMAN, E., HAMERLY, G., CALDER, B. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02).* ACM, New York.

SMITH, M. D. AND HOLLOWAY, G. 2000. The Machine-SUIF documentation set. http://www.eecs. harvard.edu/machsuif/software/software.html.

TARJAN, D., THOZIYOOR, S., AND JOUPPI, N. P. 2006. Cacti 4.0. Tech. rep. HPL-2006–86, HP Laboratories Palo Alto.

TSENG, J. H. AND ASANOVIĆ, K. 2003. Banked multiported register files for high-frequency superscalar microprocessors. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA'05).* ACM, New York.

TULLSEN, D. M., EGGERS, S. J., EMER, J. S., LEVY, H. M., LO, J. L., AND STAMM, R. L. 1996. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA'96).* ACM, New York.

ZENG, H. AND GHOSE, K. 2006. Register file caching for energy efficiency. In *Proceedings of the International Symposium on Low-Power Electronics and Design (ISLPED'06).* ACM, New York.

Received December 2008; revised March 2009; accepted April 2009